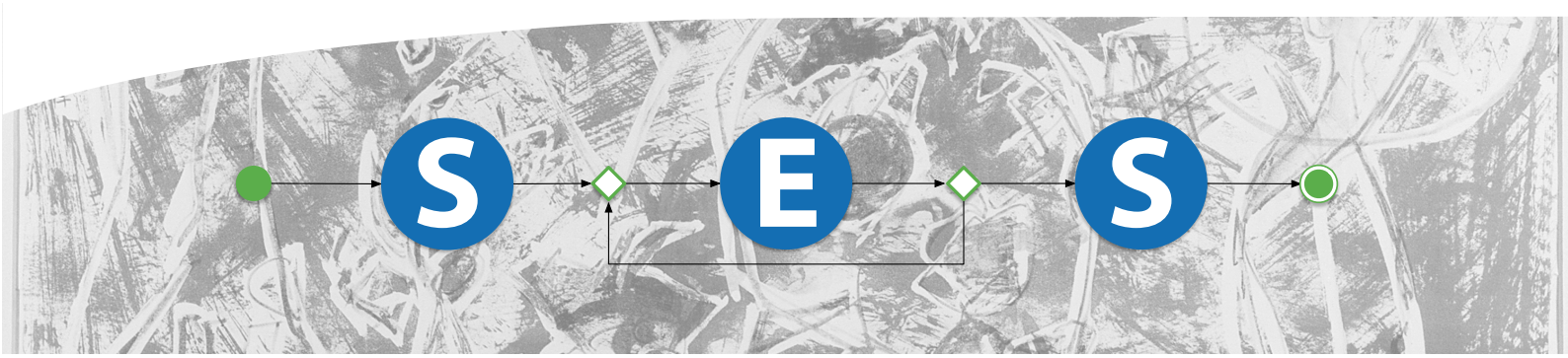
CARL VON OSSIETZKY universität OLDENBURG

# SOFTWARE EVOLUTION SERVICES

## A Framework for the Integration and Development of Flexible and Reusable Toolchains




Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften
der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels

*Doktors der Ingenieurwissenschaften (Dr-Ing.)*

angenommene Dissertation von Herrn

**Jan Jelschen**

geboren am 20. Oktober 1983 in Oldenburg.

| | |
|---|---|
| Gutachter: | Prof. Dr. Andreas Winter |
| Weitere Gutachter: | Prof. Dr. Oliver Theel,<br>Prof. Dr. Rainer Koschke |
| Tag der Disputation: | 9. Oktober 2023 |

# Abstract

The growing size and complexity of software systems, as well as a constant pressure to adapt them to address changing requirements and keep up with fast-paced technological advancements, demands their decomposition into, and assembly from, smaller, more manageable parts, and the reliance on existing off-the-shelf components. Therefore, software *integration* has become a critical challenge in software development. A particular instance of this challenge presents itself in the field of software evolution, where the modernization of large legacy software systems calls for tailored, integrated tool-support able to adapt quickly to shifting requirements.

This thesis presents SENSEI (*Software EvolutioN SErvices Integration*), a framework and method for creating flexible integrated toolchains and software systems.

To derive the solution, this thesis establishes increased *flexibility*, *reusability*, and *productivity* as its key objectives. Based on an analysis of the toolchain-building process in software evolution, and a comprehensive review of the field of tool integration and existing, related integration approaches, the focus of the thesis is further refined, and the scope of SENSEI clearly demarcated. Component-based, service-oriented, and model-driven software engineering paradigms are reviewed and identified as suitable foundations, and combined with novel concepts to form the SENSEI framework.

At the core of SENSEI lies a clear separation of concerns, distinguishing *services* that abstract from technologies and interoperability issues on the one hand, and concrete implementations in the form of *components* on the other hand. It enables domain experts to model processes to be automated as *orchestrations* of services, which are chosen from a standardizing catalog, and are technology-agnostic, abstracting interoperability issues away. Developers of individual components or tools provide service-conforming adapters and enter their implementations into a component registry. The *capability model*, a distinguishing feature of the approach, allows the declarative specification of required and provided capabilities, which is leveraged by SENSEI to automatically match services to implementing components. Using model-driven tooling, SENSEI then automatically generates an integrated, executable solution.

SENSEI confers *flexibility* due to the abstraction from technical aspects, which makes changing orchestrations easy, and its ability to simply discard and quickly regenerate integration code. The structures it establishes promote *reuse* of both individual components, and interface adapters and data transformers, which are explicitly separated from the rest of the integration code. Separation of concerns, elimination of accidental complexities, a structuring framework interlinked by capabilities, and automation substantially reduce the required effort for integration and evolution, and thus yield increased *productivity*.

The feasibility of the approach and its benefits have been demonstrated by practical application in two different case studies: In the first study, the SENSEI method and framework has been used to model and integrate the extensive tool support of a software evolution project. In the second study, SENSEI was used to model the business processes of a software application, map them to appropriate components and automatically integrate them.

# Kurzfassung

Die zunehmende Größe und Komplexität von Softwaresystemen, sowie der ständige Druck, sie an sich ändernde Anforderungen anzupassen und mit dem rasanten technologischen Fortschritt mitzuhalten, erfordern deren Zerlegung in handhabbarere Teilbausteine und den Rückgriff auf Standardkomponenten. *Softwareintegration* ist daher zu einer entscheidenden Herausforderung in der Softwareentwicklung geworden. Ein konkretes Beispiel findet sich im Bereich der Software-Evolution: hier werden für die Modernisierung von Altsystemen maßgeschneiderte, integrierte Werkzeugketten benötigt, die sich leicht an wechselnde Anforderungen anpassen lassen.

Die vorliegende Arbeit präsentiert SENSEI (*Software EvolutioN SErvices Integration*), ein Framework und eine Methodik zur Erstellung flexibler integrierter Werkzeugketten und Softwaresysteme.

Die Erarbeitung des Lösungsansatzes leiten die Kernziele *Flexibilitäts-*, *Wiederverwendbarkeits-* und *Produktivitätssteigerung*. Es erfolgt eine weitere Fokussierung durch die Analyse des Vorgehens bei der Erstellung von Werkzeugketten für die Software-Evolution, und eine klare Positionierung und Abgrenzung anhand einer umfassenden Literaturrecherche von Arbeiten zur Werkzeugintegration und verwandter Integrationsansätze. Hieraus werden komponentenbasierte, serviceorientierte und modellgetriebene Softwareentwicklungsparadigmen als geeignete Grundlagen abgeleitet, erarbeitet und mit eigenen Konzepten zum SENSEI-Framework kombiniert.

Im Mittelpunkt von SENSEI steht eine klare Trennung der Belange, bei der strikt zwischen *Services*, die von technischen Interoperabilitätsaspekten abstrahieren, und implementierenden *Komponenten* unterschieden wird. Der Ansatz ermöglicht es Fachanwendern, zu automatisierende Prozesse als *Orchestrierungen* technologieunabhängiger Services zu modellieren, die aus einem der Standardisierung dienenden Katalog ausgewählt werden. Entwickler einzelner Komponenten stellen Service-konforme Adapter zur Verfügung und tragen ihre Implementierungen in ein Komponentenverzeichnis ein. Ein Alleinstellungsmerkmal des Ansatzes ist das *Capability-Model*, das die deklarative Angabe benötigter bzw. bereitgestellter Eigenschaften ermöglicht, und der automatischen Zuordnung von Services zu geeigneten Komponenten dient. Mittels modellgetriebener Codegenerierung erzeugt SENSEI automatisch eine ausführbare, integrierte Anwendung.

*Flexibilität* erreicht SENSEI mittels Abstraktion von technischen Belangen, was Orchestrierungen leicht veränderbar macht, und durch die Möglichkeit, Integrationscode jederzeit neu generieren zu können. Durch strukturelle Vorgaben und die Trennung vom übrigen Integrationscode wird die *Wiederverwendbarkeit* von Komponenten, Schnittstellenadaptern und Datentransformatoren erhöht. Die Trennung von Belangen, die Beseitigung unnötiger Komplexität, ein durch *Capabilities* verknüpftes Strukturgerüst und die Automatisierung reduzieren den erforderlichen Integrations- und Entwicklungsaufwand erheblich und führen somit zu einer *Produktivitätssteigerung*.

Die Umsetzbarkeit des Ansatzes und seine Vorteile wurden durch die praktische Anwendung in zwei Fallstudien nachgewiesen: In der ersten Studie wurde die SENSEI-Methodik und das Framework verwendet, um die umfangreiche Werkzeugunterstützung für ein Software-Evolutionsprojekt zu modellieren und zu integrieren. In der zweiten Studie wurden mit SENSEI die Geschäftsprozesse einer Softwareanwendung modelliert, den entsprechenden Komponenten zugeordnet und automatisch integriert.

# Brief Contents

# Contents

# PART I

# Challenges

This thesis presents SENSEI (*Software EvolutioN SErvices Integration*), a framework and method for creating flexible integrated toolchains and software systems. The work is presented in five parts: Part I – Challenges, Part II – Analysis, Part III – Key Technologies, Part IV – Solution, and Part V – Evaluation. This, the first part, aims at establishing the context of the thesis, its motivation and problem statement, its relevance, and its objectives. The second part aims at further refining the objectives by analyzing the problem statement and reviewing existing work and related approaches. The third part describes component-based, service-oriented, and model-driven software engineering, whose techniques form the cornerstones of the approach. The fourth part describes the solution derived from these insights. The fifth part presents practical applications, and reflects on the adequacy and quality of the solution with respect to the original objectives.

Part I consists of two chapters: Chapter 1 introduces the context, describes the challenges, provides evidence of their practical relevance, and derives a clear problem statement and objectives. It concludes with an overview of the outline of the thesis. Chapter 2 presents the *Q-MIG* software quality and migration project as a motivating example, that will be used throughout the thesis.

CHAPTER 1

# Introduction

Software systems used and being developed today are expected to automate tasks and solve problems so extensive and complex that developing them completely from scratch is usually completely infeasible. These limitations were first felt in the nineteen-sixties, described as *software crisis*, and gave rise to the field of *software engineering* [Naur and Randell, 1968]. To overcome these challenges, McIlroy [1968] called for the assembly of applications from prefabricated **software components**. Since then, research and practice has created software engineering principles and paradigms like object orientation [Dahl, 2004; Kay, 1993], component-based software engineering [Szyperski, 1997], and service orientation [Erl, 2005], to counteract size and complexity with structure, abstraction, and reuse.

This approach to building software systems has given rise to a new challenge, the **need to integrate** individual pieces of software into a coherent whole. Due to a lack of interoperability, this is often a demanding task, introducing its own complexities. In fact, there may now be an *integration crisis* [Manes, 2003], with Gorton, Thurman, and Thomson [2003] citing estimates of up to 70 percent of "information technology spending" due to integration efforts.

Aggravating this issue is the high speed of technical progress and shifting requirements, putting software systems under constant **pressure to evolve** [Lehman, 1980, 1996]. By the early eighties, this had already eroded many large software systems to the point of having become technically outdated and hard to maintain, yet still fulfilling indispensable tasks, and being too expensive to simply discard and replace [Brodie and Stonebraker, 1995]. A study by Lientz and Swanson [1980] showed that, on average, about half of software costs were due to maintenance. This *legacy crisis* [Seacord, Plakosh, and Lewis, 2003] persists to this day [Broy, 2018], and has spawned extensive research in areas such as software reengineering, migration, reverse engineering, program understanding, analysis, and more, often subsumed now as *software evolution*.

Software integration today is approached with different tools and techniques in different areas, ranging from ad-hoc, UNIX-style pipes and filters or using scripting languages [Sanner, 1999], to enterprise application integration based on heavy-weight middleware, service orientation, and open standards efforts [Land and Crnkovic, 2004]. Few, if any, approaches directly account for the **need for flexibility** to rapidly modify and extend integrated systems. Agile methods such as Extreme Programming [Beck, 2004] or Scrum [Schwaber and Beedle, 2002], meant to bring about such flexibility on the process level, might actually have adverse effects on evolvability in the long run, due to their shunning of foresight and up-front architectural design in favor of "You ain't going to need it" (YAGNI) [Boehm, 2002; Knodel and Naab, 2014; Kruchten, 2010].

An area with a particularly pronounced integration challenge are software evolution tools [Ghezzi and Gall, 2013; Jin and Cordy, 2005b; Müller et al., 2000; Sim, 2000]. A high degree of automation through **integrated toolchain support** is an absolute prerequisite for large software reengineering and migration projects [Borchers, 1996] to succeed, and a huge body of individual tools exists, yet with little to no means for interoperability. The need for experimentation, and the ability to react quickly to new insights into the legacy systems being modernized, demand a *flexible toolchain*, which a one-off, hard-wired integration of technically highly disparate tools is scarcely able to provide. Ironically, the very software intended to improve evolvability falls victim to the same issues that turned the systems under study into legacy in the first place.

This thesis presents **SENSEI** – *Software EvolutioN SErvices Integration*, an approach for *service-oriented specification and semi-automatic integration of flexible software solutions from reusable components*. The driving motivation behind its creation has been toolchain-building for software evolution projects. But, since many of the challenges are the same when integrating software applications in general, the approach will be shown to be applicable beyond software evolution toolchains, as well. The titular **Software Evolution Services** refer to the elementary unit of functionality used in SENSEI for specification, and allude to both the application domain of software evolution, as well as the imparted flexibility facilitating continued evolution.

In the remainder of this chapter, the need for a toolchain-building support framework in the field of software evolution in particular, but also for similarly process-centric integration challenges in software development in general, are motivated in Section 1.1. Section 1.2 derives and defines the main objectives of the thesis. The chapter closes with an outline of the remainder of the thesis in Section 1.3.

## 1.1 Integration Challenges in Software Evolution and Beyond

Modernizing legacy systems is, due to their size and complexity, only feasible with a high degree of automation. Every software evolution project has individual modernization goals, and is as different as the legacy systems to be modernized. Each project requires a combination of different software evolution techniques, e.g. for anal-

ysis, reverse engineering, measurement, and visualization, as well as transformation, reengineering, and migration. These techniques have to be tool-supported, and the tools in turn have to be integrated into a tailor-fit toolchain. Thus, a major challenge of the field is the provision of *integrated* tool support, as recognized by both scientists working in the field of software evolution, as well as experts from industry:

- Borchers [1996] emphasizes the importance of establishing manufacturing processes to achieve high levels of automation, which are necessary to realize reengineering projects timely and cost-efficient. He proposes the creation of **reengineering factories**: integrated toolchains to automate the planned reengineering activities, tailored to project-specific needs based on a specification "*cookbook*".
- Bergey et al. [1999] confirm this industry insight by reporting on a reengineering project to be supported by a toolchain, integrated "in a seamless way". While the individual tools were tried and tested, the required **effort for integration was gravely underestimated**, leading to loss of "time and millions of dollars".
- Müller et al. [2000] include the lack of tool interoperability in reverse engineering as a major challenge in the field, besides a lack of adoption and awareness of tools. They criticize existing reverse engineering tools for building closed systems and ignoring interoperability. They contrast this with tools designed according to the **Unix philosophy** [Pike and Kernighan, 1984], which are kept simple, but can be combined to solve more complex tasks.
- Sim [2000] acknowledges the development of standard exchange file formats (like the *Graph Exchange Language* GXL [Holt, Winter, and Schürr, 2000]), but points out the slowness of toolchains integrated on this level, which **impedes necessary experimentation**. She advocates tool interoperability on a higher level, and discusses, among other approaches, using component-based technologies.
- Mens et al. [2005] argue, from an academic research perspective, that the tools and toolchains required to handle the software evolution challenges posed by industry-scale legacy software systems are too complex to be created by individual research groups. They propose a **common platform** or application framework to base interoperable tools on.
- Jin and Cordy [2005b] assert "a very poor record of interoperability" of software analysis and reengineering tools, and distinguish between prescriptive and non-prescriptive integration, the former enforcing the adoption of special APIs or standard exchange file formats. They propose **services as a means for abstraction**, and ontologies for data description and integration.
- Sneed, Wolf, and Heilmann [2010, pp 171f] underline the indispensability of tool support for software migration projects, and the **necessity of using individual tools effectively together**. In essence, an inverse proportionality between the degree of automation by a proper tool infrastructure, and the riskiness of the project is suggested, with full automation resulting in an almost risk-free project.

- Ghezzi and Gall [2013] point out difficulties in combining different software analysis tools, and highlight the area of empirical studies, where a lack of tool interoperability impedes the ability to **replicate empirical research**. The authors propose a full-fledged approach called *SOFAS*, based on ("RESTful") services and ontologies, as well. SOFAS will be described in more detail in Section 5.6.
- Rajlich [2014] provides a recent review of the state of research and practice, and future challenges and directions, in software evolution and maintenance. He identifies the "need for a *seamless software environment*" as a major area for future work, and confirms that the "lack of tool integration [is] a major hindrance". This shows that **tool integration is still unresolved**, and thus remains an important research topic.

In summary, research and industry agree on a **persistent need for more efficient tool integration that allows for flexibly evolving toolchains able to reuse existing tools**.

Academia has made steady but slow progress in tool integration research, focussing instead on developing and refining a large variety of techniques and tools to analyze, reverse engineer, transform, and visualize software systems. As for industrial software evolution projects, specialized consultancies are usually contracted, of which there are only a few in the market; in this situation, they have no incentives to provide interoperability beyond their own tools.

Examples of concrete challenges arising during toolchain-building for software evolution projects are given in Chapter 2, which will introduce Q-MIG, a research project with industry participation concerned with software quality analysis and software migration, which SENSEI has been applied to (Chapter 15).

Beyond software evolution, software system integration is becoming more and more important as well, as architectural paradigms have shifted from monolithic applications to service-oriented architecture, and particularly microservices [Cerny, Donahoo, and Trnka, 2018; Jamshidi et al., 2018; Newman, 2015] in recent years, with deployments to cloud infrastructures. In general, the increasing interconnectedness of software-intensive systems, driven by trends variously subsumed as the *Internet of Things* (IoT) [Atzori, Iera, and Morabito, 2010], *Industry 4.0*, and *cyber-physical systems* [Lee, Bagheri, and Kao, 2015] leads to their "rapidly growing complexity coupled with an unprecedented increase in scale" [Ebert, Hoefner, and Mani, 2015]. The interoperability of individual software parts and their integration are naturally among the most pressing challenges [Zimmermann, 2017].

The scope of this thesis is, first and foremost, defined by the toolchain-building problem in software evolution. However, an important aspect of software system integration in general is largely analogous to this: mapping business processes onto an application landscape that provides the necessary services, and deriving integration solutions. This transferability of the approach developed in this thesis will be shown in Chapter 16. Its precise objectives are defined in the following Section 1.2, and its scope will be further refined in Part II.

## 1.2 Objectives

**Software Evolution Services** introduce a conceptual abstraction layer over the tools to be integrated, which only views their provided functionality – its *services*, but hides interoperability issues due to different implementation technologies and data formats.

The **core objective** of this thesis is the creation of a support framework for the complete toolchain creation process,to increase flexibility, facilitate reuse, and thereby improve overall productivity.

The main contribution of this thesis is SENSEI ("*Software EvolutioN SErvices Integration*"), which is just such a framework, created around the idea of software evolution services. It is motivated by the need for tool integration in software evolution projects, but not necessarily limited to this application area. The core objective names three sub-objectives: 1. **increasing flexibility**, 2. **increasing reusability**, and 3. **increasing productivity**, described in more detail in the following. The *toolchain creation process* will be described and analyzed in Chapter 3 to further refine the core objective.

### 1.2.1 Increasing Flexibility

Like regular software projects, software evolution projects often require an iterative process (e.g. SOAMIG [Fuhr et al., 2012]). Requirements cannot usually be elicited up-front, completely. Although for migration and reengineering projects, at least the functional requirements of the final product are fully specified by the legacy system (the new system is expected to be functionally equivalent to the old one [Sneed, Wolf, and Heilmann, 2010, p. 25]), the very nature of legacy software systems makes it hard to foresee all process and toolchain requirements in advance. Legacy systems are huge, complex, and undocumented. To gain insight into these old systems, they must initially be analyzed – a software evolution activity which itself requires tool support.

Adapting project toolchains to newly found insights in an iterative manner is therefore a sensible approach, provided that the toolchain can sustain repeated changes without slowing down the project. Sim [2000] confirms the need for this kind of agility, underlining the importance of experimentation. Grieger and Fazal-Baqaie [2015] also emphasize flexibility of software transformation processes as a significant factor.

Thus, the notions of *agility* and *flexibility* are intertwined. In this thesis, the former is used in the context of software (evolution) projects and their processes, in the meaning established by the Agile Manifesto [Beck et al., 2001]. The latter refers to the effort required to adapt such projects' toolchains, to account for changing requirements as the project progresses. This corresponds to an understanding of flexibility as used by Eden and Mens [2006], who provide *evolution cost metrics* to quantify the degree of flexibility. The authors further highlight that flexibility is always measured with respect

to a concrete type of change. This may be used to distinguish the concept from more general terms, like *maintainability*, *extendability* [ISO/IEC/IEEE 24765, 2017], *changeability* [ISO/IEC 9126-1, 2001], and *modifiability* [ISO/IEC 25010, 2011][1].

### 1.2.2 Increasing Reusability

The lack of tool interoperability manifests as an inability to reuse both the glue code that ties the individual tools together into a toolchain, as well as the tools themselves. From literature, it seems there might be two different persuasions:

Mens et al. [2005] seem to suggest developing tools for a single platform to achieve interoperability, which could be interpreted as the kind of "closed systems" Müller et al. [2000] are warning against. Their example is *Moose*, a platform that arguably requires tools to be specifically built for it, as it restricts them to work in close confinements.Tools build for such platforms are inherently interoperable, but only within these confines. Existing tools are effectively excluded, i.e. reusing and integrating tools existing outside of such an ecosystem would be tedious and expensive.

A light-weight interoperability approach would be able to better incorporate existing tools. Müller et al. [2000] suggest the Unix philosophy as an analogy, and Jin and Cordy [2005b] suggest service-oriented concepts as a means of abstraction.

### 1.2.3 Increasing Productivity

Large software evolution projects are often chosen as an alternative to complete redevelopments, because of reduced risk and cost. Borchers [1996] points out the importance of automation for cost-efficiency. Creating project-tailored integrated tool support takes a lot of effort. Once established, the actual modernization process is mostly automated. Reducing the toolchain-building effort is therefore an opportunity to considerably reduce effort and costs overall.

In contrast to this industry perspective, Ghezzi and Gall [2013] highlight an aspect relevant to academic research: the effort required to rebuild toolchains representing experiment setups is often prohibitively high. Consequently, experiments of one research group are rarely repeated by other scientists for confirmation, an important and common practice in empirical research in other domains.

Both the flexibility of toolchains, as well as the reusability of its individual parts and the code integrating them, contribute to increasing productivity by reducing the effort required to build and adapt the overall tool support. Flexible tooling allows projects to become more agile – to react to new insights and requirements quickly, and allow for more experimentation and exploration of the solution space. Reusability eliminates repetitive work, leading to productivity gains growing over time. Such an increase in

[1]The notion of flexibility established by these standards differs to varying degrees from the one described here. The definition of flexibility by ISO/IEC 25010 [2011], for example, refers to user experience.

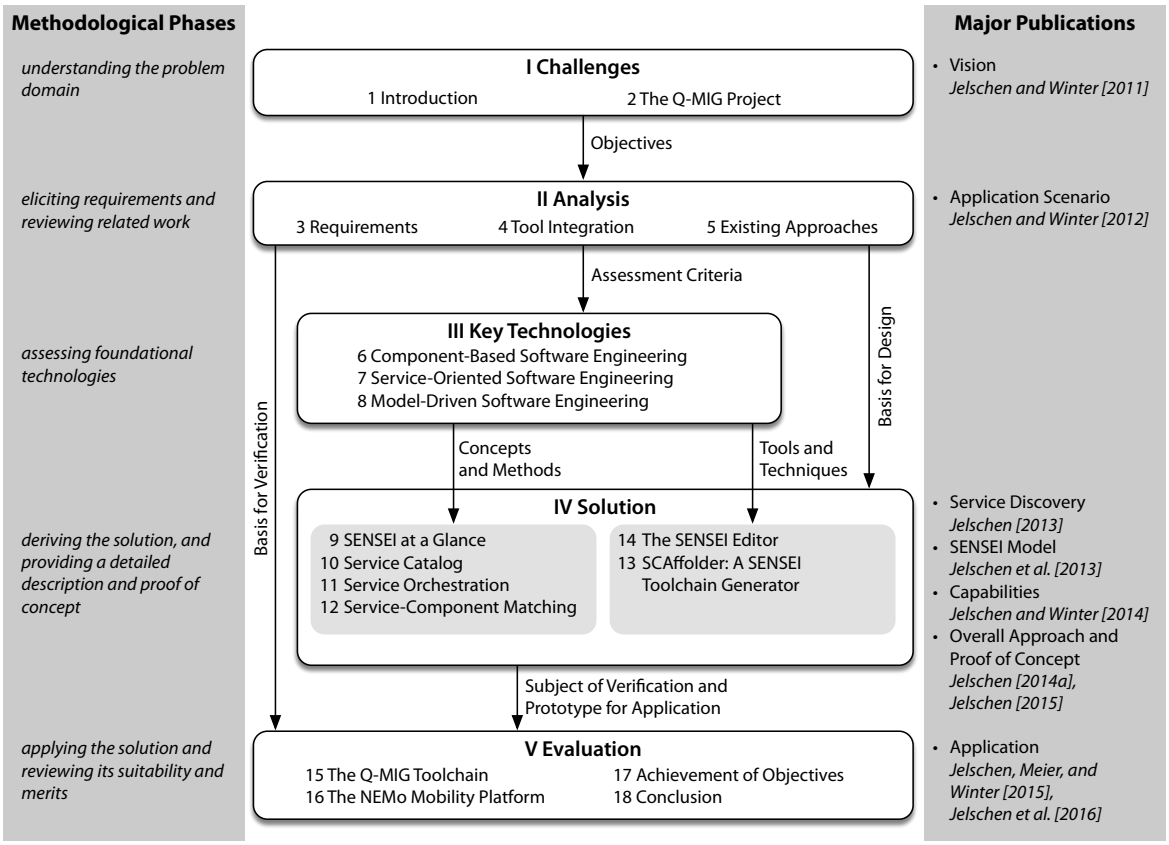



Figure 1.1: The structure and outline of this thesis, with associated methodological phases, and the most relevant scientific publications.

productivity makes software evolution projects more economically viable; in particular, it may enable carrying out projects that would be infeasible without the productivity boost gained through integrated, flexible, and reusable tool support.

## 1.3 Thesis Outline

The thesis consists of five parts: I Challenges, II Analysis, III Key Technologies, IV Solution, and V Evaluation. The overall structure is depicted in Figure 1.1, where the parts are represented by boxes, containing the associated chapters. The arrows indicate dependencies: **Part I** provides an overview of the problem domain and identifies the core *objectives* of this thesis (Chapter 1), and presents the software evolution project *Q-MIG* (Chapter 2), used throughout as a motivating example.

Using this problem and domain understanding, the objectives are further refined in **Part II**, by deriving requirements for creating a toolchain-building support framework from the general activities that make up the process of toolchain-building (Chapter 3), and by analyzing the field of tool integration to clearly place this work within it (Chapter 4). Both chapters provide reference frames for assessing and demarcating related work and existing approaches with respect to these requirements (Chapter 5).

**Part III** provides extensive overviews of fundamentals and state of the art in component-based (Chapter 6), service-oriented (Chapter 7), and model-driven software engineering (Chapter 8). The principles, techniques, and tools of these software engineering paradigms are evaluated for their suitability towards satisfying the objectives defined, and to form the foundations of the pursued solution approach, using the requirements elicited in Part I as *assessment criteria*.

With the overall objectives and requirements described in Part I and Part II as *basis for design*, the solution, SENSEI, is derived and described in **Part IV**. SENSEI is based on *concepts and methods* identified from reviewing key technologies (Part III), and establishes clear structures for tool development and integration, based on a metamodel, methods and principles to fill and work within these structures, and specifies means and techniques to automate large parts of the toolchain-building process. An overview of SENSEI is given in Chapter 9, with Chapter 10, Chapter 11, and Chapter 12 providing detailed accounts on particular parts of the overall framework. SENSEI depends on tooling to support the toolchain-building process, namely a toolchain generator, and editors to create and modify the models used within the approach. As proof of concept, prototypes have been implemented, using *tools and techniques* identified in Part III: the SENSEI editor (Chapter 13) and the toolchain generator SCAffolder (Chapter 14).

The solution is evaluated in **Part V**, to provide proof of its general, practical feasibility, and corroborate its utility with respect to the objectives. Part II provides the *basis for verification*, and Part IV provides both the *subject of verification and prototype for application*: the SENSEI approach is practically applied to concrete toolchain-building problems using SCAffolder and the SENSEI editor. Using SENSEI, toolchains have been built for the software migration and quality project *Q-MIG* (Chapter 15). Furthermore, SENSEI has also been used outside of the domain of software evolution toolchain-building, to model business processes, and subsequently derive and integrate the corresponding software support for the *NEMo* mobility platform (Chapter 16). Chapter 17 revisits the objectives of the thesis and examines their achievement. The overall results of this thesis, its scientific contributions and expected benefits are summarized in Chapter 18.

The parts of the thesis correspond to research phases as indicated in Figure 1.1 on the left-hand side. The most relevant publications are indicated on the right-hand side, in relation to the topic and stage of the thesis being covered.

CHAPTER 2

# The Q-MIG Project

The origin of this thesis are the challenges posed by having to create integrated toolchains to automate reengineering and migration processes of software evolution projects. Chapter 1 gave a brief glimpse of the field of software evolution, and provided evidence that the issue of toolchain-building largely remains an open one. In this chapter, a concrete software evolution project is presented as a motivating example to further substantiate these claims, and to provide a tangible impression of the general conditions and practical constraints characterizing such projects. In particular, the chapter introduces the processes that needed to be automated with appropriate tool-support.

First, Section 2.1 gives a high-level overview of the project. Next, a small, simple subprocess and its associated tooling is described in Section 2.2, which will be used throughout the thesis as a running example. Then, the challenges faced during the project's run are described in Section 2.3, and conclusions are drawn in Section 2.4, providing further evidence for the need for a toolchain-building support framework, and support for the objectives of this thesis.

## 2.1 Overview

Q-MIG was a BMWi-funded[1], joint research and development project of the Software Engineering Group of the Carl von Ossietzky University, and industry partner *pro et con* GmbH. It ran from January 1, 2014 until March 31, 2015. The two main objectives were to 1. extend the existing COBOL-to-Java migration tools of *pro et con* into a generic, customizable migration toolchain, and 2. complement the migration toolchain with a

[1]The project Q-MIG was funded by the Central Innovation Program SME (Zentrales Innovationsprogramm Mittelstand – ZIM) of the Federal Ministry of Economics and Technology (Bundesministeriums für Wirtschaft und Technologie - BMWi), Funding code: KF3182501KM3.

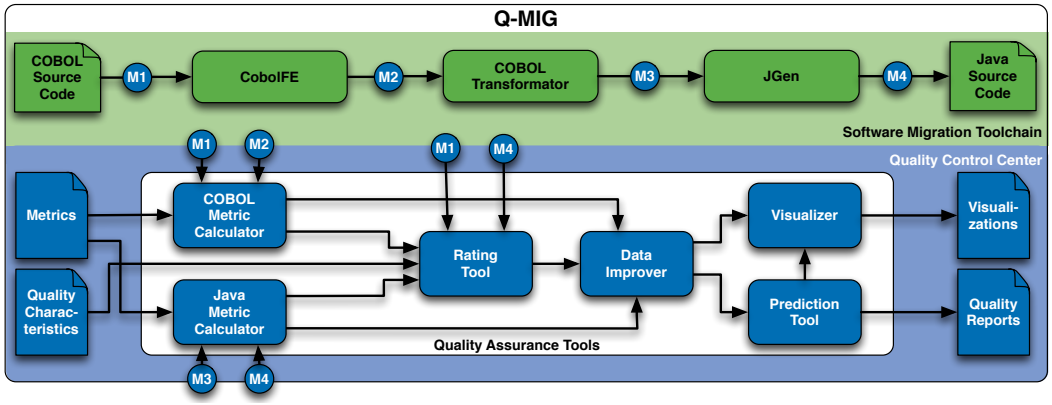


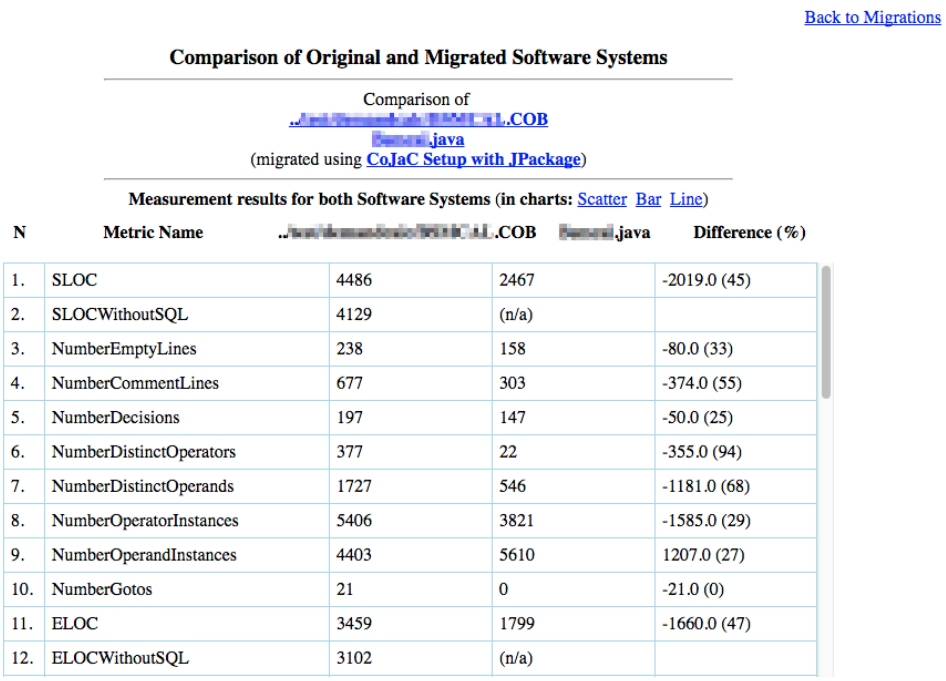

Figure 2.1: Q-MIG migration toolchain and quality control center [Meier et al., 2015].

quality control center to measure, compare, and predict the inner quality of software systems before and after migration, allowing the toolchain to be tailor-fitted towards project-specific quality goals.

While the first objective was worked on by *pro et con*, the second objective was mainly realized by researchers of the Software Engineering Group of Carl von Ossietzky University. Figure 2.1, taken from Meier et al. [2015], depicts the tools of the migration toolchain (upper part) and the quality control center (lower part), integrated at monitoring points *M1* through *M4*. Some intermediate steps, sub-activities, and tools have been omitted for simplicity, so the general setup can be conveyed clearly. A drill-down into a small part of it, considering the parsers and metric calculators more closely, is provided in Section 2.2, to serve as simple, illustrative example throughout the thesis. Chapter 15 will further describe the toolchain.

The software migration toolchain shows *COBOL source code* on the left, being processed first by a parser, the *CobolFE* (COBOL Frontend). The resulting COBOL abstract syntax tree (AST) is transformed into a corresponding Java AST by the *COBOL Transformator*. *JGen* produces compilable Java source code from this. The intermediate results between those steps can be accessed for quality anaylsis via *monitoring points*.

The quality control center calculates values for various software *metrics* on both the original COBOL (*COBOL Metric Calculator*) and the resulting Java systems (*Java Metric Calculator*). Inner *quality characteristics* like modularity, reusability, or changeability (in accordance with ISO/IEC 25010 [2011]) are used to quantify high-level observable quality properties, as opposed to measurable metrics. The characteristics are used to represent the “real” quality as assessed by human experts. By way of correlation, they provide an interpretation for metrics, which in turn allows to infer characteristics from metrics.



Back to Migrations

**Comparison of Original and Migrated Software Systems**

Comparison of
..L.COB
.java
(migrated using CoJaC Setup with JPackage)

**Measurement results for both Software Systems (in charts: Scatter Bar Line)**

| N | Metric Name | ..L.COB | .java | Difference (%) |
|---|---|---|---|---|
| 1. | SLOC | 4486 | 2467 | -2019.0 (45) |
| 2. | SLOCWithoutSQL | 4129 | (n/a) | |
| 3. | NumberEmptyLines | 238 | 158 | -80.0 (33) |
| 4. | NumberCommentLines | 677 | 303 | -374.0 (55) |
| 5. | NumberDecisions | 197 | 147 | -50.0 (25) |
| 6. | NumberDistinctOperators | 377 | 22 | -355.0 (94) |
| 7. | NumberDistinctOperands | 1727 | 546 | -1181.0 (68) |
| 8. | NumberOperatorInstances | 5406 | 3821 | -1585.0 (29) |
| 9. | NumberOperandInstances | 4403 | 5610 | 1207.0 (27) |
| 10. | NumberGotos | 21 | 0 | -21.0 (0) |
| 11. | ELOC | 3459 | 1799 | -1660.0 (47) |
| 12. | ELOCWithoutSQL | 3102 | (n/a) | |

Figure 2.2: A page from an HTML report generated with the Q-MIG toolchain. Some file names have been pixelated for confidentiality reasons.

The *Rating Tool* supports software migration experts in recording quality estimates for Java and COBOL systems. Experts can browse all relevant entities and levels (e.g. for packages, files, classes, methods, etc. in Java) and enter their ratings for each considered quality characteristics.

Metrics calculated for both COBOL and Java systems, as well as expert quality estimates, are passed into the *Data Improver* for preprocessing. The consolidated data is used by the *Prediction Tool* to either "train" prediction techniques (linear regression, artificial neural networks), to infer quality characteristics from metrics, or to project the quality of a prospective Java system if it were migrated from a given COBOL system using a particular migration setup. The *Visualizer* component provides the ability to create diverse graphs and diagrams, as well as HTML-based quality reports from measured, expert-estimated, or predicted metric and quality characteristics data for COBOL and Java systems, to aid comparison and further analysis. Figure 2.2 depicts a page from such a HTML report as an example, showing several metric values for both an original COBOL file and a corresponding Java file, along with the absolute and relative difference.

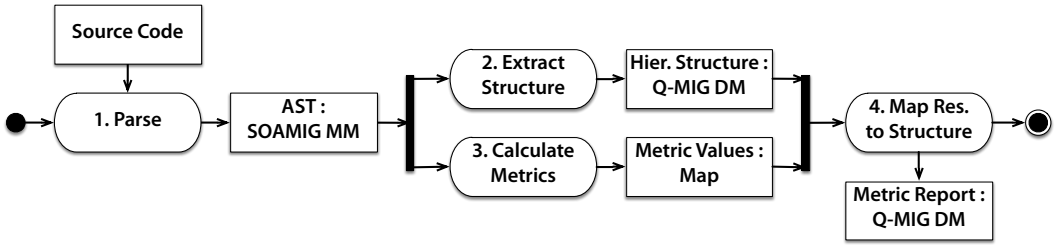



Figure 2.3: Base metric calculation in Q-MIG.

## 2.2 Example: Base Metric Calculation

This section provides a simple example of a tool-supported software evolution process. It is just a small subprocess of the overall, quality-driven migration process and toolchain developed within the Q-MIG, and will be used throughout the remainder of this thesis as a running example.

The base metric calculation process is the basis for subsequent quality analysis steps that make up the quality control center of the Q-MIG project (Chapter 2). Since the overall toolchain of the project is quite extensive, only this small sub-toolchain is presented here for illustrative purposes.

While COBOL metric values were calculated with existing *pro et con* tools, much of the Java metric calculation infrastructure was implemented from scratch for Q-MIG. Base metric calculation for both COBOL and Java takes a software system and a list of metrics, and calculates their respective values for all applicable source elements. As depicted in Figure 2.3, the process can be broken down into four basic steps:

1. **Parse**. The source code of the input system is parsed into an abstract syntax tree representation, which is more adequate for the expression and calculation of most metrics[2].
2. **Extract Structure**. The basic, structural elements are extracted from the AST for a simple, hierarchical representation. Java systems are decomposed into *packages*, *files*, *classes*, and *methods*, COBOL systems into *files*, *devisions*, and *sections*.
3. **Calculate Metrics**. All metrics in the provided list are iterated, evaluated, and the resulting values are stored with a reference to the source element the refer to.
4. **Map Results to Structure**. The metric values produced in Step 3. are merged with the hierarchical system representation of Step 2. to derive the final result, a report of base metric values for the whole system, hierarchically decomposed into its subsystem structure.

[2] *Lines of code* would be the prime counter-example; in the Java Metric Calculator, the original files are therefore passed through to also be able to evaluate metrics directly on the source code.

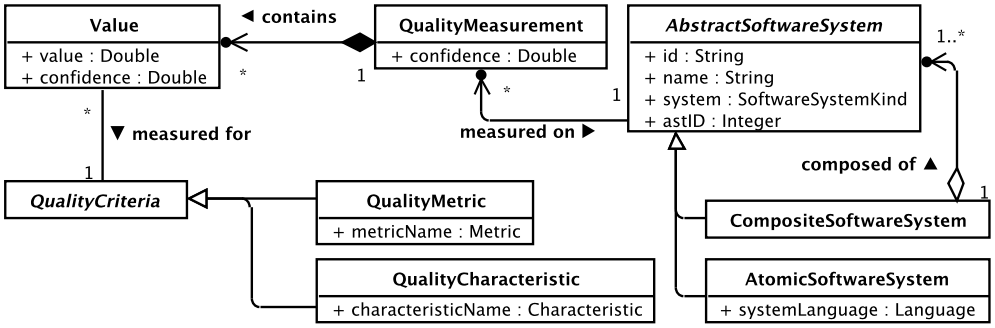



Figure 2.4: Simplified excerpt of the Q-MIG data model.

The parsers for both COBOL and Java were provided by *pro et con*. Their output conforms to COBOL and Java metamodels which have been defined for the previous project, *SOAMIG* [Fuhr et al., 2012]. The ASTs are represented in an XML-based file format also developed for SOAMIG based on the metamodels.

In Q-MIG it was necessary to define another model to represent quality measurements for storage and exchange. Due to legal restrictions, it had to be ensured that the industrial-scale software systems used for evaluation in Q-MIG would not leave the premises of project partner *pro et con*, i.e. neither source code nor abstract syntax trees were transferred to the University of Oldenburg. Instead, the whole base metric calculation was performed at *pro et con*'s site, who then only transferred the resulting metric reports conforming to the Q-MIG data model, and expressed in its corresponding XML exchange file format [Jelschen, 2014b]. A simplified excerpt of the Q-MIG data (meta-) model is depicted in Figure 2.4. Out of a total of 23 classes, 8 are shown, which form the basis for expressing quality measurements as a collection of values measured for quality criteria (such as metrics), and to describe the basic hierarchical decomposition of software systems.

For analysis, full knowledge of the source code was not necessary, but the basic structure *was* still needed to make sense of the fine-grained metric values. Therefore, this structure is extracted in Step 2. This also only leaves arbitrary, numeric identifiers in place of the original names, so that no conclusions about the nature of the original software systems is possible. As shown in Figure 2.3, Steps 2. and 3. are independent of each other and can therefore be executed concurrently. The implementation does not exploit this fact, though.

In Step 3., the actual metric values are derived. The COBOL metric calculator was developed and used exclusively by *pro et con*, and its inner workings were not

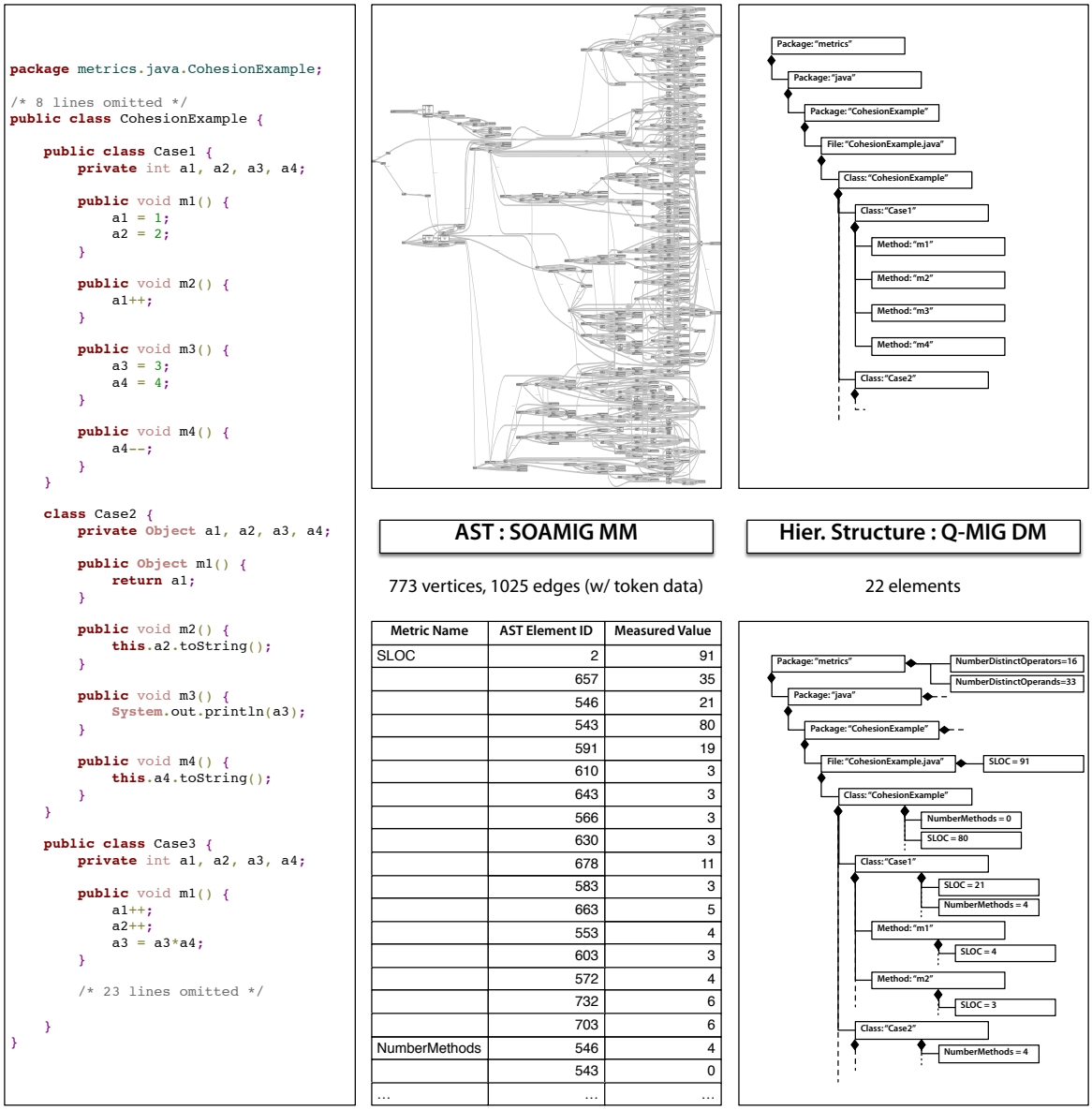



Figure 2.5: Base metric calculation example artifacts.

disclosed. In contrast, the Java metric calculator was developed from scratch by University of Oldenburg's software engineering group, internally using a TGraph-based representation [Ebert, Riediger, and Winter, 2008] of the AST. Metrics are expressed as GReQL queries (*Graph Repository Query Language*). Each metric is coded for a particular level in the system hierarchy, e.g. package, class, or method. Therefore, the input list of metrics also has to be specific about this, and list a metric together with, and once for each granularity level it should be evaluated over. In the actual implementation though, the metric calculator has been hard-wired to always loop through *all* defined metrics. The result of a single metric evaluation is a map of code element references to corresponding metric values.

In the final step, the metric values are written into the hierarchical system representation. The result is an instance of the Q-MIG datamodel, which is represented internally as another TGraph, and is written out as XML, conforming to the corresponding exchange file format.

Figure 2.5 shows examples of the artifacts that are produced during a run of the base metric calculation toolchain. The system being measured consists only of a very small Java source code file called *CohesionExample*, an example taken from the Q-MIG test suite, originally intended to test the cohesion metrics of the tools. The parser produces an abstract syntax graph (an abstract syntax tree with additional crosslinks). Full token information from lexical analysis is preserved in this structure, required for certain metrics. This makes it exceptionally large, which is also why it is only depicted as "10,000 foot" overview – even zooming in would yield little insight, as vertices carrying fine-grained token information are cluttering the view towards more interesting elements and relationships. These central, structural elements are extracted in the next step, depicted as tree structure.The actual metric calculation also uses the AST as input and produces a table mapping metric values to the AST elements they were measured on by referring to their IDs. Finally, the latter two results are merged into a single result by attaching the metric values to the corresponding elements in the hierarchical system representation, yielding the metric report for the CohesionExample system.

## 2.3 Challenges

As with every project, the overall requirements evolved as it progressed, e.g. due to new insights and added practical constraints. To cope with this, the software industry in general has moved away from rigid, waterfall-like development models, towards more agile methods, which was also practiced in Q-MIG. However, the efforts towards building the quality control center toolchain proved to be a major bottleneck, considerably impeding overall flexibility and constricting the available options for action. The following sections list examples of concrete challenges encountered during the project.

### 2.3.1 Integrating Existing Tools

The most overt challenge was the *selection and integration of existing tools*. Even though many tools for software metric calculation and analysis were readily available, it was decided to recreate most of this functionality from scratch. One aspect was that using existing tools would somewhat limit the control over the precise functionality, e.g. the kind and extent of the available metrics, and the exact calculation rules used for them. This was traded off against the development effort of creating custom tools. However, the decisive reason to develop custom tools for most of the functionality was tool interoperability, and the required integration effort.

The metric tools considered for use were found to be self-contained tool suites, meant to be used on their own. Stovepipe systems like this are hard to integrate [Boehm, 2006]. The only case in which an existing tool was chosen was to calculate code clones. Creating a custom code clone detector was considered too expensive, so existing tools were researched and evaluated under Q-MIG-specific criteria, a task of non-negligible overhead in itself. The results were compiled in an internal report [Pandey, 2014]. After an initial, broad overview, most tools were excluded quickly, mostly due to a general lack of interoperability, or technical implementation choices too different from Q-MIG's technical requirements that would have incurred major integration hurdles. Juergens, Deißenböck, and Hummel [2009] come to a similar conclusion regarding code clone detection tools presented in scientific publications, reporting that the tools themselves are either closed source, inextensible, or not publicly available, at all.

Therefore, only three tools were considered more closely, ConQAT [Juergens, Deißenböck, and Hummel, 2009], CCFinderX [Kamiya, Kusumoto, and Inoue, 2002], and DuDe [Wettel and Marinescu, 2005]. The latter was selected, almost exclusively on interoperability grounds. That said, DuDe has not been designed for interoperability, offering a self-contained solution with a graphical user interface, but no programming interface. However, it is Java-based, which allowed for a rather immediate, programmatic integration with the rest of the toolchain. This still required to review and understand the tool's implementation internals, and use them to create an adapter to the rest of the Q-MIG toolchain, a use case the tool clearly was not build for.

### 2.3.2 Reusing Custom Tools

The Q-MIG toolchain was created incrementally as a result of an iterative process, a sound practice, particularly considering the innovative nature of the project. An early increment of the Java metric calculator only supported calculating metric values for the analyzed system as a whole. With later increments, granularity levels were introduced, allowing the calculation of metric values for individual parts of a system, such as methods, classes, and packages.

The addition of these features required quite heavy refactoring, possibly due to insufficient up-front architectural design to anticipate these features, an issue often en-

countered within the scope of agile development processes [Knodel and Naab, 2014]. More significantly, these features were integrated quite tightly within the metric calculator, which made sense for the project's use cases: they always required a complete analysis of a given software system, with metric values calculated for all sub-artifacts on all granularity levels.

Only later it was realized (when recreating the toolchain using the approach developed in this thesis, see Chapter 15) that these design decisions considerably reduced the metric calculator's generic reuse value. While not really a challenge during Q-MIG, potential future projects requiring metric calculation would have to heavily adapt the tool, or it may not even be effectively reusable, at all.

### 2.3.3 Supporting a Distributed Process

Q-MIG also required the integration with data from the existing migration toolchain of the project's industry partner *pro et con*. Due to non-disclosure agreements, it turned out that *pro et con* would not be able to provide any real-world COBOL or migrated Java programs to evaluate metrics on. Therefore, parts of the toolchain had to be decoupled and relocated to the premises of *pro et con*, and additional interfaces had to be introduced to exchange measurement data transparently between the project partners.

Non-disclosure concerns are common in reengineering and migration projects, as the corporations owning the legacy systems being modernized view them as important assets facilitating their competitiveness. Projects including external partners, e.g. consultancies and outsourcing service providers, will therefore have to take organizational, process, and technical measures to ensure data security, or perform certain tasks only on the premises of the company owning the legacy systems under modernization.

In Q-MIG, fully integrating and deploying the toolchain to a distributed environment was infeasible within the frame of the project, both from an organizational (travel expenses, communication overhead) and a technical perspective (additional effort to set up a distributed computing environment). Invocation of remote toolchain parts was therefore not automated, and always required human intervention and coordination with the project partner.

This turned out to be a major impediment, since the kind and extent of the data exchanged between Q-MIG's partners evolved with the gaining of new insights as the project advanced, but the lack of automation did not allow for fast turnarounds, and the physical distance limited the ability to experiment, and debug the evolving toolchain.

### 2.3.4 Supporting Domain Experts

The central objective of software evolution projects is not tool integration – it is evolving (reengineering, migrating, modernizing) software systems. In Q-MIG, the main research questions revolved around comparative software quality analysis. Therefore, the team included data scientists, who were less experienced as programmers.

Designing data analysis experiments includes setting up the tool support, however. Lacking interoperable tools or an integration framework, this is a task that requires programming expertise. To integrate third-party tools, familiarity with all the technologies underlying each and every tool is required, or has to be acquired. For custom tool implementation, solid software development skills are even more essential, while also requiring expertise regarding the software evolution technique to be realized.

In Q-MIG, it proved hard to break work packages down into tasks appropriate to the skill set of the different team members. This resulted in team members waiting on each other, and in having to compromise on the tools' and toolchain's inner quality; both lead to impeded experimentation, and generally lowered project agility.

## 2.4 Conclusion

Q-MIG concluded as a success, in the end, although the number and extent of performed quality measurements and experiments had to be lowered to account for the overhead incurred by the described tool development and integration issues.

Within Q-MIG, the developed toolchain was used to calculate metrics, and predict quality characteristics of prospective migration results, but it was also a major outcome in its own right. In most software evolution projects, the toolchain is a means to an end, subordinate to the core objectives of the project, e.g. improving inner quality, or migrating it to a new technological ecosystem. Domain experts may perceive the toolchain merely as disposable, one-off software, and treat its development with less care because of this, potentially aggravating later issues that require changing it.

From the project experience, the following list sums up the key *lessons learned* with respect to software evolution toolchain-building, as well as derived *target criteria*, which will be revisited in Chapter 3:

- Existing software evolution tools lack interoperability. Furthermore, it requires effort to find and evaluate tools to be potentially used in a toolchain.

  **Target Criteria**: domain experts need to get an *overview of available tools* appropriate to their goals more quickly. It should also be possible to *choose tools based on functional requirements*, not technical interoperability concerns.

- Building reusable software is hard [Schmidt, 1999]. Building custom tools during software evolution projects is a means to an end, making the tools themselves merely a by-product. Within the scope of individual projects, there is usually no clear incentive to spend extra effort to make the tools reusable – although this may pay off even within the scope of the same project, if requirements change and the toolchain needs to be adapted.

  **Target Criteria**: there should be *light-weight support mechanisms* that guide design towards *interoperability* and *reusability*. Tool development should ideally be cleanly separable from toolchain definition and integration.

- Automation is key. Gaps in the toolchain requiring manual steps to be performed will slow down a project considerably, and reduce the overall ability to react to changes. If a single project team member has to manually copy some files over, this will incur a delay of a couple of minutes; if the manual tasks are more involved, this might become a few hours. However, if multiple persons of different teams, located at different offices (potentially far apart) have to coordinate and wait on each other, the projects may come to a halt for days. A distributed project requires a distributed, fully integrated toolchain.

  **Target Criteria**: there needs to be a support framework to base the integration of individual tools into toolchains on. Such a framework should provide an abstraction layer from concrete deployment schemes and interoperability issues, automating the integration process as much as possible

- The lack of interoperable tools and toolchain-building support demands that domain experts are also experts in software development and integration.

  **Target Criteria**: a process and structure is needed that properly separates the design of software evolution processes from both tool development and toolchain integration, so domain experts can define processes in terms of necessary steps and their arrangement, independent of concrete tools, implementation technologies, data formats, and general interoperability.

In their sum, these findings already point toward three major software engineering paradigms that will be reviewed for their shared objectives with this thesis: component-based software engineering (Chapter 6) is motivated mainly be *reusability* and *composability* of software building blocks, service-oriented software engineering (Chapter 7) is domain-centric and focuses on interoperability and flexibility, and model-driven software engineering (Chapter 8) offers the means to (automatically) *map* non-technical problem *specifications* expressed in the language of the domain to appropriate technical *implementations*, without entangling the two.

Besides the importance of automation through fully integrated toolchain support, another essential insight from Q-MIG is that software evolution projects require experimentation, and are subject to *change*, therefore requiring to sustain their agility. This is something software projects of any variety – not just software evolution projects – have to deal with all the time, and usually struggle to do so. In fact, the need to change is ironically what made legacy systems legacy, and gave rise to software evolution projects, in the first place. The key takeaway is that software evolution projects have to be able to *evolve* themselves – and when they do, their tool support has to follow suit.

# PART II

# Analysis

With fundamental objectives defined in Chapter 1, and the Q-MIG project introduced in Chapter 2 as motivating example providing concrete evidence for the need for toolchain integration in practice, the following chapters aim to further refine the objectives, clearly place this thesis within the overall field of tool integration, and analyze and evaluate existing work and related approaches for comparison and delimitation. This will yield a refined, clear-cut view of what SENSEI intends and does not intend to achieve, which will guide the exploration of suitable software engineering paradigms in Part III, and the design of the solution, presented in Part IV.

First, Chapter 3 refines the already established objectives by breaking down and analyzing the toolchain-building process, and deriving requirements from the individual steps. Then, Chapter 4 reviews the field of tool integration, its history, terminology, and its different aspects and branches. Both chapters provide reference frames for further analysis: The former mainly to assess the suitability of both existing approaches as well as foundational technologies for SENSEI, with respect to the objectives of this thesis. The latter is used to refine and particularize the scope of SENSEI, and delineate the approach from related work.

Therefore, both of these reference frames form a basis for the comprehensive review of existing tool integration approaches and other related work in Chapter 5. The requirements are used for evaluation, leading to an overview and comparison grid. Reviewing other approaches using a common terminology and classifying them within the tool integration space helps identifying differences in terms of focus, scope, and application areas. This way, the chapter further stresses the need for a toolchain-building support framework, and highlights SENSEI's unique features and novelty.

CHAPTER 3

# Requirements

SENSEI is intended to support the building of toolchains for automating processes, mainly motivated by the need to do so in many software evolution projects. Chapter 4 will review the field of tool integration to clarify what kinds of toolchains there are, and which of those this thesis addresses. Before that, this chapter will take a purely problem-driven approach to determine what is required to support the creation of integrated toolchains. There are different ways to go about this:

1. Study literature for challenges in software evolution toolchain-building, for example (industrial) case study reports.
2. Perform case studies to discover challenges of the toolchain-building process.
3. Analyze the toolchain-building process itself, by breaking it down into activities, and identify opportunities for automation, or support of manual tasks.

From *literature*, general challenges have been identified. They motivated the development of SENSEI and have already been presented in Chapter 1. They are at the basis of this thesis and its objectives. The requirements presented in this chapter are more specific and more readily mechanizable, but they all serve to meet the main objectives.

Several *case studies* of different size and scope have been performed, as well. The first one was based on a hypothetical project to migrate COBOL systems to Java. A toolchain prototype was built manually and described by Jelschen et al. [2012]. The conceived software evolution scenario was based on first ideas on researching the dynamics of the inner quality of software undergoing migration. These ideas evolved into the Q-MIG project, with the goal of "building a quality-driven, generic toolchain for software migration" [Q-MIG, 2015], with the ability to measure, compare, and predict the quality of software systems being – or planned to be – migrated from COBOL to Java. The Q-MIG project was used to apply and evaluate the SENSEI approach (Chapter 15). Another evaluation was performed outside of software evolution (Chapter 16),

which shows that, while this thesis is rooted in that field, the particular kind of integration problem that is being studied here is not confined to software evolution, and so neither is the derived solution, SENSEI.

Another set of studies have been performed by students under the author's supervision. These studies were each targeting a particular sub-field of software evolution: *software measurement*, *architecture reconstruction*, and *impact analysis*. The goal of the studies was to break these software evolution activities down into logical, reusable steps, and to analyze for concepts required to model and compose them as *services*. These studies have informed the design of the SENSEI meta-model (Chapter 9), and led to a publication [Jelschen et al., 2013]. They have provided both conceptual and technical insights of how to model and implement toolchains, which have informed design decisions in the creation of SENSEI. The studies will be referred to in Chapter 9.

They also guided the break-down of the general *toolchain-building process*. Its analysis for requirements is the central focus of this chapter. First, Section 3.1 describes the steps of the toolchain-building process. Section 3.2 considers the steps of the toolchain-building process one by one, and derives requirements for a support framework. A summary of all requirements is provided in Section 3.3, with an overview in Table 3.1.

These requirements will provide the frame of reference for selecting appropriate techniques to utilize for building SENSEI, and for evaluating existing approaches in Chapter 5. They will again be referred to as guiding principles in designing SENSEI (Chapter 9), and in implementing a reference prototype (Chapter 14).

## 3.1 The Toolchain-Building Process

Toolchains are built to automatically perform one or more *activities*, which are composed of interdependent *tasks*: if there are individual tools able to automate the tasks, but none that can perform the entire activity, the tools need to be integrated into a toolchain. Conceptually, the process of creating toolchains can be broken down into a series of steps, depicted in Figure 3.1. On the highest level, three phases are identified: *goal determination*, *specification*, and *implementation*. Goal determination is about defining the activities to be automated in terms of their purposes and intended outcomes. This is a prerequisite to the actual design and realization of the toolchain, and thus viewed the "zeroth" step of the process, not to be considered here any further.

Toolchain *specification* is about breaking down activities to be automated into its logical steps (*task identification*), and connecting them in such a way that they jointly achieve the overall goal (*task coordination*). Toolchain *implementation* is about mapping the conceptual process resulting from the previous phase onto concrete tools, and wiring them up in the required manner. This involves finding appropriate tools for each task (*task instantiation*), and then actually *integrating* them: tools must be enabled to be invoked by each other or by a central process manager, so their interfaces must be connected to (*adapter creation*), and data must be brought into the expected formats

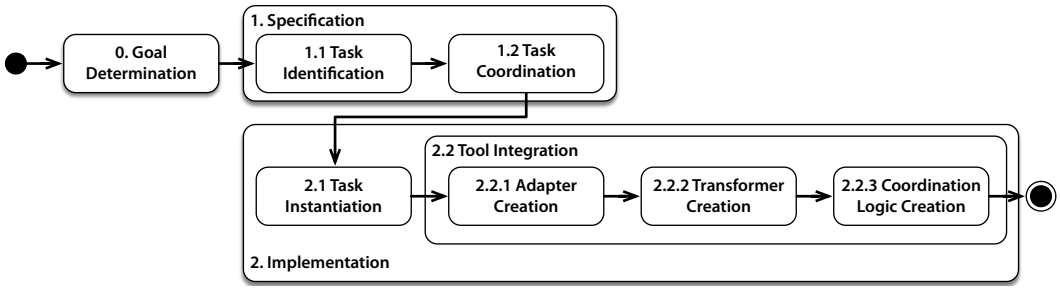



Figure 3.1: Activities of the toolchain-building process.

(*transformer creation*), before the specified process can be manifested in code (*coordination logic creation*).

This process is to be understood as an idealized view that, in practice, will probably rarely be followed to the letter. However, the individual steps of the process are all necessary for creating toolchains, and have to be performed in some way, shape, or form, which can be more implicit, and intermix and entangle some steps. The purpose of the process is mainly to identify the individual pieces of work for a clear picture of what is involved, so that it can guide the elicitation of requirements in Section 3.2.

Breaking down the toolchain creation process into its constituent parts also allows to assign responsibility for individual steps to different roles. At least two distinct roles can be readily identified:

**Domain experts** know and understand the problem that needs to be addressed with a toolchain. They do not necessarily have the technical skills required to build tools or integrate them into toolchains.

**Tool developers** know the technical characteristics of the tools they have created. They possess software development skills, but do not necessarily have experience with the technologies used in tools they did not develop themselves, or with methods and techniques required for tool integration.

A third role, responsible for taking the tools made by tool developers and combining them into an integrated toolchain that conforms to the specification provided by domain experts, could be introduced: **integration experts**. However, this role's tasks are precisely those that this thesis seeks to automate, as manual integration would work against its objectives, particularly the need for *flexibility*.

The toolchain-building process reveals that only domain experts have the knowledge to perform the specification steps, while implementing a conforming toolchain may well be outside their area of expertise. Tool developers have the required, technical skills, but are usually not involved in the process, unless tools get specifically created for a particular project. Put differently, domain experts know *what* is to be done (spec-

ification in terms of tasks and activities), whereas tool developers know *how* to realize it technically (implementation in terms of tools and toolchains).

This chapter intentionally refrains from using the more specific terminology which will be introduced later when presenting SENSEI (Chapter 9), to avoid confusing the problem and solution spaces. That being said, in SENSEI, *tasks* roughly correspond to *services*, and *tools* get encapsulated by *components*.

### 3.1.1 Toolchain Specification

Step 1.1 identifies the necessary tasks on the way towards the defined goal. For example, creating side-by-side comparison charts for quality metrics of COBOL and migrated Java systems, metrics have to be calculated on COBOL (first task) and Java systems (second task), and the result has to be rendered into the desired quality reports (third task). Conceptually, this step is kept separate from deciding which tools to use, or how to implement the tasks (Step 2.1), even if in practice those steps might get mingled. This step is also not (mainly) concerned with sub-tasks necessary solely for technical reasons, e.g. data format transformations. However, the borderline between functional and technical tasks can be a bit fuzzy. For example, to calculate metrics, the source code will probably have to be parsed first, a very common and natural reverse engineering task. As a possible criterion for exclusion, parsing is never an *end in itself*, but rather a *means to an end*. Still, if the technical task is sufficiently complex, it makes sense to explicitly identify it, so that a dedicated tool can be chosen or created in Step 2.1, as opposed to handling the data transformation solely as pre- or post-processing as part of Step 2.2.2.

Step 1.2 concludes the specification of the toolchain, by describing the intended data and control flow. Data flow specifies which tasks' outputs should be used as input for which other tasks. Defining data flow might be sufficient for simple toolchains, where the data flow completely determines control flow as well, i.e. where tasks are performed as soon as all input data becomes available. If a process requires the toolchain to perform certain tasks conditionally or repeatedly, explicit control flow has to be specified, accordingly. A simple, graphical depiction of the intended data flow, only, is shown in Figure 2.1 (p. 12). Notice, though, that in that figure, the boxes represent tools rather than tasks, indicated by the way they are labeled.

### 3.1.2 Toolchain Implementation

Once the specification is complete, the toolchain can be realized technically, to automate the software evolution project's processes. Again, note that this is a purely conceptual model of how the tool support for a process is built. In reality, although all of the steps mentioned here probably have to occur in some form, they may not be separated as cleanly as presented here, at all.

Step 2.1 represents a decision to be made for each task that has been defined in the specification: can the task be instantiated – fully or partially – by an existing, freely or commercially available tool, or does it have to be created from scratch? There are a lot of aspects to be factored into these decisions, e.g. the availability of appropriate tools, their level of interoperability, the required level of control over a tool's implementation, or the ability to customize it, etc. Obviously, there is a tradeoff to be made between the effort required to implement a custom tool, and the level of control, flexibility and adaptability a generic tool usually cannot offer. However, even with a readily available tool, the effort required to integrate it into the overall toolchain can also be prohibitive.

Step 2.2 aims at combining all tools into an integrated toolchain. This is potentially simpler, or even trivial, if all or most of the tools are custom-made, as they can be designed for interoperation. In fact, the whole tool support could be instantiated by a single, monolithic application realizing the whole toolchain. A lack of modularity in such a solution will impede its evolution, though, should the project's parameters change. This step consists of three substeps: creating *adapters*, creating *transformers*, and creating the *coordination logic* between all the tools.

Step 2.2.1 is about the tools' interfaces and the way they can be addressed. For example, one tool might offer a command-line interface, another has a programming interface (API) in a certain programming language, and yet another can be accessed as web service. To enable different tools to interoperate with each other, or be coordinated by some central controller, some translation is needed, so that the tools can speak each others languages, or are made to all speak a common "lingua franca". This is referred to as creating *adapters* for the tools, and is a prerequisite to define control flow of toolchains. The metaphor of "speaking the same language" does not extent to data the tools are exchanging, however.

Step 2.2.2 complements the adapters with *transformers* to take care of differing ways in handling data. There are different levels of such a transformation, from just changing the representation or format of the data (e.g. data marked up as XML vs. JSON), or transforming the way the data is modeled (e.g. between tool- or vendor-specific schemas for modeling an abstract syntax tree of a Java program), to translating the data into something entirely different (e.g. deriving a control-flow graph from an abstract syntax tree). The latter case borderlines on being a tool in itself, as opposed to just a transformer attached to one. This situation corresponds to technical subtasks discussed as part of Step 1.1. The former cases represent data transformers needed to realize data flow integration for tools lacking compatible means for data interoperability.

Step 2.2.3 concludes the toolchain-building process by tying everything together. It encompasses the creation of an application that encompasses all the tools, as well as the required adapters and data transformers, implements the toolchain's use cases, and coordinates the tools accordingly, i.e. invokes them in the right order, and passes data between as specified in Step 1.2. While adapters and transformers will be required in almost any case (to a varying degree, depending on the level of interoperability of the

tools involved), this step can theoretically be realized only partially, or left completely un-automated. If the Q-MIG project is any indication, though, manual steps in the toolchain can severely impede progress.

## 3.2 Toolchain-Building Support Framework Requirements

The objective of this thesis is to provide assistance to domain experts and tool developers in the toolchain-building process by creating a *support framework*, SENSEI, as established in Section 1.2. Using the steps of the toolchain-building process, requirements for such a support framework will be derived, guided by the three sub-objectives, also introduced in Section 1.2, from which the following key evaluation criteria can be derived as

- the degree of *flexibility* afforded to adapt toolchains for experimentation, and to account for evolving requirements,
- the ability to *reuse* existing parts in the creation of toolchains, and
- the overall *productivity* boost conferred by reduced toolchain creation effort.

To elicit requirements, the following sections go through the steps of the toolchain-building process one by one. Each section will discuss the particular challenges, and conclude with one or more requirement sentences summarizing the observed needs. Before that, issues that do not pertain to any step in particular, but to the toolchain-building process as a whole, are addressed here: the lack of separation of concerns, and the diversity of tasks and tools in any given application domain.

Section 1.1 briefly summarized statements of software evolution experts regarding the challenges of toolchain-building. All the issues discussed there come down to a general lack of interoperability of software evolution tools. This means that, before being able to create toolchains, domain experts have to establish those missing interoperability means first (Objective 3 – productivity). This is a task that would be a much better fit for tool developers, as they are the experts for their tools. The fact that these aspects of toolchain-building get mixed up in practice is a major reason for the lack of reusability (Objective 2 – reusability) of integration code, and for its brittleness (Objective 1 – flexibility).

The envisioned support framework has to support both domain experts and tool developers, by freeing the former from having to deal with technical integration issues as much as possible, so they can concentrate on their actual goals, and providing the latter with the means needed to make their tools interoperable. It might sometimes be necessary that the same persons take on both roles, e.g. if suitable tools are not available for all the tasks, and custom ones have to be built. Still, clearly separating between these two roles and associated activities, namely the specification of toolchains on the one side, and the provision of tool interoperability means and the actual integration of tools on the other side.

Drawing this borderline is an absolutely essential, fundamental requirement, which, apart from the argument given here so far, also follows from software design principles such as *abstraction*, *separation of interface and implementation*, and *separation of concerns* [Bourque and Fairley, 2014], which are well aligned with the overall objectives of the thesis.

There is another requirement that is independent of any particular step in the toolchain-building process. It is more straight-forward, and may even seem obvious: to be of maximum use, a support framework must be able to support the whole range of the application domain's techniques and tools.

Many existing software evolution tool suites, workbenches, and integration approaches (see Chapter 5) limit their scope to software analysis and reverse engineering. While this may be appropriate in many scenarios, for industry-scale migration or reengineering projects, these approaches are of limited value, as there is an integration barrier between tools facilitating analysis and fact extraction, and software transformation tools. As an example, large-scale refactorings in a reengineering project need a tight integration between the analysis phase of finding "code smells" [Fowler et al., 1999], and the refactoring phase, which changes the system undergoing reengineering to remove the code smells. The same is true for using refactoring in continuous software evolution.

These two requirements are summed up as follows:

**Separation of Concerns** The support framework must establish a clear separation between toolchain specification and toolchain integration.

**Comprehensiveness** The support framework must support the complete range of tools and techniques available in a given application domain.

### 3.2.1 Task Identification

Building on the goals defined for the toolchain, each one has to be broken down into one or more tasks that can together achieve the overall goal. These tasks can often be performed using standard software evolution techniques, for which tools already exist. Such standard techniques have to be inspected for their appropriateness given project-specific needs, and alternatives have to be gathered and compared.

In the Q-MIG project, only a few tasks were identified which were both recognized as representing standard techniques, and which later got instantiated with third-party tools. Among these was the task of calculating the code clone percentage, for which a wide array of mature clone detection techniques exist. Though there are myriads of publications, including several surveys over the field (e.g. Roy, Cordy, and Koschke [2009]), which help to gain an overview, spotting the relevant properties with respect to project and toolchain integration needs, takes a time- and effort-consuming literature

review. The result of such a literature and market review for Q-MIG has been documented in a report [Pandey, 2014], which shortlisted three tools, and compared them according to nine criteria.

A perceived issue is that the information relevant for supporting the decision for or against certain techniques and tools is not organized in a way which is easily queried for particular properties. Hence, the following requirements to support this step in the toolchain creation process are derived:

**Task Discovery** The support framework must aid domain experts in finding existing techniques relevant to a given task.

**Task Description** The support framework must provide domain experts with a means to describe required properties of tasks in a standardized way.

### 3.2.2 Task Coordination

When all tasks required for each project goal that needs to be tool-supported has been identified, they have to be coordinated in a way that matches the respective goal and produces the desired output. This is – at least – a two-fold step:

First, the tasks consume and produce data. To achieve an overall goal, tasks have to build on each other's results. Therefore, a way to specify data routing is required.

Second, the tasks have to be performed in a certain order to achieve a common, meaningful goal. For simple, batch-like data-processing jobs, a data-driven chain of tasks is sufficient. More complex processes might require the definition of optional, alternative, concurrent or iterative paths. Such workflows require an additional means for control flow specification.

There are more dimensions to task (or tool) integration, a widely used classification scheme being that of Wasserman [1990], which was discussed in Section 4.3.1. For the objectives of this thesis, data and control integration are the essential qualities needed in a support framework for toolchain-building. Deeper issues to be considered in terms of data integration, as discussed in Chapter 4, will be addressed briefly in Section 3.2.5.

The central requirements to support this step are thus considered to be as follows:

**Data Flow** The support framework must aid domain experts in specifying the data flow between tasks.

**Control Flow** The support framework must aid domain experts in specifying the control flow between tasks.

### 3.2.3 Task Instantiation

With all tasks identified and their goal-directed coordination defined, specification is complete. The first step in implementing the toolchain is instantiating tasks, i.e. finding and selecting appropriate tools for their automation.

Finding appropriate tools and evaluating them for their suitability with respect to the capabilities required by a task can be tedious, as elaborated in Section 3.2.1. Analogous to the issue of finding tasks in the first place, matching them up with appropriate tools is not straight-forward, because the information necessary is not arranged for the purpose of software evolution toolchain-building.

Depending on the project's goals and the tasks' required capabilities, there might not always be a single tool to fit the bill. For example, a task to calculate metrics might request a certain set of metrics to be supported. If there is no single tool that can calculate all of them, but several tools each support a subset of metrics so that they complement each other in the desired way, then the support framework should aid domain experts in finding such matches for tasks. Finding solutions "the other way around", i.e. when there are tools which can support more than one of the tasks in the toolchain specification, should be supported, as well.

This is an area where the tool developers also need to be supported, as it is them who supply the tools and information about their capabilities. If they are given a standardized framework to associate their tools with the tasks they can perform, and the capabilities they are offering, it will also ease the job of domain experts.

Another central issue that tool developers, not domain experts, should be responsible for, is the level of interoperability of their tools. Non-interoperable tools mean that domain experts will have to create appropriate solutions themselves. It is reasonable to assume that they will integrate the tools according to their specific needs, leading to tight coupling, and point-to-point, non-reusable integration logic. Tool developers can provide *generic* interoperability means, which only has to be created *once* to work in arbitrary toolchains. A support framework for toolchain-building therefore needs to aid tool developers in making their tools compatible with it.

From the issues described here, these requirements are derived:

**Tool Discovery** The support framework must aid domain experts in finding tools that match their task requirements.

**Tool Description** The support framework must aid tool developers in specifying which tasks their tools can support.

**Tool Interoperability** The support framework must aid tool developers in creating interoperable tools.

### 3.2.4 Adapter Creation

The creation of adapters for tools is necessary, because no assumptions can be made regarding the nature of existing tools' interfaces. They may offer command-line or graphical user interfaces, APIs for particular programming languages, or expose its services via standards like CORBA [*CORBA* 2020], or SOAP/WSDL- [Booth et al., 2004] or REST-based [Fielding and Taylor, 2002] web services.

The only way to remedy this issue, and to avoid having to create custom adapters, is to have all tools agree on a single interface standard, or enforce a kind of "superstandard" that itself can provide bindings to the different kinds of aforementioned interfaces. Partly, this is covered by **Tool Interoperability**, which calls for corresponding support for tool developers to provide compatible adapters as part of their tools. To dispense with custom adapters, in addition, the support framework has to dictate a single, common interface standard, which is embodied by the following requirement:

**Uniform Interfaces** The support framework must impose standardized, uniform interfaces on tools.

### 3.2.5 Transformer Creation

The creation of transformers is kept conceptually distinct from the creation of adapters in the previous step: Adapters are about imposing uniform interfaces so tools can be addressed and *controlled* in a consistent, generic way. Transformers are about the *data* that gets consumed, produced, and exchanged between tools. Existing tools often expect their data in specific, non-standardized formats, so to interoperate, data has to be transformed to be exchanged.

As already discussed in Section 3.1.2 (Step 2.2.2), the differences can be in both concrete and abstract syntax – data differences on the level of semantic domains should be addressed by explicit tasks in the specification phase. These differences can be remedied by appropriate mappings between the tools' data formats. In an ad-hoc integration, this is done point-to-point, with the clear drawback of having to have $n^2$ transformers implementing appropriate mapping rules for $n$ different data formats. Worse, those transformers will not be easily reusable, without putting some thought into it and designing an integration architecture that supports it.

Therefore, the minimum requirement for a proper support framework for toolchain-building is to make transformers first-class citizens, so they can be built in a standard way, and used to fill up a library for future reuse.

On closer look, there is a lot more room for optimization in this area: For example, software evolution projects are often data-intensive, which means processing it all can take substantial amounts of time, even if the supporting toolchain is fully integrated. This can impede projects and limit their agility, improving upon which is part of the

core objective of this thesis. Such issues could be addressed by keeping data in different formats *synchronized*, i.e. only propagating changes instead of re-transforming whole data sets. Aspects of this will be discussed in Chapter 4; however, the focus of this thesis is firmly on supporting the process of toolchain-building. Therefore, its scope is clearly delimitated against any deeper investigation of data integration, and the proposed solution will be expected to only support the following:

**Reusability** The support framework must promote reusability of data transformers.

### 3.2.6 Coordination Logic Creation

The last step in building toolchains is the creation of the coordination logic, which controls individual tools as previously specified, and passes data between them, going through adapters and transformers as necessary. It allows domain experts to use toolchains as seamless, single entities.

When toolchains need to change (because of evolving project parameters), parts of the coordination logic usually have to be adapted, or discarded and rewritten from scratch. This, again, limits the agility of software evolution projects. With the requirements already in place, a lot of the complexity and effort required for creating and adapting this "glue code" is already taken care of: **Separation of Concerns** enforces that all steps of the toolchain-building process are kept clearly separate, so that this step has really only to deal with the creation of the actual coordination logic, and everything else has been taken care of, where possible in a reusable manner.

There is no reuse value to the remaining code for the tools' coordination, though, as it embodies the individual needs of the projects it is written for. However, these individual needs can be assumed to have been specified as per the requirements on **Data Flow** and **Control Flow**. The coordination logic that makes the specifications executable is therefore completely *discardable*. Furthermore, with **Uniform Interfaces** allowing to make simplifying assumptions regarding tool interfaces, and with **Reusability** taking care of incompatible data, the coordination logic becomes largely, if not completely, automatically *derivable*.

The domain experts can therefore be supported to have to evolve their toolchains on specification level, only, which is far simpler, as it abstracts from any technical interoperability issues, and also a lot faster, allowing for shorter toolchain development turnarounds, and more agile projects. The execution of toolchains according to specifications should be completely taken care of by the support framework:

**Automatic Coordination** The support framework must provide automatic tool coordination and toolchain execution in conformance with its specification.

## 3.3 Summary

All requirements are directed at the *core objective* of the thesis (see Section 1.2). An overview over all of them is provided in Table 3.1.

The first and last requirements are arguably special: The first is a *sine qua non* for most of the following requirements. The demand for strong separation of concerns [Parnas, 1972], which is substantiated by the remaining requirements, is the key foundation to enable *reuse* (Sub-Objective 2).

The latter requirement is the opposite, being enabled by the conditions set by the previous requirements, and providing the most marked payoff by demanding automation, to increase *flexibility* (Sub-Objective 1) and overall *productivity* (Sub-Objective 3).

The objectives of the thesis, and the requirements refining them, are the starting point for deriving the key concepts and making the design decisions underlying SENSEI. Part II of this thesis is dedicated to reviewing existing technologies and other related work, whose properties can help support and fulfill the devised requirements. The requirements will be used to assess the suitability of existing methods and techniques to use as a basis for creating the toolchain-building support framework, as well as to delimit this thesis and its proposed approach, SENSEI, from related work.

To this end, existing techniques and approaches will be rated on a simple, four-scale range to reflect the degree of support for each of the requirements. These degrees are:

**No Support** The approach does not provide any means specifically aimed at supporting the requirement, nor does it comprise any features that could be applied to support it (with less effort than otherwise required). It may rely on concepts or techniques that would actually counteract or impede the requirement's satisfaction.

**Foundational Support** The approach does not necessarily provide any means specifically aimed at supporting the requirement, but it comprises features that can serve as a sound basis for its realization.

**Substantial Support** The approach provides means specifically aimed at supporting the requirement, or it comprises features that can be applied to support it with substantially less effort than would otherwise be required. It has some limitations, requires modest modifications, or some other additional work to fully satisfy the requirement. Offering substantial support does not necessarily imply that it is easy to extent an approach to fully satisfy a requirement.

**Full Support** The approach provides means specifically aimed at supporting the requirement, satisfying it fully. While no additional effort is needed to support this particular requirement, using the approach might necessitate adaptions with respect to the remaining requirements. In particular, an approach supporting one requirement fully may still rely on concepts or techniques that would counteract or impede satisfying one or more of the other requirements.

| Name | Requirement | Support |
|---|---|---|
| Separation of Concerns | The support framework must establish a clear separation between toolchain specification and toolchain integration. | Process |
| Comprehensiveness | The support framework must support the complete range of tools and techniques available in a given application domain. | |
| Task Discovery | The support framework must aid domain experts in finding existing techniques relevant to a given task. | Task Identification |
| Task Description | The support framework must provide domain experts with a means to describe required properties of tasks in a standardized way. | |
| Data Flow | The support framework must aid domain experts in specifying the data flow between tasks. | Task Coordination |
| Control Flow | The support framework must aid domain experts in specifying the control flow between tasks. | |
| Tool Discovery | The support framework must aid domain experts in finding tools that match their task requirements. | Task Instantiation |
| Tool Description | The support framework must aid tool developers in specifying which tasks their tools can support. | |
| Tool Interoperability | The support framework must aid tool developers in creating interoperable tools. | |
| Uniform Interfaces | The support framework must impose standardized, uniform interfaces on tools. | Adapter Creation |
| Reusability | The support framework must promote reusability of data transformers. | Transformer Creation |
| Automatic Coordination | The support framework must provide automatic tool coordination and toolchain execution in conformance with its specification. | Coordination Logic Creation |

Table 3.1: Summary of requirements for a toolchain-building support framework.

CHAPTER 4

# Tool Integration

For software evolution projects to be successful, its activities must be tool-supported, and the tools need to be integrated into a toolchain to automate the processes. As argued in Chapter 1, this challenge remains unresolved in general, with existing approaches (see Chapter 5) addressing it only partially. This chapter provides historic context, and introduces basic concepts, terminology, and diverse classification frameworks to arrive at a broad, multi-perspective view of tool integration. It will be used to place and delimit SENSEI, and will serve as a common reference frame for the remainder of this thesis, particularly for the review of related work presented in Chapter 5.

In the following, a brief retrospect of the origins of tool integration is given, and lessons for the SENSEI approach are derived (Section 4.1). Terminology is discussed in Section 4.2. Section 4.3 presents classification and partitioning schemes of tool integration. For each one, an embedding of SENSEI is given, refining objectives and delimiting the scope of this thesis. The chapter concludes with a summary in Section 4.4.

## 4.1 A Brief History of Tool Integration

Tool integration became a field of research beginning in the 1980s, with many authors [Asplund and Törngren, 2015; Boehm, 2006; Wasserman, 1990] referring to Buxton and Stenning [1980] as one of the earliest works of the area. Software engineering as a discipline had been in its infancy before that, being born in the late sixties [Naur and Randell, 1968]. In the intervening years, structured programming was developed [Dahl, Dijkstra, and Hoare, 1972; Dijkstra, 1968], and the development process began to be analyzed and considered explicitly [Royce, 1970], to give just two examples. Also, the realities of both computer users and software developers witnessed a major shift from mainframe computing to mini computers to workstations and personal computers [Wirth, 2008].

Taking the first steps from computer programming as an art towards a more rigorous discipline of software development, as well as a widened scope that, for example, recognized requirements analysis, design, and quality assurance, as important activities of the overall process, led to a need for additional, different, and more integrated tools. The availability of personal workstations with color displays able to depict graphical user interfaces enabled completely new kinds of tools [Endres, 1996]. This led to research and development of *Integrated Project Support Environments* (IPSE) in the eighties (Section 4.1.1) and *Computer-Aided Software Engineering* (CASE) tools in the nineties (Section 4.1.2). Key takeaways and their relevance for this thesis are summarized in Section 4.1.3. Further developments in tool integration, particularly in software evolution and embedded systems development, will be reviewed in Chapter 5.

### 4.1.1 Integrated Project Support Environments

By the beginning of the eighties, there was a rising awareness for the need for explicit, structured development processes [Boehm, 2006]. With many tools having become available to support individual activities and steps of the overall development process, there was a desire to integrate them to reduce the amount of error-prone, manual effort through better automation. This led to academic interest and several (large) projects researching *Integrated Project Support Environments*: started off by the Stoneman project, aimed at developing a comprehensive development environment for Ada [Buxton and Stenning, 1980], these projects included *Aspect* [Brown, 1988], *Eclipse*[1] [Bott, 1989], and *ISTAR* [Dowson, 1987], among many others. Besides the IPSEs themselves, important outcomes of these research efforts were the interface standard *Portable Common Tool Environment* (PCTE) [Long and Morris, 1993], and the *CASE Data Interchange Format* (CDIF) [Parker, 1992], soon to be replaced by *XMI (XML Metadata Interchange)* [2015].

The standardization efforts also led to the development of the NIST/ECMA *Reference Model for Frameworks of Software Engineering Environments* [Martin, 1993] and the *Reference Model for Project Support Environments* [Brown et al., 1993]. The former is a large catalog of *services* that environments should provide. A major goal was to enable side-by-side comparisons of IPSEs and standards like PCTE, by mapping their respective concepts to the reference model's services.

The latter focuses on services supporting concrete software engineering activities, as opposed to integration infrastructure services. Regarding terminology, notable distinctions are made between human-performed *tasks* and machine-provided *services* on the one hand, and conceptual machine functionality described as *services* and its actual implementation in *tools* on the other hand. Figure 4.1 illustrates these relationships.

[1] Unrelated to the well-known, open-source IDE and rich-client platform, which originated at IBM, and is now developed under the auspices of the Eclipse Foundation [*Eclipse Modeling Project* 2020].

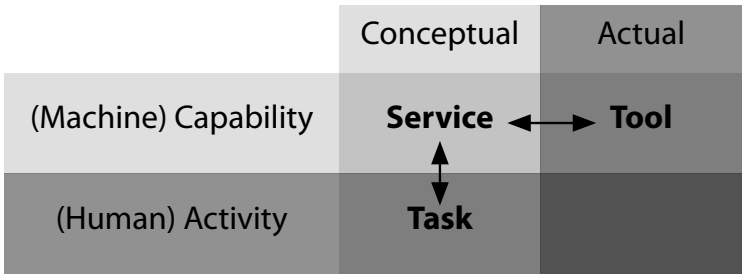



Figure 4.1: Faithful reproduction of a diagram found in Brown et al. [1993], delimiting the terms *service*, *task*, and *tool*.

### 4.1.2 Computer-Aided Software Engineering

By the Nineties, IPSE research had evolved into the study of CASE tools (*Computer-Aided Software Engineering*). The origin of the term is attributed by Chikofsky [1988] to an article that had appeared in the Wall Street Journal in 1986. Brown and McDermid [1992] criticize the efforts made within the IPSE movement as being focused too much on framework infrastructure and integration issues, while failing to address "broader issues like user functionality and productivity". Boehm [2006] uses similar language, describing IPSE research as "overfocused", and CASE as having "broadened their scope". This may be considered curious, given that IPSE research *had* to provide the very foundations, i.e. integration infrastructure, since there had not been any integrated environments before. Brown and McDermid further find fault with PCTE as being too "broad and complex", and lacking concrete software engineering support. They make the point that, even though IPSEs were all about integration, the whole concept of what (tool) integration is was not well understood at all.

This last issue raised might truly be a weak spot. Indeed, the first major treatise considering tool integration *ab initio* is widely regarded to be by Wasserman [1990], and there are more recent indications that the basic principles of tool integration might still not be very well understood by the scientific community [Wicks and Dewar, 2007].

Brown and McDermid [1992] and Sharon and Bell [1995] distinguish IPSE and CASE along the following aspects:

- IPSEs are independent of specific development methods, while CASE tools are not extensible.
- IPSEs support collaborative work, while CASE tools are aimed at single users.
- IPSEs have been applied mainly in research and engineering, while CASE tools address information system development.
- IPSEs provide necessary foundations to integrate tools of potentially different vendors, while CASE products are usually by a single vendor (incompatible with the tools of competing vendors).

Again, these criteria, the first two in particular, seem to indicate that in fact IPSE research had the much *broader* scope. In contrast to IPSE, CASE has focussed on tool-supported software engineering *methodologies* rather than tool integration. Some authors have used the term *ICASE* to highlight the integration aspect of a CASE tool discussion.

CASE technology, as characterized by Schmidt [2006], often prescribed the use of graphical modeling and programming languages, and was aimed at facilitating code generation from such high-level artifacts. For a brief time, CASE tools were very successful, commercially. However, studies showed [ElShazly and Grover, 1993; Lending and Chervany, 1998] that most companies that invested into the technology only used the tools briefly before dismissing them again, if they were ever properly adopted, at all. By the mid-nineties, scientific interest into the field had also waned, with relevant conferences and workshops disappearing, or dropping the term "CASE" from their title to reflect a changed focus of research [Ocampo, Albizuri, and Botella, 1998].

Reasons given for the failure of most CASE tool efforts in the literature include:

- steep learning curves [Kemerer, 1992; Ocampo, Albizuri, and Botella, 1998], and a markedly more formal process developers were not used to, and were not trained for [Lending and Chervany, 1998],
- unmotivated developers, who found the tools, often forced upon them by management, unhelpful [Chau, 1996; Lending and Chervany, 1998],
- inflexibility for lack of customizability [Kelly and Tolvanen, 2008, pp. 360f], [Schmidt, 2006], with methodologies hard-wired into each tool [Endres, 1996],
- vendor lock-in for lack of standardization [Stahl et al., 2006, pp. 12-13],
- graphical modeling languages ill-matched for the target platforms, which were the operating systems, not higher-level, full-featured middleware solutions, and code generation technology too immature to bridge this gap [Schmidt, 2006].

### 4.1.3 Lessons Learned

In more recent works on tool integration, IPSE is hardly discussed and seems mostly forgotten. The oft-cited, extensive standard by Martin [1993] is often reduced to a single figure, sometimes referred to as "toaster", that was featured prominently in the original document, and was meant to give an overview of the logical organization of services into categories, and how they supported the integration of individual tools. Instead, it was interpreted as a reference architecture by many authors, whose disregard of the standard's comprehensive set of services seems to have aggravated its authors to the point that they removed the figure and put it into the appendix in later editions.

This is regrettable, because the crucial distinction made here between conceptual services and actual implementations is lost, for example, in most treatises of service-orientation. Based on their analysis, Brown, Feiler, and Wallnau proposed an integration scheme for *federated environments* resting on *services*, curiously applying princi-

ples of service-oriented architecture (and using its terminology, as in "flexible, service-oriented integration") roughly a decade before the SOA hype (see Chapter 7).

Brown, Feiler, and Wallnau [1992] highlight IPSEs complexity as a result of absolute genericity, as well as a focus on data integration, only, based on central repositories as major issues. CASE technology is criticized too, though, for an overall lack of interoperability. CASE still has a visible presence in tool integration research, although mainly to point out why it has not prevailed. The various reasons for this can be summarized into three categories: ignorance of *human factors*, *insufficient maturity* of the utilized techniques and the targeted hard- and software systems, and *technical unsuitability* of the CASE tools themselves to solve the problems they were advertised for.

Much of the issues CASE tools were not able to tackle have been picked up by model-driven software engineering (MDSE), relying on richer target platforms and programming languages, standards like the Unified Modeling Language [Booch, Rumbaugh, and Jacobson, 1999], research and development of new model transformation languages, and customizability through domain-specific languages (DSLs). However, if the focus of IPSEs was firmly on tool integration, and had already shifted for CASE, MDSE has even more different objectives, being more of a development methodology aimed at portability and interoperability [Kleppe, Warmer, and Bast, 2003, pp. 4f; Gašević, Djurić, and Devedžić, 2006, p. 111].

The design of SENSEI (see Part IV) is informed both by fundamental research findings as well as the mistakes of past tool integration efforts:

- The distinction between *conceptual* and *actual* [Brown et al., 1993] is a major cornerstone of the approach, manifested by services (Chapter 7) and components (Chapter 6), respectively, to facilitate flexibility, reusability, and an abstraction level appropriate for domain experts.
- In contrast to IPSE, and many later tool integration efforts, SENSEI does not focus exclusively on data integration. It is mainly process-oriented and designed to be complemented with existing data integration approaches, if necessary. Section 4.3 will provide classification schemes for different kinds of tool integration, and illustrate the place SENSEI takes within them.
- SENSEI is designed to be generic (see Section 3.2), light-weight, and technology-agnostic, to avoid the issues that have been cited for the failure of most CASE tools, i.e. vendor lock-in and inflexibility.
- While SENSEI aims to be generic, it avoids over-standardization by focusing on providing practitioners the means to model concepts and integration solutions of their problem domain themselves, instead of trying to anticipate every conceivable integration task with a fully comprehensive catalog of services.
- Instead, it relies on *model-driven* techniques (Chapter 8), which evolved from CASE origins, to bridge conceptual and implementation layers, and facilitate automation through model transformation and code generation.

## 4.2 Basic Terminology

Up until this point in the thesis, the terms *tool*, *toolchain*, *tool integration*, and *interoperability* have already been used without a formal definition. One obvious source for this terminology is the ISO/IEC/IEEE 24765 [2017]:

**Definition 4.1: Software Tool**

Several alternative meanings for just the term *tool* are offered by the standard, but a more appropriate definition is provided for the term *software tool* as follows:

> a computer program used in the development, testing, analysis, or maintenance of a program or its documentation.
>
> a software product providing automatic support for software life-cycle tasks. ISO/IEC/IEEE 24765 [2017, p. 424]

In this thesis, "tool" can always be assumed to refer to a software tool, and, more specifically, will often be a software *evolution* tool, as software evolution toolchain integration is the major motivation for this thesis. A tool, by this definition, supports development and maintenance of a software system, itself being a piece of software.

An important distinction for tools is whether they are *interactive* or not: *interactive tools* guide and support users performing a specific activity – there is a constant back and forth (i.e. input and output) between the user and the tool, e.g. modeling a class diagram with the help of a UML tool. *Non-interactive tools* are invoked by users to perform a task fully automatically. The tool receives input when it is started, describing the task it should do, how it should do it, and on what data it should operate. It then performs the task without requiring further human intervention, and outputs results while it is running, and/or when it terminates. Examples for these kind of tools include parsers, fact extractors, transformers, code generators, metric calculators – in short, the tools from the Q-MIG example (Chapter 2) fall into this category. Because these tools are used (interacted with) differently, they also require distinct integration approaches. As this thesis is motivated by the need to automate software evolution processes, it focuses on integrating non-interactive tools.

In a seminal work of the field, Wasserman [1990] defines *tool integration* as follows:

> tool integration is intended to produce complete environments that support the entire software life cycle.

This refers to the objective of tool integration, and unsurprisingly, given the paper's topic, there is a focus on creating software engineering environments. Wasserman considers five distinct types of tool integration – platform, presentation, data, control,

and process integration – which is the central contribution of the paper. These types, along with classification schemes by other authors, will be discussed in Section 4.3.

To close in further on an appropriate definition for tool integration, its intended outcome is considered. Wasserman seems to consider that to be a *software engineering environment* – a term used here to subsume modern IDEs, but also IPSEs and CASE suites [Terry and Logee, 1990]. However, such an environment might be meant to support many different software engineering activities, some of which might have little to no overlap in terms of their supporting tools. Therefore, not all tools have to be fully integrated with all other tools contained within an environment to provide adequate support. More importantly, this kind of integration is usually aimed at interactive tools. In general, the result of integrating tools will be referred to as *toolchains* (as is done by other authors, such as Biehl [2013] and Erdmenger and Uhlig [2011]):

**Definition 4.2: Toolchain**

A toolchain consists of multiple individual tools, that have been combined in a way that allows them to conjointly support one or more particular activities by fully automating their execution.
An *automated* toolchain is a toolchain that eliminates all need for human interaction and performs the supported activities fully automatically.

A toolchain is not restricted to be an actual *chain*, i.e. there can be control flow branches or concurrency. To stress non-interactivity, the notion of *automated toolchain* is also introduced. However, this distinction will rarely be made, as this thesis only considers non-interactive toolchains. So, unless explicitly noted otherwise, a toolchain in the context of SENSEI is always meant to refer to an automated toolchain. Interactive and non-interactive integration schemes can complement each other, though, which will be briefly discussed as an outlook in Section 16.4.2.

Returning to *tool integration*, Thomas and Nejmeh [1992] state:

> “Tool integration is about the extent to which tools agree. The subject of these agreements may include data format, user-interface conventions, use of common functions, or other aspects of tool construction.”

This alludes to Wasserman’s integration types, on which their work is based. They point out another important aspect, namely that there are (at least) two perspectives towards tool integration that have to considered: that of the *users*, and that of the *builders* of integrated tools, coinciding with a desire for *ease of use*, and *ease of integration*, respectively.

Still, these definitions remain rather abstract, partly because they fail to mention, or only vaguely allude to, the *underlying objective* of tool integration. Obviously, tool integration is not pursued as an end in itself. This is what Brown and McDermid [1992] criticized about IPSE research – having lost sight of the actual goal of better supporting

software developers. Yang and Han [1996] generalize this, by pointing out that the ultimate goal of tool integration is *increasing productivity* (which aligns with the objectives of this thesis, see Section 1.2). They go on to classify different ways to achieve control and data integration, and define three approaches of integration on different points along a spectrum representing a trade-off between flexibility and reusability on one end, and performance and productivity on the other.

Explicitly considering the aim of improving overall productivity makes integration efforts *testable*, i.e. it allows to check whether it is successful, and even quantify the extent of the improvement. Because integrating tools requires effort itself, this raises the question, pointed out by Wicks and Dewar [2007], of whether there is a point at which a more sophisticated integration solution would actually *decrease* the overall productivity. In essence, tool integration is aimed at reducing *accidental complexities* [Brooks, 1987], e.g. removing the need for human intervention as much as possible to reduce the room for human error. A different accidental complexity arises from the need to transform back and forth between different data formats of the individual tools involved (unnecessarily increasing time complexity), but optimizing for performance may be in conflict with other objectives such as reusability, as stated above. The different aspects must be traded off against each other, to find an integration strategy that offers the highest overall productivity gain. In summary, the following definition will be adopted:

**Definition 4.3: Tool integration**

Tool integration comprises the combination of tools into a toolchain to support a particular software engineering activity with minimal accidental complexity and maximum productivity gain.

What is not addressed by this definition is an issue raised by Thomas and Nejmeh [1992], namely that the views of two stakeholder groups must be distinguished: toolchain users and toolchain developers[2]. Users of toolchains need it to be *well-integrated* to be productive, while toolchain developers need the individual tools to be *easy to integrate*. These different views are, again, highly related to the trade-off between a performance-tuned toolchain versus its flexibility and the reusability of its parts.

Although Thomas and Nejmeh use the term differently, *interoperability* is usually taken as the property that expresses ease of integration. For example, Wegner [1996] defines it as follows:

> “Interoperability is the ability of two or more software [tools] to cooperate despite differences in language, interface, and execution platform.[3]”

[2] In software evolution projects, these roles might be fulfilled by the same people, which is part of the problem.

[3] Wegner addresses interoperability of software *components*. The definition has been narrowed down

This is a rather broad definition, but it establishes an important point, namely that interoperability is a property of a *set of tools*. This makes sense: individual tools may be prepared to work in concert with others, but if they do not all make *compatible* preparations, then this will not necessarily improve their interoperability.

The IEEE standard ISO/IEC/IEEE 24765 [2017] offers three definitions for interoperability (though none for *tool* interoperability, specifically), one of which is heavily linked to CORBA technology, and is therefore of little general use here. The other two definitions for interoperability are:

> “ [T]he ability of two or more systems [...] to exchange information and to use the information that has been exchanged. ”

> “ [T]he capability to communicate, execute programs, and transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units. ”

Notably, the first definition focuses on the ability to exchange data, only, which is compatible with the views of Thomas and Nejmeh. In contrast, the definition by Wegner does not consider data exchange explicitly, at all. The latter definition offered by the ISO/IEC/IEEE 24765 is the most comprehensive one, but it is not really phrased in terms of integration ease. In this thesis, the following definition is adopted:

**Definition 4.4: Tool Interoperability**

Tool interoperability is a property of a set of tools, and a measure proportional to the amount of effort required to effectively integrate these tools into a toolchain.

This wording intentionally refrains from explicitly referring to any particular means of interoperability. By this definition, it is therefore irrelevant *how* interoperability is achieved; it is only measured in terms of integration ease. Requiring the integration to be *effective* is meant to stress the need to minimize accidental complexity, and to maximize productivity gain, as per Definition 4.3.

There have been several approaches put forward to classify and measure the degree of interoperability, such as *Systems of Systems Interoperability* (SOSI, Morris et al. [2004]) and the *Levels of Conceptual Interoperability Model* [Tolk and Muguira, 2003]. Gürdür, Asplund, and El-Khoury [2016] provide a brief overview.

Concluding the terminology discussion, an important takeaway, aside from the provided definitions themselves, is that tool integration spans a wide field with many aspects that are emphasized differently by different groups in the overall tool integration community. The next section therefore provides several breakdowns of, and classification schemes for, tool integration by different authors, to sub-divide the field as a whole, and be able to more precisely place this thesis within it.

here by replacing them with *tools* to avoid confusion. In in-depth introduction to software components is provided in Chapter 6.

## 4.3 Dimensions of Integration

The cornerstone for the study of tool integration from first principles was laid by Wasserman [1990], to which most works on tool integration refer. Other authors have elaborated his model further [Thomas and Nejmeh, 1992], or advanced independent classification schemes [Brown and McDermid, 1992; Yang and Han, 1996] that may be placed within Wasserman's categories. Also of interest are classification schemes aimed at the infrastructure used as an integration basis, described in terms of design patterns [Karsai, Lang, and Neema, 2005], and in terms of sophistication and scope of the resulting solution [Fuggetta, 1993].

It is important to understand these different aspects of tool integration to refine the objectives of this thesis, and to place it within a frame of reference for clear delimitation and comparison with related approaches. The subsequent sections will review the classification systems of the aforementioned authors, followed in every instance by a placement of SENSEI within each respective framework.

### 4.3.1 Integration Types According to Wasserman

Wasserman [1990] distinguishes between five types of tool integration: *platform integration*, *presentation integration*, *data integration*, *control integration*, and *process integration*.

**Platform integration** refers to the provision of a basic, common infrastructure, that also serves as virtual environment and abstraction layer that hides underlying discrepancies in terms of hardware, software, and network distribution. In today's terminology, this role is taken on by *middleware* solutions (e.g. service-oriented middleware like Tuscany SCA [Laws et al., 2011] and *WSO2* [2020]).

**Presentation integration** refers to the provision of a "consistent user interface". Particularly in terms of *graphical* user interfaces, this requires all tools to adhere to the same design language, or human interface guidelines.

**Data integration** refers to the provision of data sharing and management facilities. For tools to be able to use and manipulate each others data, compatibility of data formats, models, and in-memory representation must be established, as well as protocols to access the data (these aspects of data compatibility are discussed in Section 12.1.3).

**Control integration** refers to the provision of event notification and tool invocation facilities. For tools to be able to notify and invoke each others functionality, compatibility of interfaces (describing *what* can be accessed) and protocols (describing *how* to access it) must be established.

**Process integration** refers to the provision of facilities for explicit representation, design, management, and automation of development processes consisting of individual steps that require the support of multiple tools working in concert.

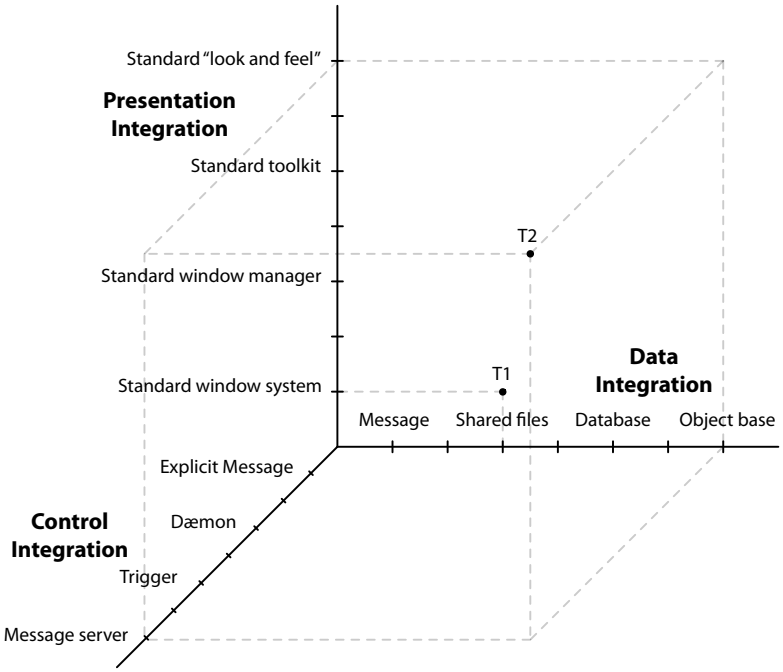



Figure 4.2: Faithful recreation of "[t]hree dimensions of tool integration" as presented by Wasserman [1990].

Platform integration is a foundational layer allowing tools to make basic assumptions about each other, and establishing a certain degree of uniformity. It is therefore usually not independent of data and control integration, even though Wasserman seems to treat all types as independent dimensions. Platforms at least provide basic means for data exchange and invocation, e.g. through the file system and shell of an operating system (arguably, the shell also offers rudimentary presentation integration, and basic graphical user interface integration may be provided by a window manager). Modern middleware solutions offer sophisticated integration environments, e.g. message-oriented middleware (enterprise service buses). Also, agreement on a particular level of data integration may dictate means of control integration, and vice versa. It should be noted that Wasserman [1996] briefly reiterates his integration types, with a definition for platform integration narrowed down to providing network transparency, only.

Process integration is addressed only briefly by Wasserman [1990], as both process management tools, as well as software processes in general, were still in their infancy. It is also rather vaguely defined: the definition given here is influenced by the field of workflow management, and by other authors discussing this aspect in the context of tool integration [Asplund and Törngren, 2015]. Asplund and Törngren point out that Wasserman himself redefined process integration later [Wasserman, 1996], although

this might just appear this way because of the condensed, brief form of the second definition. While tool integration research has seldom addressed this aspect, it is the main concern in workflow management, business process automation, and scientific workflow systems (see Section 5.8 and Section 7.5).

For each of the remaining integration types, presentation, data, and control, Wasserman gives some examples of concrete means to achieve integration in these "dimensions". Figure 4.2 provides an overview. Each point in this space identifies a combination of integration techniques. Tools that can be located on the same, or on neighboring points, are more interoperable than those not sharing any techniques on either axis.

The methods to achieve the different types of integration that are shown on the axes in Figure 4.2 are to be taken as examples – they neither represent a comprehensive list, nor should their arrangement necessarily imply any notion of some methods providing "better" integration than others. Generally speaking, the outer methods can be considered more sophisticated. The depicted integration methods are only selected examples, some of which are a bit dated.They are also actually too abstract to fix a particular point in the space. For example, point *T1* uses "shared files" for data integration, but for a working integration this needs to be narrowed down to a concrete file system – Wasserman gives the Unix file system as example.

Thomas and Nejmeh [1992] offer an extension to Wasserman's framework, refining his integration types by defining interoperability properties for them. They can be useful to classify and compare tools more precisely in terms of interoperability measures provided by them. For example, data integration is subdivided into *conformity*[4] and *data exchange* (agreeing on common data models for persistent data and for ad-hoc data exchange, respectively), *non-redundancy* (sharing data instead of replicating it), *data consistency* (preserving consistency as multiple tools manipulate the data), and *synchronization* (communicating with other tools directly, as opposed to exporting and importing data). Their work also shows the multi-facetted nature of tool integration, for which Wasserman provides fundamental, but necessarily coarse-grained, classes.

**Placement and Delimitation of SENSEI in Terms of Wasserman**

The requirements elicited from analyzing the toolchain-building process (Chapter 3) emphasize supporting tool and toolchain builders, as opposed to toolchain users. Therefore, SENSEI aims to provide the means and methodology for the creation and evolution of tailor-made tooling, not the tooling itself.

Due to the focus on non-interactive toolchains (Section 4.2), *presentation integration* is considered to be out of the scope of this thesis, which positions SENSEI somewhere in the bottom plane of Figure 4.2. All other types are taken into account: As a general prerequisite, *platform integration* will be addressed by component-

[4]Actually referred to as *interoperability* in the original, and renamed here to avoid confusion with terminology of this thesis, as established in Section 4.2.

**Placement and Delimitation of SENSEI in Terms of Wasserman (cont.)**

based and service-oriented concepts and technology, reviewed in Chapter 6 and Chapter 7, respectively. Since SENSEI is about automating processes, *process integration* is maybe the most important integration type. Model-driven engineering (Chapter 8) will provide essential ingredients to derive a language for modeling processes, and to automatically transform them into executable toolchains. *Control* and *data integration* are prerequisites of process integration. SENSEI emphasizes the former over the latter due to its process-centric view; this will be elaborated further in the context of the following classification schemes.

### 4.3.2 Integration Levels According to Brown and McDermid

Brown and McDermid [1992] offer a classification of integration that overlaps with that of Wasserman [1990] to a degree, while other aspects (team and management integration) go beyond. However, these non-technical issues are not within the scope of this thesis. The technical aspects are *interface* integration, which maps to Wasserman's presentation integration; *process* integration, which is similar, yet somewhat narrower than Wasserman's type of the same name; and *tool* integration, which is defined mostly in terms of data integration, but also seems to include control integration aspects.

While these categories do not add considerable value to the discussion at this point, the authors went on to further sub-divide their concept of tool integration into five levels, which can be considered orthogonal to the Wasserman-based classification. They are expressed here in terms of individual tools, i.e. describing levels of *interoperability*. Integration solutions and toolchains can be classified in terms of the minimum level of interoperability they require from tools to be applicable. From lowest to highest sophistication, and amount of common conventions, the levels are the following:

**Carrier level** refers to tools agreeing on no more than a common representation form for all data, e.g. UNIX command line tools reading and writing byte streams, which allows them to be combined by pipes. Each tool has to perform its own analysis and preprocessing of the input data; assumptions made regarding the structure of the data cannot be made explicit, or be enforced, technically. On this level, basic preprocessing is therefore implemented, and performed, redundantly.

**Lexical level** refers to tools agreeing on data conventions allowing them to recognize, and break input up into individual tokens. This allows them to process parts of an input they understand (e.g. based on particular keywords), while ignoring the rest (which may follow a syntax that is only understood by some other tool). On this level, the same data may have to be processed repeatedly through a toolchain, an example being the processing of a LaTeX document, which usually requires multiple passes of (at least) *latex* and *bibtex*.

**Syntactic level** refers to tools agreeing on the data model (schema), and concrete syntax rules to exchange. The latter may be achieved by using a standard exchange format, while the former is, by its nature, specific to the kind of data being exchanged. On this level, basic data handling can be factored out of individual tools, completely.

**Semantic level** refers to tools agreeing on semantics of operations on data, either by each adhering to semantic conventions implicitly (which couples them together tightly), or by explicitly factoring out data operations. On this level, tools have a common understanding of both the structure and the meaning of the data they process, eliminating the need for multi-pass processing.

**Method level** refers to tools agreeing on common processes, either by each adhering to them implicitly, or by explicitly factoring out process conventions. On this level, tools either understand their role in an overall process, and notify each other about their activities, or processes are represented externally, so tools can be controlled to appropriately support and automate them, accordingly.

The first three levels are degrees of data integration, only. The distinction is useful, as the interoperability of tools on the lower levels is clearly limited. For example, while the UNIX philosophy is a powerful one, carrier-level interoperability is useful for ad-hoc, small-size, simple integrations, only. For integrating diverse tools that have not been designed for each other, this lowest level offers almost no help at all, as potentially complex data transformers will be required for each relevant tool combination. Toolchains integrated at either the lexical or the syntactic level will scale badly for large amounts of data. The syntactic level improves interoperability somewhat, as common data handling logic can be factored out of individual tools, but if based on exchanging whole files, integrations remain inflexible [Sim, 2000].

The semantic level can be achieved with data integration means, only, but real interoperability and flexibility improvements are facilitated only when also introducing control integration. This is necessary if data operations are factored out of individual tools, as they will have to be able to invoke this external functionality.

Method level integration is basically synonymous with process integration, for which both data and control integration means are prerequisites. Brown and McDermid seem to implicitly assume that method level integration must be event-based (Thomas and Nejmeh [1992] make similar assumptions), with tools knowing their role in the processes they are involved in to some degree. However, they also point out that factoring out process knowledge would provide greater flexibility.

**Placement and Delimitation of SENSEI in Terms of Brown and McDermid**

To support process automation, it seems clear that SENSEI must aim for the *method level*, the highest one. The lower levels, concerned with data integration, have attracted the highest attention of researchers [Asplund and Törngren, 2015], meaning

**Placement and Delimitation of SENSEI in Terms of Brown and McDermid (cont.)**

a large body of work already exists regarding these aspects, as will be shown in Chapter 5. SENSEI will build on these foundations, rather than reinvent anything.

Regarding individual tools, a focus will be put on supporting tool developers and domain experts in making them more interoperable, using existing techniques, but ensuring their consistent use, overall. This includes establishing concepts and structures to build data transformers, covering data interoperability up to the semantic level, and adapters, providing method-level interoperability (compare Section 3.2, particularly requirements *Tool Interoperability*, *Uniform Interfaces* and *Reusability*). In terms of integrating these tools into toolchains, SENSEI will provide means for method level integration by "glueing" tools together into toolchains with auto-generated control logic, while being designed to be complemented with existing approaches to address the (data) integration challenges of the lower levels.

### 4.3.3 Integration Patterns According to Karsai, Lang, and Neema

Karsai, Lang, and Neema [2005] provide a constructive means to distinguish tool integration approaches in the form of two *architectural patterns* (in the sense of Buschmann et al. [1996, p. 12]). Both patterns use model-driven foundations for data integration, providing mechanisms to map data model concepts of different tools onto each other, and transform data accordingly. They also both make use of *tool adaptors* to provide uniform interfaces, and *semantic translators* to take care of data conversion. The patterns apply these means differently, though, with one being based around a single *integrated model*, and the other focusing on *process flows*:

**Integration based on integrated models** refers to an architecture using a common, integrated metamodel, combining all concepts of the tools to be integrated. Such a model needs to be tailor-made for those tools to be able to represent specific features and peculiarities of each tool, and not lose data while exchanging it. Tool adaptors connect individual tools to a *common model interface* (e.g. a messaging middleware like service buses). An *integrated model server* also connects to this interface, and uses semantic translators to transform tool data into formats of the integrated metamodel for storage, and back into tool-specific formats.

**Integration based on process flows** refers to an architecture using explicit workflow representations that can be enacted to support and automate processes using tools with direct tool-to-tool data integration. The architecture uses tool adaptors to connect individual tools to the *backplane* (again, modern middleware stacks can be used to fill this role). Semantic translators also connect to the backplane and transform between the formats of tool pairs needing to exchange data.

The major prerequisite for using the first pattern is significant overlap in the concepts used by individual tools, i.e. it is most appropriate if all tools basically manipulate the same (data) artifact, but potentially from different views, or focussing on different details and aspects. Tools can operate in any arbitrary sequences, relying on the fact that changes of one tool become available to all, and will be reflected, accordingly.

In the second pattern, the prerequisite is the opposite: each tool should only be linked to few other tools. It is most appropriate for tools that cooperate in a well-defined, step-by-step process, where tools only interoperate with direct predecessors and successors in the toolchain. Data is not persisted, centrally, but send directly from tool to tool, and tool invocation in accordance with defined workflows can be automated.

Conventional wisdom would suggest that the integrated metamodel approach scales better with the number of tools to be integrated: with each new tool, only one translator has to be created, as opposed to one per tool that is already part of the integration. Put differently, to integrate $n$ tools requires at least $n$ translators with an integrated metamodel, whereas without it, up to $n^2$ tool-to-tool translators are needed.

Karsai et al. have practically applied and evaluated both patterns, and observed the opposite. Even though the case study performed for the integration based on integrated models had small tool metamodels with large overlaps, adding a new tool became significantly more complex as the number of tools increased. In fact, Karsai et al. observed this while integrating the *fourth* tool, and suggest that this pattern is not well-suited to integrate more than three or four tools. In contrast, the case study performed for integration based on process flows showed no increase of effort required to integrate additional tools.

One reason for these results is that integrating metamodels is all but trivial (e.g., see Burger et al. [2016], Meier and Winter [2016], and Tunjic and Atkinson [2015]). Because all concepts are poured into a single metamodel, any change may, in principle, have arbitrary side effects. With tool-to-tool integration, changes are always confined locally. Similar experiences have been made in enterprise integration, trying to create, maintain, and evolve *business-wide data models*: Josuttis [2007, p. 38-39] reports that such endeavors usually had the metamodel grew too complex to remain effectively manageable, and would actually lead to tight coupling, because of concepts of the central metamodel "seeping" through the interfaces and into individual applications.

Also, the math of $n^2$ translators for $n$ tools is a worst case upper bound, that will not be reached in most practical applications. In the simplest case, when the toolchain is *actually* a chain (akin to an assembly line), $n - 1$ translators are required for $n$ tools, at least one less than the integrated metamodel solution.

In summary, both architectural patterns have valid use cases. E.g., Karsai et al. argue that "incremental change propagation" (transmitting changes individually instead of complete data sets) is easier with process flows, while for traceability (following the interrelations of metamodel concepts across tools), integrated models are better suited. Due to their case study results, they favor process flows, but also point out that this

pattern does not prohibit using an integrated metamodel (for some or all of the tools) by adding a tool that plays the role of integrated model server, yielding a hybrid solution.

**Placement and Delimitation of SENSEI in Terms of Karsai, Lang, and Neema**

SENSEI embraces *integration based on process flows*, and the results of Karsai et al. provide strong evidence supporting this decision, given the objectives of this thesis (see Section 1.2): the pattern lends itself to the integration of non-interactive toolchains, and, according to the authors, scales better with the number of individual tools. This is a key prerequisite to ensure the *flexibility* of integrated toolchains. Concepts introduced by Karsai et al., have also been identified in this thesis, by analyzing the toolchain-building process: separating *adapters* and *transformers* from individual tools, and treating them as first-class citizens is crucial for *reusability*.

The adoption of the process flow-based architecture is also another justification for emphasizing control and process integration over data integration. As has been stated before, SENSEI will be designed to be complemented with data integration solutions of arbitrary sophistication. This role can be filled with an integration approach based on integrated models for a hybrid solution.

### 4.3.4 Integration Effectiveness According to Yang and Han

Yang and Han [1996] highlight the effectiveness of tool integration strategies with respect to the *productivity* gain of toolchain users. The main contribution is a classification system consisting of three classes with increasing effectiveness. In addition, they provide subtypes of control and data integration, and three integration *paradigms*; the latter are on a similar conceptual level as the architectural patterns of Karsai, Lang, and Neema [2005], and are therefore not considered any further.

For both control and data integration, four categories of techniques are defined, with an implied increasing level of sophistication: The control integration categories span from invoking tools *indirectly* using their pre-existing user interfaces ($C_1$), over using external *triggers* like database events ($C_2$) and using *message servers* ($C_3$), to direct *procedure calls* ($C_4$). For data integration, the categories prescribe to use either *intermediate files* ($D_1$), databases ($D_2$), message passing ($D_3$), or internal *canonical representations* common to all tools ($D_4$).

Yang and Han argue that control and data integration cannot be viewed in total isolation. The techniques used for either integration type have to be matched up judiciously; some categories, such as $C_3$ and $D_3$ imply each other quite directly.

These categories of integration techniques are referenced by the central classification scheme of their work, which is based on the level of *productivity increase* that can be gained by using tools integrated into toolchains with increasing degrees of sophistication. The three classes distinguished by Yang and Han are the following:

**Class 1: Eliminating user-generated delay** refers to increasing productivity by using basic tool integration that automates data provision for, and invocation of, individual tools. This integration class can be achieved with $[C_1, D_1]$ techniques.

**Class 2: Eliminating tool-generated delay** refers to increasing productivity by using incremental data processing, and parallelizing toolchain execution and essential manual work and input by users. This integration class can be achieved with $[C_4, D_4]$ techniques.

**Class 3: Introducing tool construction** refers to increasing productivity by 1. using toolchains with fine-grained inter-tool synchronization to achieve highly interactive task assistance, and 2. building toolchains from self-contained, independent, reusable tools to reduce the integration effort required. This integration class can be achieved with $[C_2, D_2]$ or $[C_3, D_3]$ techniques.

Another way of thinking of these classes is that they are aimed at removing *accidental complexities* [Brooks, 1987], i.e. factors that require additional effort not because of the task's inherent complexity, but because of how it is performed.

With class 1 integration, the tedious and error-prone manual labor of tool users is reduced to the essential minimum by integrating the tools into a toolchain, and thereby automating tool invocation and data exchange. Yang and Han seem to somewhat downplay the impact of this class of integration. In the experience of this author (e.g., with the Q-MIG project, see Chapter 2), the productivity gain from this level of integration can be very substantial, particularly when factoring in time lost due to human error, and technical, organizational, and communication hurdles, e.g. having to exchange data with other project partners by e-mail. The delays caused by these circumstances were in the order of days, and sometimes weeks, while the running time of individual tools was in the order of seconds, minutes, and occasionally hours. So, even though Yang and Han correctly point out that individual tool performance is exactly the same with and without this level of integration, the room for productivity improvement can potentially exceed the other classes by several orders of magnitude.

Class 2 integration aims at having toolchains run "in the background", while users continue to work. The idea is to cut down periods in which users have to wait on tools to return results to a minimum. This requires more fine-grained data integration for tools to be able to effectively work on incremental changes rather than having to wait for complete data sets to process in bulk, and asynchronous control integration mechanisms. Depending on the tasks and algorithms at hand, i.e. whether small or localized changes can be exploited to reduce execution time, this integration class can offer substantial productivity improvements, particularly for processes that require incremental improvement, and thus repeated execution of supporting toolchains.

Class 3 integration has two very different aims: first, to increase the productivity of users through collaborative interaction with (expert system-like) toolchains. And second, to increase the productivity of tool and toolchain developers. The former is

arguably a natural progression of Class 2 that further deepens integration, while the latter is not considered by the first two classes at all.

It is not clear why these two aspects have been combined into a single class. One argument of Yang and Han is that the kind of tools involved are potentially very different (in terms of implementation technology, for example), requiring a loosely coupled integration mechanism. However, reducing the amount of effort needed to *build* toolchains to increase productivity, as opposed to increasing productivity by *using* a toolchain, is clearly a facet that is relevant in all three integration classes. In general, the description by Yang and Han is a bit ambiguous as to what *tool construction* is meant to refer to here: the ability of users to build toolchains in the context of highly interactive, tool-supported tasks, the ability of tools to "construct" a document in closely interleaved cooperation with users, or the ability of tool developers and domain experts to construct interoperable tools and integrated toolchains, respectively.

In summary, the three categories of Yang and Han [1996] could be considered a bit unbalanced, but they allow to make several key distinctions which could be expanded into a refined classification scheme:

1. Separating productivity improvements achieved by *automating tool coordination* through toolchain integration (Class 1), from those achieved by breaking down the processes themselves in a way that allows for *increased concurrency* for better overall tool performance (Class 2).
2. Separating productivity improvements achieved by *passive*, background toolchain support (Class 1 and Class 2), from those achieved by more *active* toolchain support that facilitates a direct collaboration between users and tools (Class 3).
3. Separating productivity improvements achieved by *using* toolchains to solve a given task more effectively (Class 1 and Class 2), from those achieved by *building* toolchains more effectively (Class 3).

**Placement and Delimitation of SENSEI in Terms of Yang and Han**

Improving productivity is one of the three major objectives of this thesis, introduced in Section 1.2. As explained, the greatest productivity gains are to be expected when only *Class 1* integration is achieved, which is possible with relatively primitive means. Relying on such simple means would, however, negatively impact the other two objectives: low-level integration makes only minimal assumptions about the tools to be integrated, requiring individual integration logic that tightly couples tools together, thereby sacrificing *flexibility*. For the same reasons, neither individual tools, nor the integration code will be *reusable*.

SENSEI will therefore aim at *Class 2* integration, focusing on control (and process) integration aspects, while leaving the data integration aspects to complementary approaches, i.e. SENSEI will be able to make control-based optimizations, e.g.

**Placement and Delimitation of SENSEI in Terms of Yang and Han (cont.)**

identifying opportunities for concurrent execution, while a chosen data integration approach might, for example, minimize data exchange redundancies.

As discussed, *Class 3* mixes two aspects: SENSEI will not address user interaction improvements, as its scope is confined to non-interactive toolchains. The second aspect, supporting tool developers and domain experts in the toolchain-building process is at the heart of SENSEI, and will be supported by providing a conceptual framework guiding their work into explicit structures.

Its conceptual nature also means that SENSEI does not dictate concrete implementation techniques, however, the lower integration categories would not suffice to achieve the aspired integration classes. With an explicit adapter concept, SENSEI will ensure that even tools offering only $C_1$ interoperability can be integrated.

### 4.3.5 Integration Infrastructure Classification According to Fuggetta

Fuggetta [1993] provides a classification not of integration or interoperability, but of its subjects, tools and toolchains. Previously, in Section 4.2, the concept of "environment" has been avoided in favor of the term "toolchain", partly because the former term is often only vaguely defined, and carries a connotation of presentation integration and interactivity.

The classification by Fuggetta introduces more clear definitions for different stages of tool collections and integrations. While the work offers quite a fine-grained taxonomy, with concrete CASE products as examples, the main distinction is *tools*, *workbenches*, and *environments*, which are also shown in Figure 4.3:

**Tools** support individual *tasks*, which is in agreement with Definition 4.1.

**Workbenches** support one or more *activities*.

**Environments** support all major activities of a particular domain.

Figure 4.3 adds **toolchain** as the generic result of an integration, and another kind not included by Fuggetta, the **automated toolchain** (as per Definition 4.2). Fuggetta focuses on interactive toolchains *supporting* its users while performing activities, but does not consider toolchains aimed at *automating* these activities.

Fuggetta's classification is based on the scope of provided support, introducing the terms *task* and *activity*. Both are performed by humans (e.g. software developers). Recall Figure 4.1 [Brown et al., 1993] in this regard, which contrasted *tasks* with *services* and *tools*. An example for an activity would be testing. Activities consist of multiple tasks that need to be performed to complete the activity, e.g. for testing this would comprise writing test cases, executing tests, and debugging. Section 3.1 also used this terminology.

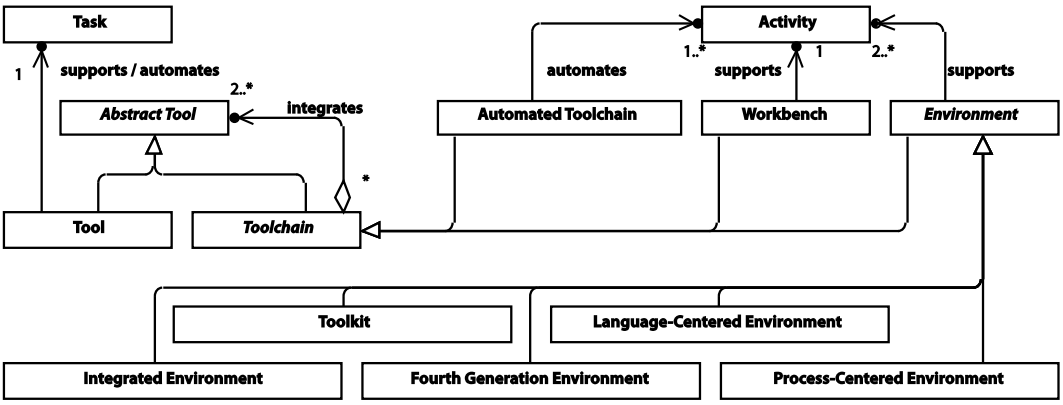



Figure 4.3: Class diagram of tool integration terms and their relationships, inspired by the integration infrastructure classification of Fuggetta [1993].

The first category, *tool*, is congruent with the definition introduced in this thesis for the term (Definition 4.1), particularly in terms of a single tool supporting only a single task. Fuggetta provides a fine-grained taxonomy of tool classes and subclasses, which will not be discussed further as it is very specific to one particular domain, namely software development tools.

*Workbenches* are characterized by Fuggetta as focused on supporting a single activity to a few activities, typically exhibiting presentation integration through coherent user interfaces, control integration in terms of the ability to directly invoke tools, and data integration through a shared data repository. There is a sub-division of this class into eight types of workbenches, which will not be discussed any further, here. The classification is according to the type of activities being supported, which is natural giving the definition, but which is, again, specific to classical software engineering phases. One category, *maintenance and reverse engineering* workbenches, deserves mentioning though, as many products reviewed in Chapter 5 fall into that category.

*Environments* are formed from individual tools and workbenches, and are characterized by a scope that covers the entirety of a particular domain – in Fuggetta's discussion, the complete software development life cycle. More generally, the environment term can be extended to fully encompassing other application domains, e.g. software evolution. This view also implies that, what is considered a workbench and what is considered an environment depends on the context: for example, a reverse engineering *environment* may be considered a *workbench* in the context of the whole field of software evolution.

Fuggetta provides five subclasses of environments, also depicted in Figure 4.3, and described in the following:

**Toolkits** refer to a loose collection of tools with only a low level of integration. With data exchanged via files export and import and mostly manual tool invocation, or partial automation through shell scripting, this corresponds to carrier level integration, and (barely) Class 1 integration effectiveness.

**Language-centered environments** are build around, and focused on, a single programming language. They provide presentation, limited levels of control, and no process integration. Data integration is based on an internal model of the considered programming language, i.e. syntactic or semantic integration level, which couples tools tightly together. The inaccessibility of integration mechanisms implies low reusability, and makes for poor overall extensibility and interoperability.

**Integrated environments** provide "standard mechanisms" for tool integration, facilitating presentation integration through common GUI libraries, data integration through extensible repositories, and control integration through means for inter-tool communication. Integrating existing tools can be achieved by wrapping, although the required effort may be high, depending on the comprehensiveness of provided integration facilities, and tool interoperability requirements. In contrast to the previous classes, integrated environments are frameworks to facilitate tool integration rather than ready-made solutions, first and foremost addressing tool and toolchain *builders*, as opposed to the environments' users.

**Fourth-generation environments** refer to tightly-integrated application development environments based on closed-off integration mechanisms, and hard-wired, proprietary development methodologies. In fact, this class seems to comprise CASE tools exhibiting most or all of the criticisms raised against them (see Section 4.1.2). In terms of integration effectiveness, these environments may reach Class 1 or Class 2, but only for applications whose needs match the environment-embedded development procedures very closely, with basically no room for customization, severely limiting flexibility and areas of use.

**Process-centered environments** refer to environments that provide means for "process-model execution" and "process-model production", i.e. process or workflow engines able to automatically coordinate tools according to explicitly provided process descriptions, and editors that facilitate the modeling of such processes. Like integrated environments, the objective is the provision of a framework for tool and toolchain builders, with a focus on process integration, and varying support for the other integration types.

**Placement and Delimitation of SENSEI in Terms of Fuggetta**

The classification scheme by Fuggetta is most readily applicable to existing tool integration solutions, and will be referred to in the review and assessment of related work in Chapter 5. SENSEI itself does not quite fit into the classification, which can nonetheless be used to highlight what sets it apart.

**Placement and Delimitation of SENSEI in Terms of Fuggetta (cont.)**

Workbenches are not necessarily meant to be extensible, and are defined by a limited scope, whereas SENSEI explicitly aims to make no restrictive assumptions to support the whole field of software evolution (compare the *Comprehensiveness* requirement, 31), or even completely different application domains. Toolkits are extensible and flexible only with a lot of manual effort, providing very little productivity improvements. Both language-centered and fourth-generation environments are limited in scope, similar to workbenches, but along different dimensions (a particular programming language and a single tool vendor, respectively).

SENSEI is aimed at toolchain-building. The tools realizing the support framework it defines may be classified as integrated environments, or more specifically, process-centered environments. Still, both environments and workbenches carry the connotation of user interaction and presentation integration, which is why Figure 4.3 contains another class that is not part of the original work of Fuggetta: *automated toolchains* are distinct, because they remove any need for interaction. They represent the result of an application of SENSEI; above all, SENSEI does *not* aim to provide integrated tool support for particular domains or specific problems itself – rather, it will support the creation and evolution of them, yielding flexible automated toolchains built from reusable parts.

## 4.4 Summary

This chapter gave a brief overview of two past tool integration research phases, the first focused on creating integrated project support environments (IPSE), which then developed into computer-aided software engineering (CASE) research. IPSE research seems to have laid many important foundations, but few results made it out of the lab and into practice. CASE contrasted this with a very practice-driven approach, but overcorrected to the point of inflexible solutions with narrow application areas and poor customizability, and pronounced vendor lock-in. The lessons that can be learned from these past research directions (summarized in Section 4.1.3) have shaped the architecture of SENSEI.

More recent tool integration research builds upon principles and practices of software system integration (component-based) and enterprise integration (service-oriented). Core ideas of software development approaches that emerged with CASE tools live on in the more general framework of model-driven software engineering, which has also found applications within tool integration. These three software engineering paradigms will provide a foundation for SENSEI, and have therefore each been given a dedicated chapter in Part III.

The bulk of this chapter, however, was dedicated to establishing a broader understanding of the field of tool integration, by discussing and defining terminology, and reviewing different classification schemes of tool integration. It was shown that tool integration is a very large, multi-faceted field that can be dissected along various dimensions and understood from different views.

The different classification schemes have been used to sharpen the boundary that defines the scope of this thesis, and to clarify the objectives: Driven in particular by the requirements of software evolution projects, with challenges such as a large body of pre-existing, diverse tools with poor interoperability, and a need for factory-like processes and their automation, a focus has been put on control and process integration. This focus also coincides with the areas that seem to have been least addressed in research and practice. In fact, some scholars of the field seem to implicitly equate tool integration with data integration. This makes it even more important to delineate SENSEI from other integration approaches with particular preconceptions about the scope of tool integration, and place it clearly within the presented classification frameworks to avoid confusion and misunderstandings.

While this chapter placed SENSEI within the different classification frameworks, Chapter 5 will use these reference frames, together with the set of requirements elicited in Chapter 3, to place related approaches for comparison, evaluation, and delimitation, and to highlight the unique contributions of this thesis.

CHAPTER 5

# Existing Approaches

This chapter takes a look at various related works and existing approaches that, to some degree, address tool integration, either in general, or within the field of software evolution. A goal is to present existing tool integration approaches, and compare them with the objectives and requirements of this thesis using the holistic view and different dimensions and classification schemes introduced in Chapter 4. This will give further examples of the very wide, multi-faceted field of tool integration, and will clearly delineate existing tool integration approaches, their objectives, scope, and application areas, from SENSEI, emphasizing its distinctiveness, and contribute to a clear-cut picture of what its aims are.

Existing research and approaches usually focus on only one, or a few tool integration aspects – as does this thesis. A problem arises if the scope and its boundaries are not made explicit: this may lead to a limited preconception about what tool integration is or should entail. For example, some practitioners that are concerned with tool integration – be it for software evolution, regular software engineering, or other projects – may have been exposed to and worked with techniques that provide data integration, only. Regardless of a discussion about whether other aspects deserve attention, as well, or which are the most relevant, ignorance of the breadth and multi-faceted nature of tool integration will inevitably lead to misunderstandings.

The approaches presented in the following sections have been organized into the following categories: *exchange file formats* (Section 5.1) address technical aspects of data integration through standardization. *Common data models* (Section 5.2) do the same on a conceptual level. *Software evolution workbenches* (Section 5.3) are pre-integrated solutions, and support only a subfield of software evolution such as reverse engineering. *Software evolution environments* (Section 5.4) are less limited in scope, and are often more extensible and feature more explicit tool integration support. *Component-based, service-oriented, and model-driven integration* (Section 5.5)

subsumes approaches that are based on either one, or a combination of multiple of the practices and principles described in Chapter 6, Chapter 7, and Chapter 8, respectively.

Building to varying degrees on concepts from these three engineering paradigms, two concrete approaches have been identified that, at first glance, seem particularly similar to SENSEI. These are *Software Analysis as a Service* (SOFAS) by Ghezzi [2012] and the *Tool Integration Language* (TIL) by Biehl [2013]. They are described in Section 5.6 and Section 5.7, respectively. Comparing certain design decisions, these sections will occasionally refer to properties of SENSEI, which has not yet been introduced in detail – it will be described comprehensively in Part IV. However, the exact details should not be necessary to understand the comparisons and distinctions being made here.

In tool integration, the process integration dimension is rarely studied or addressed [Asplund and Törngren, 2015]. Since it is a focus of this thesis, *workflow-based integration* research and approaches will be reviewed in Section 5.8, most of which aim at automating scientific analysis processes (*scientific workflow systems*).

Section 5.9 subsumes previous work that does not necessarily present concrete integration approaches, but has contributed to the field on a conceptual level. The chapter closes with a summary in Section 5.10, which also concludes Part II of this thesis. It provides a comparative overview of the different kinds of approaches that have been described (Table 5.1), and highlights the gaps that SENSEI is meant to fill.

## 5.1 Exchange File Formats

Exchange file formats (EFF) aim at improving data interoperability of tools supporting the format, i.e. of the five dimensions introduced in Section 4.3.1, they only address data integration. They usually prescribe concrete syntax rules, and abstract syntax on a meta-metamodel level (see Section 8.3), which in turn allows to specify the abstract syntax (the metamodel) of the data to be exchanged, as well as the data itself (the model). In the classification by Brown and McDermid [1992] (Section 4.3.2), exchange file formats provide at least the lexical integration level, and lay the foundations for the syntactical level. For full syntactic level integration, common standardized data models (schemas, metamodels, or ontologies) expressed in the terms of the format are needed, additionally.

The scope of exchange file formats is narrow when projected onto the requirements elicited in Chapter 3. For example, even though these formats are data-centric, they do not provide any support for specifying data flow (*Data Flow* requirement). Truly, the only aspect affected is transformer creation, which can become easier if many tools adopt the same exchange format. The *Reusability* requirement demands reusability of transformers: a standardized exchange file format allows to factor out parsers and unparsers, which can then be used as a foundation (as a program library) for all transformers. However, exchange file formats alone do not ensure data transformer reusability; particularly, they do not necessarily facilitate the separation of tools and transformers.

Examples of standard exchange file formats are:

- CDIF (*CASE Data Interchange Format*) [Parker, 1992], which is not widely used anymore, and has been superseded by
- *XMI (XML Metadata Interchange)* [2015], which is widely used in the context of UML, and model-driven architecture, and
- GXL (*Graph Exchange Language*) [Holt et al., 2006], a graph-based XML format developed specifically for the domain of software evolution.
- MSE (*Moose Interexchange Format* [sic]) [Ducasse et al., 2011] is a format with an s-expression-like syntax. It was developed for the Moose platform [Nierstrasz, 2012], which had previously used CDIF, and then XMI. Ducasse et al. believe previous attempts to unify tool-specific formats (such as the efforts that led to GXL) were unsuccessful, so they chose to develop another format.

If ontologies – in the sense of the *web ontology language* [W3C OWL Working Group, 2012] – are employed to specify common data models, RDF (*Resource Description Framework*) [Cyganiak, Wood, and Lanthaler, 2014] may be used, as is the case in OSLC [*Open Services for Lifecycle Collaboration* 2020].

## 5.2 Common Data Models

Common data models complement exchange file formats by providing definitions of concepts and their relationships of the data to be exchanged (i.e. the abstract syntax), whereas the exchange file format provides the concrete syntax and "carrier" medium (see Lethbridge, Tichelaar, and Ploedereder [2004], and compare Brown and McDermid's carrier level integration, presented in Section 4.3.2). Using an exchange file format without a common data model is possible (e.g., GXL can be used with or without referring to a schema), but only provides carrier level integration. The other way around, common data models can also be used for data exchange in the absence of common syntactic conventions, but will reduce the reusability of transformers. Both combined provide full syntactic level integration.

Ontology-based approaches [Jin and Cordy, 2005b; Würsch et al., 2012] are often described to introduce a *semantic* level. However, this depends strongly on the definition of the term "semantic". Ontologies, using *semantic web* technology, seem to adopt an understanding of semantic information that refers to a (meta-)model enriched with interrelationships (such as integrity constraints) as comprehensive as possible [Hesse and Mayr, 2008]. Hesse and Mayr go on to argue that only mental models can truly be semantic, since representing a model in any form would turn semantics back into syntax. In this regard, ontologies and metamodeling may have their advantages and disadvantages in different application areas due to the technological basis each approach usually employs, but on a fundamental level, they do not seem to differ in the level

of "semantic richness" they can achieve. For example, in MOF-based metamodeling, integrity constraints can be formalized as OCL constraints.

Brown and McDermid [1992] define their semantic integration level in operational terms, i.e. they require the explicit definition of data manipulation operations that are allowed on the data. This is different from the syntactic level, on which it is possible to test for data integrity, but not ensure that all operations preserve data integrity. Also, even if data integrity is retained, operational semantics may prohibit certain transitions from one state of the data to another.

Creating common data models for a domain as wide and diverse as software evolution is all but trivial. With respect to the requirements of Chapter 3, their coverage is minimal, providing support mostly only for the transformer creation step, and to a smaller extend for task identification due to their provision of a universal language to describe consumed or produced data. This is in contrast to a large body of work in tool integration, both for general software engineering tools, as well as for software evolution tools, which concentrate on this aspect, often exclusively. And it is true that a common data model can increase tool interoperability considerably. The objective and the approach of this thesis is different, however: it is focused on control and process integration, to provide flexibility and reusability for productivity increases in the long-term. As has been discussed before in Chapter 4 (Section 4.3.3), integration through a single data model common to *all* tools may, in fact, not be the most flexible and scalable solution, particularly if the activities to be supported are mainly non-interactive. Furthermore, such all-encompassing models may become hard to maintain, depending on their scope and resulting size and complexity, and may even foster tight coupling. In any case, common data models, and approaches centered around them, are seen as complementary to this thesis, as opposed to rivaling it.

Several approaches have tried to address the inherent complexity in creating an all-encompassing, generic data model for software evolution tools by introducing a hierarchy of granularity levels. For example, Lethbridge, Tichelaar, and Ploedereder [2004] define low-level models, which represent software systems down to concrete programming language syntax, middle-level models, which represent more general concepts common to many programming language (at least within a paradigm, such as object orientation), and high-level models, which represent the architectural level. Also, the generality of approaches is often limited (e.g. to object-oriented languages, or even a single programming language), as well as the coverage of the software evolution field in terms of supported activities (e.g. only considering reverse engineering / software analysis). There are models focussing on representing the software system under study, while others extend to software analysis subjects, such as code clones.

Examples of common data models include the following:

- DMM (*Dagstuhl Middle Metamodel*) [Lethbridge, Tichelaar, and Ploedereder, 2004] has mid-level granularity, as the name suggests. Its scope does not extend to representing, and therefore exchanging, fully detailed descriptions of programs.

It is based on concepts common to object-oriented programming languages, so its applicability for other paradigms (e.g. functional) is limited. DMM can only represent mid-level source code concepts – it does not encompass documentation, source code history, issue tracking data, or program analysis results. It is also too coarse-grained to base analyses on it that require program details down to the level of individual statements and expressions, such as code clone detection.

- FAMIX (*FAMOOS Information Exchange Model*, with FAMOOS being the *Framework-based Approach for Mastering Object-Oriented Software Evolution*, an EU-funded research project) [Ducasse et al., 2011; Tichelaar, Ducasse, and Demeyer, 2000] is a "family of metamodels" on a similar level of abstraction as DMM, used in the context of the Moose platform [Nierstrasz, 2012]. The scope is wider than DMM, as there are extensions for representing source code history, concepts of aspect orientation, duplication, and co-evolution. FAMIX is able to represent concepts common to procedural and object-oriented languages.
- SEON [Würsch et al., 2012] is a "pyramid of ontologies" for the domain of software analysis, with layers ranging from *general concepts* at the top, over *domain spanning concepts* (e.g. code clones), *domain specific concepts* (e.g. issue tracking, source code, history), down to *system specific concepts* (e.g. concrete constructs of particular programming languages, like for Java). The scope is therefore wider than DMM, for example, and it is meant to be extensible, which is a major motivation for the use of semantic web technology. Several concrete ontologies are available for download [*SEON - Software Evolution ONtologies* 2016]. SEON is part of the SOFAS approach, which will be discussed in Section 5.6.
- KDM [*Knowledge Discovery Metamodel* 2016], defined within the technical space of OMG's *Meta Object Facility* [2013], is a common data model for software modernization efforts. It is far more comprehensive and complex than DMM, for example, and also has a wider scope. Besides source code organizational aspects, middle level code representation, call and control flow graph elements, it also supports modeling databases used by a system under study, state-based representations, user interfaces, platform resources of a system's environment, both static and dynamic high-level architectural views, and build management [Pérez-Castillo, De Guzmán, and Piattini, 2011].
- OSLC (*Open Service for Lifecycle Collaboration*) provides standard specifications to represent and exchange data in forward engineering domains, such as change, configuration, project, performance, quality, architecture, application lifecycle, requirements management, and more. It is not meant to address software evolution, and in contrast to models such as DMM and FAMIX, none of their specifications is aimed at representing source code itself. They utilize semantic web technologies (RDF) and promote a minimalistic approach.

A set of integrated metamodels was also developed for the CDIF exchange file format [Electronic Industries Association, 1994], but seems to have fallen into oblivion.

OSLC goes beyond specifying common data models by also defining standard interfaces and inter-tool communication means (based on REST [Fielding and Taylor, 2002]). It therefore stands out by also providing support for adapter creation (*Uniform Interfaces* requirement). However, Leitner, Herbst, and Mathijssen [2016] come to the conclusion that, while OSLC is a solid foundation for standardized data formats and interfaces, it does not provide support regarding workflows. Furthermore, interface specifications are deemed not precise enough to be reused in different use cases. Since OSLC is geared towards general (forward) software engineering, and not software evolution, it will not be considered separately.

In the course of developing toolchains, problem-specific data models may be developed. Due to their nature, their reusability is rather limited. Examples include the Q-MIG data model (see Chapter 2), which is able to represent the basic hierarchical organization of Java and COBOL source code, along with associated measurements of various metrics, and information about the migration processes and tools used to transform COBOL into Java software systems.

As part of his thesis, Kraft [2007] created a "hierarchy of schemas", a set of models organized into five levels, spanning the low- and middle-level in terms of Lethbridge, Tichelaar, and Ploedereder [2004]. The models include a fine-grained abstract syntax graph-based schema on the lowest level, to higher-level abstractions such as call, control flow, and program dependency graphs. They only address reverse engineering and analysis of C++ source code.

## 5.3 Software Evolution Workbenches

Workbenches support only a part of a larger application domain – software *engineering* workbenches do not span the whole software lifecycle, e.g. they may support analysis and design, but not coding or testing activities [Fuggetta, 1993]. Transferred to software evolution, the supported subdomains of the most well-known workbenches are virtually always reverse engineering and program analysis. Thus, by definition, their scope does not extent to the full range of software evolution activities as demanded by the *Comprehensiveness* requirement.

For migration and reengineering, there are mostly only individual tools. The few integrated solutions are hard to characterize – their integration is typically very loose, indicating a toolkit, but their scopes are too narrow to be classified as an environment. An example of this is the COBOL-to-Java converter *CoJaC* by *pro et con* [Erdmenger and Uhlig, 2011], which technically consists of a couple of individual tools integrated into a toolchain. It hardly constitutes a workbench – in Fuggetta's terms it may be best classified as a single tool, even though it has "sub-tools". This thesis is aimed at creating a toolchain-building support framework – existing reengineering and migration toolchains represent the intended result of such a framework, except they were build

"manually". Their objective is not tool integration at all, which is why they are not considered any further in this chapter.

Since software evolution workbenches focus on reverse engineering, their tools are often interactive, to allow users to extract, browse, and visualize data, and derive knowledge and higher-level abstractions. They therefore usually follow the *integration based on integrated models* pattern (Section 4.3.3): there is no concept of process integration, and consequently they lack support for task coordination (requirements *Data Flow* and *Control Flow*). In the toolchain-building process, this is preceded by task identification. This also refers to process integration. Still, workbenches can be considered providing at least foundational support to finding appropriate techniques to apply (*Task Discovery* requirement), since they offer presentation integration instead, i.e. users can find what they are looking for, e.g. by browsing through the categories and menu entries of an integrated menu bar. However, there is no support for further describing task properties in line with the *Task Description* requirement, as this presupposes process description capacities.

Workbenches concentrate first and foremost on providing data- and presentation-integrated tools for use, as opposed to providing the means to build toolchains, separating toolchain specification and integration (*Separation of Concerns* requirement) is not among their objectives. As a result, task instantiation (particularly the *Tool Discovery* requirement is not supported by workbenches, either. Extensibility is not a central concern, and so corresponding support for tool developers is mostly lacking, as well (requirements *Tool Description*, *Tool Interoperability*, *Uniform Interfaces* and *Reusability*). Finally, due to non-existent process integration, there is also no automatic tool coordination (*Automatic Coordination* requirement).

The following are considered examples of software evolution workbenches. Not all of these examples may classify themselves as such due to different definitions and terminology. Here, they are reported on using a consistent meaning of the term workbench, as defined by Fuggetta [1993] (Section 4.3.5), and contrasted with environment.

- Rigi [Kienle and Müller, 2008] is a reverse engineering workbench with a repository-style architecture, with the repository decoupling fact extractors that fill it, from analyzers and visualizers that inspect and present it to the user. In this regard, the Rigi system can be extended by tools that output or read its exchange file format, i.e. basic interoperability means are established, but are on a rather low level, and cover only the data integration dimension. In addition, Rigi offers customization and extension means in the form of the custom *Rigi Command Language* (RCL), which can be used to script analyses.
- Bauhaus [Raza, Vogel, and Plödereder, 2006] is a reverse engineering workbench developed at the Universities of Stuttgart and Bremen, partly based on Rigi (a modified version is used as graphical user interface for several visualizations, [Koschke, 2000, p. 317]). It provides a variety of program analyses like control and data flow analysis, deadlock and race condition detection, dead code analysis, ar-

chitecture reconstruction, feature and protocol analysis, static and dynamic trace analysis, code clone detection, and metric calculation. These techniques are implemented on top of either of two program representations provided by Bauhaus, the *InterMediate Language* (IML), and *Resource Flow Graphs* (RFG). The former supports a fine-grained program representation on the level of abstract syntax trees / graphs, while the latter is a more high-level, language-independent model [Czeranski et al., 2000], like the middle metamodels described in Section 5.2. Their design has also been an influence on the Dagstuhl Middle Metamodel [Lethbridge, Tichelaar, and Ploedereder, 2004]. While Bauhaus, programmed mostly in Ada, has been implemented with extensibility in mind, its central aim is to readily support reverse engineering activities, not to serve as a platform for toolchain-building.

- Dali [Kazman and Carrière, 1999] is a workbench for architecture reconstruction, with a graphical user interface also based on Rigi. It highlights extensibility, and provides repository-style data integration facilities, which individual tools can interoperate with either programmatically, or indirectly through a data exchange file format. Being based on a relational database enables Dali to utilize plain SQL to express architectural patterns for analysis and manipulation. The Dali workbench has a narrow scope, which focusses on architectural analyses. Its means for extensibility are mostly limited to data integration.
- CIA, the *C Information Abstraction System* [Chen, Nishimoto, and Ramamoorthy, 1990], is a workbench for analyzing properties of C code, including subsystem decomposition, topologically sorted function call relations ("program layering"), dead code detection, and coupling information.
- SWAGKit [Holt, Godfrey, and Malton, 2003; *SWAG Tools* 2020] is a reverse engineering toolkit[1] focussed on architectural analysis and visualization. SWAGKit is based on a pipes-and-filters architecture and consists of several small, individual tools for parsing C/C++ programs, combining the resulting graphs, identify a hierarchical subsystem structure, and aggregate information to arrive at a coarse-grained, architecture-level representation. On top of SWAGKit, more high-level tools have been implemented, including *Beagle* [Godfrey and Zou, 2005] and *Kenyon* [Bevan et al., 2005] for software history analysis. SWAGKit is structured as a pipeline, with its tools and filters adhering to the UNIX philosophy [Pike and Kernighan, 1984], making it very flexible and extensible. At the same time, this makes integration means fairly low-level, i.e. pipelines are integrated using

[1] SWAGKit classifies itself as toolkit, and given its very loosely coupled tools, this is in conformance with Fuggetta's definitions (see Section 4.3.5). However, toolkits are, according to Fuggetta, a kind of environment, not workbenches. Workbenches and environments are distinguished by scope, while toolkits are defined by a low degree of coupling. Since these are orthogonal concerns, it seems natural that there could also be toolkits with a narrow, workbench-like scope, which is why SWAGKit is listed here, and not in Section 5.4.

filters written in scripting languages. In terms of the integration levels defined by Brown and McDermid [1992] (Section 4.3.2), this would classify as carrier level integration, the lowest of their five levels representing the degree of sophistication regarding tool integration means.

- g$^4$re [Kraft, 2007] (*GCC and Generic GXL Graphs for Reverse Engineering*) is a C++ reverse engineering workbench consisting of several, individual tools, using a pipe-and-filter architecture similar to SWAGKit. They are loosely coupled into a toolchain based on the GXL exchange file format (see Section 5.1), i.e. relying on data integration, only. The g$^4$re workbench does not seem to provide any further means to automate concrete reverse engineering processes, or integrating the individual tools required by them [Kraft, 2007, pp. 53ff]. Tools seem to be either invoked manually, or chained by ad-hoc means, e.g. UNIX pipes.

- SHriMP [Storey, Best, and Michand, 2001] (*Simple Hierarchical Multi-Perspective* tool) is a reverse engineering workbench focused on providing different views and interactive visualizations for Java program navigation and comprehension. It is implemented in Java itself, and uses JavaBeans technology to realize a component-based architecture, providing data and control integration. While this makes the workbench extensible, new functionality has to be specifically written for the SHriMP system, whereas there do not seem to be any particular means provided to integrate existing, external tools.

- GUPRO [Ebert et al., 2002] (*Generic Understanding of Programs*) is an integrated workbench for reverse engineering. It has a repository architecture and follows the *extract-abstract-view* metaphor for its data processing. Programs are extracted into a graph-based representation, graph queries are used to yield higher-level abstractions, which can be viewed as tables, or in a source code browser. While the use of the graph query language GReQL makes GUPRO a very generic workbench [Kullbach and Winter, 1999], its interoperability means are limited to file exchange based on GXL.

- Columbus [Ferenc et al., 2002] is a plugin-based (component-based) reverse engineering workbench. It provides an infrastructure for fact extraction tools, and comes with a C/C++ extractor and a corresponding linker plugin, as well as several exporter plugins. All further analysis steps are therefore external, i.e. of the extract-abstract-view process mentioned above, Columbus focusses on the extract part, only. Plugins must be written using Columbus' plugin API, and are meant to either parse (and link) source code into a common data model such as the provided C/C++ metamodel, or export data from such an internal representation to a format readable by external tools (e.g. for visualization or further analysis steps). This means that interoperability relies on providing adapters and transformers, and exchange file formats (data integration).

- ConQAT [Deißenböck et al., 2010, 2008] (*Continuous Quality Assessment Toolkit*[2]) is a software quality analysis workbench, which provides facilities metric calculation, software architecture conformance analysis, code clone detection, and visualization. It is built on top of the Eclipse rich-client platform, and therefore provides presentation integration. In addition, existing and user-definable analyses are based on a pipes-and-filters architecture, and the workbench provides the means to graphically define data-flow-driven chains of individual filters. ConQAT is itself the basis for *TeamScale* [Heinemann, Hummel, and Steidl, 2014], which aims at providing real-time software quality feedback.

Put in context of the objectives of this thesis, Workbenches could serve as target platforms for the higher levels of integration and toolchain-building support being aimed at, e.g. implementation-agnostic process definition and enactment. ConQAT is a kind of "outlier" with respect to its classification as workbench, because of its support for specifying custom data flows (*Data Flow* requirement), which is unique among all other workbenches presented. With this feature, it is also a candidate to be classified as a workflow-based integration framework (Section 5.8) In general, however, the constriction of workbenches to only a part of the whole software evolution field is in conflict with the *Comprehensiveness* requirement, making environments a potentially more suitable choice.

## 5.4 Software Evolution Environments

Fuggetta [1993] distinguishes environments from workbenches by their support of the *whole* software engineering lifecycle. In this chapter, software engineering is swapped out for the domain of software evolution. By definition, software evolution environments satisfy the *Comprehensiveness* requirement by covering the whole domain. Some products are hard to classify within this framework though, because no single solution can reasonably provide means to support every imaginable software evolution activity, at least not up front. Therefore, solutions that emphasize extensibility and the provision of an integration framework over concrete software evolution functionality are also considered environments here. This is very much in agreement with Fuggetta's classification, as there is a sub-category, *integrated frameworks*, dedicated to those kinds of solutions.

Based on these criteria, examples of software evolution environments include the following:

[2] ConQAT classifies itself as a toolkit. Using the classification scheme of Fuggetta (Section 4.3.5), it would rather fall into the workbench category. Even though, internally, ConQAT follows a flexible pipes-and-filters architecture, its filters have to be written for its specific ecosystem (as opposed to tools running directly on top of the operating system). Conversely, ConQAT provides more sophisticated means for integration than toolkits, which rely on operating system means (e.g. UNIX pipes) and low-level scripting facilities.

- Moose [Ducasse, Gîrba, and Nierstrasz, 2005] is an "agile reengineering environment" build around the FAMIX metamodel family (see Section 5.2). Its functionality is heavily focused on reverse engineering, which implies classifying it as software evolution workbench rather than environment. However, it also includes a *refactoring engine* [Ducasse, Lanza, and Tichelaar, 2000], which extends its scope from only analyzing systems, to also modifying them based on the findings. Furthermore, an emphasis is put on being extensible, and Moose provides data interoperability through its central, FAMIX-based repository, as well as import and export facilities for common exchange file formats. Its tool integration framework offers an API for tools to register and discover each other, i.e. programmatic control integration. Previous versions of Moose were implemented in VisualWorks Smalltalk [Ducasse, Gîrba, and Nierstrasz, 2005], but for the latest versions, the system was migrated to Pharo [Black et al., 2018; *Moose* 2020] (another Smalltalk dialect and environment). Tools written explicitly for the environment may also benefit from a degree of presentation integration. *Softwarenaut* [Lungu, Lanza, and Nierstrasz, 2014], an architecture recovery workbench, is built on top of (an older version of) Moose.
- Orion-RE [Alvaro et al., 2003] is a software evolution environment, mainly aimed at migrating legacy software systems towards a component-based architecture and object-oriented programming languages. For this, individual tools, as well as an XMI-based repository, are integrated using a software bus middleware based on Java remote method invocation and CORBA. This provides the basis for data and control integration. Alvaro et al. present a process model to guide software evolution projects, but also explicitly point out that the lack of a workflow engine is a limitation of Orion-RE, i.e. there is no support for process integration.
- MoDisco [Bruneliere et al., 2010b] is a software evolution environment heavily based on model-driven techniques, Object Management Group's MDA and ADM (*Architecture-Driven Modernization*) standards, and their implementation in the Eclipse ecosystem, i.e. centered around EMF (Eclipse Modeling Framework; see Section 8.7, Steinberg et al. [2008]). MoDisco is programmatically extensible through its plugin architecture. It also has a very rudimentary "workflow" component [Bruneliere et al., 2014], which allows to chain model transformations, but does not seem to provide any means to specify more complex data and control flow schemes.
- DMS Software Reengineering Toolkit [Baxter, Pidgeon, and Mehlich, 2004] is a commercial software evolution environment developed by Semantic Designs. Its main focus is on program transformation, i.e. supporting legacy system reengineering and migration, but it also program analysis and comprehension tools, e.g. for control and data flow analysis [*DMS Software Reengineering Toolkit* 2020]. A particular feature of this environment is its strong focus on enabling concurrency, using the proprietary PARLANSE language to specify partial ordering of

software evolution process steps, so they can be parallelized as much as possible for optimal run-time performance. The DMS Software Reengineering Toolkit is very generic so it can be tailored to each customer's particular requirements. Otherwise, it does not seem to be geared much towards interoperability, owed to the fact that it is a commercial product.

In terms of the requirements elicited in Chapter 3, these environments' performances are varying. In particular, the DMS Software Reengineering Toolkit is left aside in the following, as it is not meant to be used as an integration platform, at least not by third parties.

In general, none of the environments make a strong separation between toolchain specification and integration / implementation (*Separation of Concerns* requirement), partly because toolchain-building support is rudimentary, at best. Many of the other requirements depend on this, which is why these environments cannot satisfy requirements *Task Discovery*, *Task Description* and *Tool Discovery*, either. As *integrated* environments, they do provide foundational data and control integration means, but support for users specifying data and control flow (process integration / task coordination, requirements *Data Flow* and *Control Flow*) is really only present in MoDisco, which has a very simple workflow concept. As a consequence, workflow automation (*Automatic Coordination* requirement) is also only supported by MoDisco, and again, only in a very basic way. All environments build on component-based platforms, or at least offer well-structured APIs and interfaces. They therefore inherit the properties of CBSE technology, in general, with respect to

- basic means for tool specification (*Tool Description* requirement),
- the creation of interoperable tools with standardized interfaces, at least within the environment (requirements *Tool Interoperability* and *Uniform Interfaces*), and
- reusability of transformers (*Reusability* requirement), even though this concept is not usually made distinct from adapters.

Essentially, software evolution environments are therefore rated the same as component-based technology, in general (see Section 6.5). They do not constitute ready-made toolchain-building frameworks by themselves, but they may provide suitable target platforms on which to build the support framework envisioned by this thesis. Individual solutions may fare slightly differently from this generalized view. For a particular project, choosing a certain software evolution environment instead of a general-purpose component framework may be beneficial, if many of the software evolution tasks that need to be performed can be supported by tools already integrated into the environment. In the more general case that requires to integrate very diverse tools, a software evolution environment could hinder rather than help due to technical constraints imposed by it. Modern component frameworks may offer platform independence, distributed computing[3], and bindings to various programming languages and technologies.

[3]Even if a software evolution environment is platform-independent, it may still require that all tools

## 5.5 Component-based, Service-Oriented, and Model-Driven Integration

This section clusters integration approaches whose primary features are derived from component-based, service-oriented, and/or model-driven principles and techniques. There are many such approaches, showing that these software engineering paradigms have proven themselves as well-suited foundations for tool and systems integration.

The following list describes examples of such integration approaches from different domains: only some are explicitly aimed at software evolution tool integration, while others are geared towards tool integration in general, or at even more general integration of arbitrary software systems. Also, not all of these approaches are also paired with a concrete implementation in the form of an integration framework or platform, and rather describe methodologies.

- Toolbus [Bergstra and Klint, 1998; Jong and Klint, 2003] is a component-based, message-oriented middleware. It distinguishes three layers: On the bottom, *computation* is taken care of by tools (components). On the top, *coordination* is taken care of by the bus, providing process integration by running scripts in a special process description language called TScript. In between, *representation* of data exchanged between individual tools and the bus is done in a common exchange format called ATerms. Toolbus has been used to integrate the tools of the *ASF+SDF Meta-Environment* [Brand et al., 2001], another example of a software evolution environment geared mainly towards program transformation (in that regard it is similar to the DMS Software Reengineering Toolkit).
- OASIS [Jin and Cordy, 2003, 2005a,b; Jin, Cordy, and Dean, 2003], the *Ontological Adaptive Service-Sharing Integration System*, is an integration approach that has been applied in the reverse engineering domain. It uses adapters with in- and out-filters (transformers in the terminology introduced in Chapter 3) to map between the internal data formats of individual tools, using a domain ontology as common data model. OASIS also has a service concept that is separate from the tool providing it (a tool can provide multiple services – the opposite does not seem to be true, though). Using the adapters, services can be invoked on one tool, but operate on the data of another, given that this other tool supports all the concepts necessary to execute the service. This information is part of the ontology. OASIS focusses strongly on enabling data integration. In this regard, in can be considered complementary to the objectives of this thesis.
- Winter and Ebert [2005a,b] propose "*Using Metamodels in Service Interoperability*". The approach is based on component-based and service-oriented principles, and strongly separates abstract services and concrete, implementing components

run on top of it, impeding integration if different tools need to run on different operating systems, for example.

(*Separation of Concerns* requirement). However, at its core it actually builds on model-driven techniques, and aims at data integration. It uses domain-specific reference metamodels, and model transformations that map from the individual components' metamodels into such a reference model. There are many similarities to the OASIS methodology. An important difference is that Winter and Ebert propose to have one reference metamodel per service (although services that operate in the same domain may well be sharing a reference metamodel), while in OASIS, there is a single domain ontology for the whole integration solution (compare the integration patterns by Karsai, Lang, and Neema [2005] presented in Section 4.3.3).

- Bézivin et al. [2005] propose a model-driven approach for data integration using the *ATLAS Model Management Architecture* (AMMA), which is based in the EMF technical space (see Section 8.7). Similar to Winter and Ebert [2005b], a reference metamodel called the "logical pivot" is constructed. The individual tools' metamodels are mapped into this using the ATLAS transformation language (ATL). Like all approaches that focus on data integration only, there is no real overlap with the objectives of this thesis; rather, both can be used complementary. To integrate meta-tools (tools that can be parameterized with a metamodel, such as many program transformation workbenches and environments, or "metaCASE" tools like MetaEdit+; see Chapter 8), Bruneliere et al. [2010a] lift model-driven data integration to the meta-metamodel level.
- Within the *SENSORIA* project [Wirsing and Hölzl, 2011], concepts for model-driven generation of integrated software systems from higher-level descriptions like BPEL-based orchestrations [Gönczy, Hegedüs, and Varró, 2011] have been developed. Such an approach can support the automatic derivation of toolchains (*Automatic Coordination* requirement) from process-oriented descriptions (requirements *Data Flow* and *Control Flow*). SENSORIA was a very large European research project focused on service-oriented development and software systems integration methodologies and techniques, with many more outcomes than just this one. However, it was neither focused on software evolution, nor was tool integration or toolchain-building among its concerns. Its relation to this thesis is therefore not as an alternative, but as a rich source of advanced methods of service-orientation.
- MOFLON [Amelunxen et al., 2008] is a metamodeling framework compliant with OMG's Meta Object Facility. It is able to generate repositories from metamodels and triple graph grammars (TGG), a formalism that can be used for bi-directional model transformation (see Section 8.7). This has been used for data integration of tools: for each tool's metamodel, a repository is generated. Adapters have to be created manually, to synchronize between the tools' internal data and their generated repositories. Furthermore, a TGG must be constructed to establish mappings between tool pairs, for which MOFLON will generate another repository.

Because of the standardized interfaces of the generated repositories, a generic *integrator* component can then facilitate data exchange between the tools, utilizing the operational rules within the TGG-generated repository.

- ModelBus [Hein, Ritter, and Wagner, 2009] is a model-driven service bus framework for the integration of diverse, distributed software engineering tools. At its center is a model repository which allows to address models using URLs, so that individual tools need only exchange references instead of the actual data. The repository is equipped with versioning and merging facilities, to handle concurrent, distributed processes accessing and manipulating its data. Besides model-driven data integration, ModelBus builds on service-oriented techniques and standards like web services, BPMN, and BPEL, and includes frameworks implementing those standards, like an orchestration engine, to also provide control and process integration [*ModelBus* 2017]. It is a quite comprehensive middleware, and goes far beyond the objectives of this thesis in many regards. However, it does not provide the abstraction from technical concerns demanded by the *Separation of Concerns* requirement, and therefore cannot support task identification (requirements *Task Discovery* and *Task Description*) and instantiation (requirements *Tool Discovery*, *Tool Description* and *Tool Interoperability*) of the toolchain-building process as envisioned in this work (see Section 3.1). It has been applied, for example, in the field of embedded, safety-critical applications development, complemented with a common meta-model for the domain [Baumgart, 2010], and used as basis of a reference technology platform [Armengaud et al., 2011; Baumgart et al., 2012].

The approaches subsumed in this section are too diverse to generalize their overall performance in terms of the requirements for a toolchain-building support framework. Most of them have objectives that are quite different from what this thesis is trying to accomplish, making comparisons even more difficult. The reason these approaches have been included, in many cases, is their support for individual requirements. Others provide practical evidence of the expedience of particular design decisions or technology choices. Most approaches actually focus on data integration, only. Since they are often classified simply as integration or interoperability approaches, it is important to notice that these approaches are not alternatives, but are actually orthogonal to the objectives of this thesis.

In general, all the integration frameworks presented in the last sections are *pre-integrated* solutions, rather than approaches that aid the integration process itself. In this regard, they do not address the central objective of this thesis, at all. Their number does further emphasize the general need for integration, though. Being pre-integrated, workbenches and environments cannot be tailored to fully support and automate specific processes, and are therefore always only applicable to a limited number of problems. Furthermore, if the processes being supported are evolving, these approaches lack the *flexibility* to be adapted, accordingly.

There are two service-oriented approaches, SOFAS [Ghezzi, 2012] and TIL [Biehl, 2013], which have objectives similar to those of this thesis. Therefore, they are discussed in more detail in Section 5.6 and Section 5.7, respectively.

## 5.6 SOFAS: Software Analysis as a Service

*Software Analysis as a Service* (SOFAS) is an approach developed by Giacomo Ghezzi, described in his dissertation [Ghezzi, 2012] (a *cumulative* dissertation, i.e. a collection of publications by Ghezzi, framed by an introduction and a conclusion). The first paper on the topic was published in 2008, and the thesis was completed in 2012. The objectives are described as follows:

> “ [SOFAS is aimed at providing a] distributed and collaborative software analysis platform to enable seamless interoperability of software analysis tools across platform, geographical and organizational boundaries.
>
> [Ghezzi, 2012, p. 2] ”

Therefore, there is some overlap with this thesis in both application area and objectives: SOFAS targets software analysis, i.e. the reverse engineering phase of software evolution. This thesis is aimed at covering the whole field of software evolution, which extends beyond reverse engineering, software analysis and visualization, to software restructuring, refactoring, and transformation, to also be able to directly support reengineering, migration, and continuous maintenance and development.

Both approaches are aimed at tool integration. SOFAS provides this for a sub-field of software evolution. In terms of data integration, it goes beyond the objectives of this thesis, by also providing an ontology for data standardization. In contrast, this thesis deliberately excludes this integration aspect to be filled by a complementing approach or technology – in fact, SOFAS' software evolution ontology “SEON” could fill this gap. Others have been mentioned, e.g. in Section 5.2 and Section 5.4. This thesis strives to provide holistic support for the toolchain-building process, putting an emphasis on reusability and flexibility. A major cornerstone in achieving this is the a clear separation of task and process specification concepts on the one hand, and their technical imlementation on the other (*Separation of Concerns* requirement). This is viewed as a distinguishing quality not present in SOFAS.

Another commonality is that both approaches are service-oriented; Chapter 9 will describe in detail how service-oriented principles are applied in the approach proposed by this thesis, SENSEI. SOFAS provides *SCoLa* (“SOFAS composition language”), a workflow language based on BPEL, which is able to reference SOFAS' RESTful web services, and be executed by an interpreter. Several design decisions were made in favor of “simplicity and ease of use over expressiveness and feature richness” Ghezzi and Gall, 2013 (e.g. simplifying BPEL, and opting for WADL instead of WSDL for service description). SENSEI's service orchestration language is completely technology-agnostic, which

keeps it simple, as well, but also provides maximum flexibility. SENSEI orchestrations can be either interpreted, or used as input for a code generator. The former approach is realized by *SNOrcInS* ("SENSEI Orchestration Interpreter Service") [Küpker, 2015]. The latter approach, described in detail in Chapter 14, is implemented by SCAffolder, a model-driven code generator targeting *Service Component Architecture (SCA)* [2015], which is what the name alludes to. Both SOFAS and SENSEI offer basic control and data flow support, and a graphical orchestration designer.

One of SOFAS focuses is supporting empirical research, i.e. to reduce the effort required to setup the tooling for experiments (particularly in the area of *mining software repositories*), and make them easier to repeat for verification. SENSEI can be used to this end, as well. It was conceived to support activities comprising the whole field of software evolution, targeting industrial migration and reengineering projects, and research projects with industry relevance and participation. The "distributed" and "collaborative" aspects, as well as the ability to span "geographical" and "organizational" boundaries, are present in both aspects, too (SENSEI's abilities in this regard are exemplified in the Q-MIG case study, described in Chapter 15).

### 5.6.1 Comparison

SOFAS fully supports task identification through its searchable service catalog (*Task Discovery* requirement), task coordination (requirements *Data Flow* and *Control Flow*), uniform interfaces (*Uniform Interfaces* requirement), and the automatic execution of defined workflows (*Automatic Coordination* requirement). Certain technical details are hidden from normal users, by offering a simplified view of services through the user interface. There is also the ability to specify "abstract services". But in general, services are described in a technical manner, i.e. the separation of specification and integration is not as clear and rigorous as desired in this thesis (*Separation of Concerns* requirement). The application domain of SOFAS is limited to software analysis (*Comprehensiveness* requirement). Due to the tight coupling of toolchain specification and integration, there is little need for specifying required service properties (*Task Description* requirement), since users implicitly select an implementation when they select a concrete service. For the same reasons, automatic task instantiation (requirements *Tool Discovery* and *Tool Description*) is not supported, either, and there is no mention of any specific means to assist the implementation of new tools, or the inclusion of existing ones (*Tool Interoperability* requirement). SOFAS uses an ontology as common data model, and all its analysis services have been specifically built to support it. Due to this, SOFAS has no concept of data transformers (*Reusability* requirement).

The following lists a number of additional properties of both SOFAS, and SENSEI, the approach proposed in this thesis. SENSEI is described in detail in the chapters of Part IV. However, detailed knowledge of SENSEI should not be necessary to follow and comprehend these statements.

- For SOFAS, RESTful web service technology has been chosen, as opposed to SOAP-based web services, to provide simpler, standardized service interfaces. SENSEI similarly strives for simple and uniform interfaces, but goes one step further by completely decoupling conceptual services from their technical implementations. SENSEI services are similar to the view of SOFAS software analysis services that is exposed to users. However, how they are implemented, including which technology is used to describe their *technical* interfaces, is interchangeable in SENSEI, while it is fixed in SOFAS (using WADL [Hadley, 2009]).
- In both SOFAS and SENSEI, services are atomic insofar as they do not possess operations. SOFAS makes some additional assumptions about all its software analysis services, namely that each offers REST resources, on which certain HTTP methods must be available. These have a general semantic that is the same for all software analysis services. SENSEI makes no such assumptions, as it needs to be applicable to software evolution techniques of any kind, not just analyses.
- All data used, exchanged, and produced by SOFAS software analysis services must conform to the SEON ontology. SENSEI does not provide an ontology, or other form of a common data model. Service parameters are typed, but these data structures are merely names describing the kind of data, but come with no restriction regarding its structure, technical representation, or transmission protocol. In SOFAS, service input data must be either strings or files; SENSEI makes no such restrictions. A particular stipulation regarding the signature of all services in SOFAS is that they must be queryable using SPARQL.
- In SOFAS, data produced by one service and consumed by another must conform to the same SEON types. This allows for workflow validation, using custom annotations in the WADL-based service descriptions. The SOFAS orchestration editor can furthermore suggest services from its catalog which fit to a service that is already part of an orchestration. A similar feature is possible in SENSEI due to the conceptual data structures defined in its service catalog. How the data is technically represented is specific to concrete service implementations in SENSEI, and these representations do not necessarily have to match up. If two services are supposed to exchange data, and no pair of corresponding implementations can be found that are compatible in this regard, SENSEI's composition finder will try to automatically insert one or more data transformers instead.
- SOFAS invokes analysis services asynchronously, and then uses polling to check whether an analysis has finished execution; these technical details are taken care of automatically, and are hidden from normal users. In SENSEI, synchronous service invocation is assumed, as well. Concrete implementations are free to use actual synchronous calls, or simulate a synchronous behavior using asynchronous calls, and polling or callbacks to retrieve results. Just as in SOFAS, SENSEI users are not usually exposed to such implementation details.

- Both SOFAS and SENSEI have a service catalog. Since the former makes no strong separation between services and their implementations, its service catalog also takes on a role similar to SENSEI's separate component registry. The service catalog in SOFAS is structured by a taxonomy, while that of SENSEI is not. However, a method to discover, describe, and classify software evolution services to fill the SENSEI catalog and derive an adequate taxonomy is sketched by Jelschen [2013].
- In SOFAS, all services must either be data gatherers, basic software evolution analyses, or composite software evolution analyses. SENSEI, being more general, makes no such assumptions about its services.
- The orchestration languages of both approaches are similar in expressiveness; both feature structured control flow elements, and both do not have exception handling. The language of SOFAS is called SCoLa, and has control *links*, which allow to specify a partial order on services. Together with the *flow* construct for parallel execution, this enables slightly more flexible synchronization than in SENSEI, which only has the structured control flow concept for concurrency. SCoLa is based directly on a simplified version of BPEL, while SENSEI is, once again, technology-agnostic in this regard. A unique feature of SENSEI are its capabilities, a simple means to declaratively specify required service properties, which can be utilized to express complex processes in a very clear and concise manner.
- SCoLa allows to define *abstract* analysis services and workflows, to use as templates. In SENSEI, all services can be considered abstract; the mapping to concrete implementations is done automatically. SENSEI provides means to declaratively specify required capabilities of individual services within an orchestration, to influence this mapping, and have it select components that meet the requirements.
- SOFAS features a set of service management tools that take care of cross-cutting concerns like logging and monitoring during workflow execution. Due to SENSEI's complete decoupling of service orchestrations from their technical realization, such features are easy to realize within the approach, too. SCAffolder, for example, automatically generates logging statements into the code it produces. A possible extension would be to enable users to declaratively request certain properties using SENSEI's existing capability language (see Section 18.3.1).
- SOFAS is meant to be extended with new services mainly by the research group in which it was conceived, and remains to be maintained. Users can use the existing services to design software analysis workflows, but are not expected to integrate new services themselves. This is in stark contrast to SENSEI, which is explicitly aimed at facilitating the integration of arbitrary tools, existing or newly developed. Therefore, SENSEI does not include a predefined set of services or components, although several have been modeled and implemented, respectively, to apply the SENSEI approach in different application domains.

### 5.6.2 Summary

SOFAS and SENSEI are both service- and process-oriented approaches for tool integration in the field of software evolution, and share many similar features. The scope of SOFAS is narrower, focussing on software analysis. Its concepts are also fused more strongly to certain technologies, such as WADL-based, RESTful web services. SENSEI has a broader scope and is therefore more generic, and potentially more flexible. For example, SENSEI can be mapped to many different target platforms and middleware technologies, to support the integration of arbitrary software evolution tools, and to be tailored towards different requirements, like runtime performance, or distributed execution. SENSEI orchestrations can either be interpreted, or used to generate executable toolchain code; implementations exists for both approaches.

This increased flexibility might entail a higher initial adoption overhead. For pure software analysis projects, SOFAS may be the more pragmatic toolchain-building framework choice, particularly if most or all the required services are already available. Projects that span outside of the confines of software analysis can only be fully supported by SENSEI.

The separation of abstract services from implementing components is unique to SENSEI, and enables features like automatic matching of services to components. This is further supported by service capabilities, a powerful tool for declarative specification of required functionality, facilitating concise expression of otherwise complex software evolution processes.

SOFAS is complemented by a common data model in the form of the SEON ontology. SENSEI does not have a comparable feature, relying instead on transformers to translate between tool-specific data formats. Again, this makes SENSEI potentially more flexible, but when starting out from scratch, the overhead of creating the required transformers may be greater, while in the long-term, their reusability will pay off. It is also possible to combine SENSEI with SEON, i.e. use the ontology for the data integration of software analysis tools, and use direct transformers for everything else. One could also opt to combine the complete SOFAS framework with SENSEI, e.g. by making SOFAS analyses available as SENSEI services.

## 5.7 TIL: Tool Integration Language

In his dissertation, Biehl [2013] describes TIL, the *Tool Integration Language*, and the toolchain-building methodology and support framework build around it. The work is structured as a short monograph, with a set of nine published papers appended. For many of the details, the monographic part refers to one of these papers. The papers appeared between the years of 2010 and 2012.

TIL aims at classic forward engineering tool integration, and to the field of embedded systems development, in particular. Due to this, it has been recognized as a

related work only after its completion, as papers on TIL have been published mainly in outlets of the embedded systems community, while SENSEI publications have mostly targeted the software evolution community. Similarities between the two approaches have therefore been developed independently. As will be described in the following, the parallels are mostly found with respect to technical issues and design decisions, while there are considerable differences with regard to each approaches' objectives, and accordingly, their respective conceptual frameworks.

TIL is a language for designing high-level models of toolchains. A graphical editor, the TIL workbench, was implemented to support this activity. Based on TIL models, integrated toolchains are derived through model-driven code generation, targeting a *Service Component Architecture* framework (FraSCAti) as its component model and runtime. So far, these are all aspects that are very similar to SENSEI. Going through the requirements of this thesis will uncover the differences.

### 5.7.1 Comparison

TIL provides a reasonable level of abstraction from implementation details, but there is a one-to-one relation between actual tools, and their representation in TIL, i.e. the separation is not as strong as demanded by the *Separation of Concerns* requirement. As mentioned before, TIL is not targeted at software evolution tool integration (*Comprehensiveness* requirement), and its reliance on standards like OSLC would make its application in this domain quite laborious, at best.

There does not seem to be support for task identification (requirements *Task Discovery* and *Task Description*). The service discovery process described by Biehl, Gu, and Loiret [2012] instead refers to the extraction of TIL tool adapter metamodels and interfaces from an existing OSLC-based specification.

TIL supports data flow specification (*Data Flow* requirement) using the *DataChannel* and *TraceChannel* concepts. Support for control flow specification (*Control Flow* requirement) is weaker, or at least less obvious, because TIL is aimed at automating more interactive use cases, which is why the language is inherently reactive (event-based). Its *ControlChannel* concept is more akin to transitions in finite state machines (and it can have guard conditions). The *Sequencer* allows to model daisy-chained invocations of tools and data or trace channels, but there are not further control flow structures, e.g. no loops.

Due to the lack of a sufficiently clear separation between specification and integration, there is no support for task instantiation in terms of matching tasks to tools (*Tool Discovery* requirement), and specifying provided properties of tools (*Tool Description* requirement). In fact, TIL models are much more technical than SENSEI orchestrations, and do not actually describe processes. TIL's most elementary language concept is that of a *ToolAdapter*, which has no direct correspondence to a task or step in a process.

However, support for aiding developers in creating ToolAdapter implementations

(*Tool Interoperability* requirement) is quite strong, offering stub generation like SENSEI does. TIL goes beyond that, though, and can even provide full adapter code generation for certain cases [Biehl, Gu, and Loiret, 2012], as well as facilities to automatically derive prototypical data transformations [Biehl, Hong, and Loiret, 2012]. The uniformity of adapter interfaces (*Uniform Interfaces* requirement) is ensured through this code generation approach, as well, and by having them conform to SCA and OSLC standards. The infrastructure of TIL does not seem to provide any particular means for the reusability of once-produced transformers (*Reusability* requirement); being able to partially generate them makes this less of a concern in TIL.

Finally, automatic tool coordination (*Automatic Coordination* requirement) is fully supported by TIL, and its approach to achieving this is probably the greatest similarity to SENSEI, as both rely on model-driven transformations and code generation for an SCA framework for this. Still, there is also a fundamental difference here, rooted in the distinct nature of TIL and SENSEI's orchestration language: TIL assumes a scenario in which embedded systems developers use different interactive design tools. For example, a developer may work in one tool for a while, then save his changes. This may trigger some action automated by a TIL toolchain, e.g. transforming and transmitting the modified data to another tool. In contrast, SENSEI is aimed at fully automating the processes that have been modeled as service orchestrations, without user interaction.

The following list enumerates several further similarities and distinctions between the two approaches:

- Biehl [2013, p. 6] distinguishes four stakeholder roles: users of tools and users of toolchains, as well as creators of tools and creators of toolchains. SENSEI also distinguishes several roles: domain experts use tools and toolchains, but are also the ones modeling processes in need of tool support. Executable toolchains are derived from that description, i.e. toolchain creation is automated. Tool creation is the responsibility of tool developers. In addition, SENSEI introduces catalog maintainers as curators of service catalogs, responsible for creating standardized, abstract description of software evolution tasks in the form of services.
- Both TIL and SENSEI are model-driven, and its central concepts are described in a metamodel. As suggested by the name, TIL focuses on the tool integration language, which corresponds to SENSEI's service orchestrations. In addition, the SENSEI metamodel features the service catalog, component registry, and capabilities layers. TIL does not have comparable concepts.
- The design of the tool integration language, and SENSEI's orchestration language differs considerably. With TIL, tools ("ToolAdapters") are integrated directly, while SENSEI uses services, which are mapped to implementing components in a separate (automated) step. TIL is more technical, while SENSEI's service orchestrations serve as an abstraction layer, hiding technical detail. TIL's execution semantics are event-based, and have been mapped to finite state machines [Biehl, 2012], while SENSEI's orchestration language is process-oriented like a flowchart

(but with structured control flow akin to Nassi-Shneiderman diagrams [Nassi and Shneiderman, 1973]). TIL has features for user interactivity, which are not currently present in SENSEI. Conversely, SENSEI has a more fully-featured set of control flow structures, including loops, conditional branching, and concurrency, whereas TIL only has the ability to chain multiple steps in a sequence.

- Both approaches feature an editor to model processes, the TIL Workbench, and the SENSEI editor (see Chapter 13), respectively. Both are based on the Eclipse Rich Client Platform. In SENSEI, this is actually a set of three integrated editors, to model not only service orchestrations, but also the services themselves, contained in a service catalog, and components, contained in a component registry.
- TIL models are independent of any particular implementation technology, just as SENSEI models are. Both approaches have a code generator (SENSEI's is called SCAffolder, and is described in Chapter 14), and for both the Service Component Architecture (SCA) was chosen as concrete target platform. TIL also generates artifacts for OSLC compliance [*Open Services for Lifecycle Collaboration* 2020], while SENSEI's SCAffolder does not support it. As an alternative to being used for code generation, SENSEI models can also be interpreted [Küpker, 2015].
- TIL's tool adapters are split into two parts: the external part represents the interface needed to access the integrated tool in a uniform way, and can be generated automatically. The internal part represents the adapter logic that maps between this interoperability interface and the tool's own interface, and has to be implemented manually. This is very similar to SENSEI: SCAffolder also implements a stub generator, which produces the technical interface descriptions, SCA artifacts, and other code stubs, required for interoperability, from a SENSEI model containing the service descriptions the given tool is implementing. The rest of the adapter logic has to be created manually. For a few kinds of interface technologies, TIL includes templates that allow it to generate both parts of the adapter, completely.
- In TIL, tool adapters need to be associated with a metamodel of the data used by the adapted tool. For OSLC-compliant tools, the metamodel may be discovered, automatically [Biehl, Gu, and Loiret, 2012]. Data channels connecting tools similarly need to be associated with a transformation, supplied in one of several supported model transformation languages. In certain cases, a heuristic approach can be used to automatically generate a prototypical model transformation based on the source and target metamodels [Biehl, Hong, and Loiret, 2012], reducing manual effort. SENSEI's abstraction level in this regard is higher: on the service and orchestration levels, no assumptions are made about data structuring or technical representations. SENSEI's tooling is able to insert transformers automatically, if necessary, during toolchain generation. Transformers have to be implemented manually, but no assumptions or stipulations are being made regarding their implementation technology, offering greater flexibility.

### 5.7.2 Summary

TIL and SENSEI are toolchain-building frameworks with several technical similarities, such as a high-level language for toolchain or process specification, used as the basis to automatically generate an executable, fully integrated solution. Both use model-driven techniques to achieve this, and both feature an implementation that targets SCA as component framework. On closer inspection, however, major differences regarding objectives, target domain, and corresponding design decisions become apparent. The most obvious difference is TIL's focus on integrating forward engineering tools, particularly in the field of embedded and safety-critical systems. Its dependence on OSLC may constitute an impediment to its application for building software evolution toolchains. Conversely, SENSEI has been kept very generic, and shown to be applicable to domains other than software evolution (see Chapter 16).

Comparing the tool integration language with SENSEI's orchestration language reveals further differences, as both have been designed for markedly disparate use cases. TIL is event-based, and has a dedicated concept to represent toolchain users in its models, who can trigger, e.g. tool executions and data transformation and transmission, as well as receive notifications. The approach is therefore more appropriate for usage scenarios with user interaction, and the integration of interactive tools, e.g. requirements management and software design (UML) tools. Ongoing research addresses the extension of SENSEI to support the modeling of interactive systems, which is briefly described in Section 16.4.2, but otherwise, this is beyond the scope of this thesis.

Conversely, SENSEI is meant mainly for the full automation of (software evolution) processes. Therefore, it features a process-oriented orchestration language with a full set of control flow structures, such as loops, conditional branches, and concurrency, neither of which are present in TIL.

Furthermore, TIL seems to be aimed at integrating rather large tools or tool suites used in software development. SENSEI services will usually be of finer granularity, although in the end, this is up to catalog maintainers designing the services. The separation between abstract services and implementing components is much stronger in SENSEI: in TIL, there is a one-to-one relation between the tool adapters used in its models and concrete tools to be integrated. In SENSEI, this is fully decoupled, i.e. a component (wrapping around a tool) may implement arbitrarily many services, and a service may be implemented by arbitrarily many components.

With TIL, several approaches have been developed that are beyond the scope of this thesis, e.g. the heuristic-based generation of transformations for data integration. Conversely, the strong separation of services and components, the capability concept for declaratively specifying required service properties, and the ability to automatically match such service instances to appropriate components, are examples of features that are unique to SENSEI.

## 5.8 Workflow-based Integration

Most of the integration approaches discussed so far address data integration. Software evolution workbenches and environments may also offer control, platform, and presentation integration to varying degrees. The component-based, service-oriented, and model-driven approaches usually address either data or control integration. However, *process integration* is offered by only a few solutions, and in most cases the support is rudimentary. Among the workbenches, ConQAT provides means to specify data-driven chains of individual processors. With its concept of orchestration, service orientation offers techniques and frameworks for process automation, with ModelBus being an example of a middleware incorporating an orchestration engine for this purpose.

The fields of workflow management and (business) process automation (see also Section 7.5) fit into this gap, although they are not necessarily concerned with integration. Workflow management systems may, for example, only serve to automate the communication, notification, and exchange of information between different participants of a process, while the actual tasks are still performed manually by (human) actors. This allows to monitor progress, and ensure adherence to processes established in a business for (e.g. quality assurance). Processes may also be partly or fully automated, if tasks can be executed by appropriate software applications. Only at this point it becomes an automation, and a software *integration* matter.

A particular variant of frameworks for process modeling and automation are *scientific workflow systems*, aimed at automating the steps of data- and computationally intensive, scientific analysis processes, such as in bioinformatics (genomics), computational chemistry, or data mining in general. Because many of these applications require immense computational power, or the processing of very large data sets, scientific workflow systems also provide facilities for executing processes massively parallel, and may be tailored towards grid computing or computer cluster environments.

Since techniques and frameworks in other domains (business process automation, service orchestration) have already been covered in this chapter, and before that in Section 7.5, this section focusses mainly, but not exclusively, on scientific workflow system. Not all workflow systems that are mainly advertised as “scientific” are necessarily restricted to those domains. The following list provides some examples:

- Apache Taverna [Hull et al., 2006; Wolstencroft et al., 2013] allows to model and execute workflows of REST- and SOAP-based web services, as well as local scripts, and mainly caters to the needs of bioinformatics. Workflows are modeled in terms of data flow, only (*Data Flow* requirement); control flow cannot be specified (*Control Flow* requirement), and instead follows the data flow, implicitly. Execution can occur locally, distributed over a network, or within grid and cloud computing environments. Wolstencroft et al. [2013] note the need to discover suitable (web) services to support tasks, and potentially replace one (unreliable or otherwise inadequate) implementation for another. For this, they point to on-

line registers like the *BioCatalogue* [Bhagat et al., 2010] of web services for life sciences. However, finding, replacing, and integrating services will still require a lot of manual effort, since service interfaces do not seem to be standardized. That means, two web services may *conceptually* offer the same functionality, but their service descriptions can differ, as they are not separated from their technical interface descriptions (*Separation of Concerns* requirement).

- TraceLab [Keenan et al., 2012] is intended to support the design and execution of processes modeling requirements traceability experiments. It is based on Microsoft .Net technology, which makes it dependent on the Windows platform. Also, components for the platform have to be written in a .Net-compatible language. Even though Keenan et al. state that "component[s] can be written using almost any memory-managed language", their meaning becomes more evident later, when it is explained that Java components must utilize the Java / .Net bridge IKVM [Frijters, 2014] to be used with TraceLab. Control flow is visually specified in the form of a precedence graph, meaning the execution of multiple components can, by default, occur concurrently, if all their predecessors have completed execution. Data exchange uses a blackboard architecture [Buschmann et al., 1996, p. 71]: components in a workflow are configured to write to, and read from, a central, shared "workspace".
- Bio-jETI [Margaria, Kubczak, and Steffen, 2008] is a service-oriented integration and orchestration platform for bioinformatics. It is based on jABC [Lamprecht, Margaria, and Steffen, 2014; Lamprecht, Steffen, and Margaria, 2016; Margaria et al., 2006], which provides the service modeling and orchestration capabilities, and jETI [Margaria, Nagel, and Steffen, 2005], a tool integration framework that helps build wrappers around file-based command line tools, fits them with uniform interfaces (*Uniform Interfaces* requirement), and makes them available for remote invocation. Bio-jETI features foundations for separating specification and implementation in the form of the SIB concept (*service independent building block*)[4]. Yet, this does not seem to be used to specify and automatically match specific task needs to adequate tool provisions (requirements *Task Description*, *Tool Discovery* and *Tool Description*).It further provides *constraint-guarded and constraint-driven workflow design* [Lamprecht, 2013, pp. 40ff]. Using model-checking techniques, it allows to synthesize complete and correct workflows to be (semi-)automatically synthesized from "loose" (incomplete) workflow specifications (e.g. with necessary intermediate steps missing). Such techniques may also be applicable to automatically match tasks to tools (*Tool Discovery* requirement). The SIB browser and the TaxonomyEditor plugin help find appropriate blocks for tasks (*Task Discovery* requirement), and the SIBCreator provides support for integrating tools into the framework (*Tool Interoperability* requirement).

[4]This terminology stems from intelligent telecommunication networks, where jABC originates from [Lamprecht, Steffen, and Margaria, 2016; Margaria et al., 2006; Margaria, Kubczak, and Steffen, 2008].

In general, workflow systems mainly support task coordination (requirements *Data Flow* and *Control Flow*) automatic execution (*Automatic Coordination* requirement). Most scientific workflow systems are actually limited to expressing data flow, and so support the *Data Flow* requirement, but not the *Control Flow* requirement. This more simplistic model of processes, in which control flow implicitly follows data flow, has lower expressive power, i.e. certain forms of (more complex) processes cannot be realized within the corresponding frameworks, at all [Lamprecht, 2013, pp. 177ff].

Support for all other requirements varies, but mostly these tools are simply neither tailored towards tool integration, nor software evolution. An exception is Bio-jETI, which has a comprehensive feature list that satisfies many requirements of this thesis, but is still geared towards slightly different objectives, and a different application domain. Lamprecht, Steffen, and Margaria [2016] identify two shortcomings themselves, namely insufficient user assistance in finding the right SIBs (*Task Discovery* requirement), and an unintuitive approach to data flow specification (*Data Flow* requirement).

## 5.9 Conceptual Works

This section describes previous works that have made contributions on a more conceptual level than the concrete integration approaches presented so far. Still, they are aimed at providing integration solutions, rather than merely establishing classification schemes, which is why they appear here rather than in Chapter 4. The *Software Bookshelf* [Finnigan et al., 1997], described in Section 5.9.1, is a seminal work describing a reference architecture for integrating software evolution tools into workbenches. Section 5.9.2 provides a short description of the *Reference Model for Frameworks of Software Engineering Environments* [Martin, 1993], and is acknowledged here mainly for its early use of a high-level service concept that is also a core aspect of SENSEI. Section 5.9.3 summarizes toolchain-building requirements and lessons learned presented by Kienle [2006], and compares them with the objectives, requirements, and general direction taken by this thesis.

### 5.9.1 Software Bookshelf

The *Software Bookshelf* [Finnigan et al., 1997] can be described as a conceptual and high-level architectural reference framework for building a reverse engineering workbench. A bookshelf is used as a metaphor for a well-organized repository and single point of truth for information on a (legacy) software system. Finnigan et al. introduce three main stakeholders they name *builder*, *librarian*, and *patron* to analyze requirements for a software bookshelf. The builder creates the infrastructure and the actual tools that make up the bookshelf. The librarian fills the bookshelf with information by using the integrated reverse engineering tools. The patron uses the bookshelf to browse information.

The reference architecture identifies three basic building blocks, namely a *repository*, *tools* such as parsers and other reverse engineering and software analysis tools, and a *user interface*. All three are connected via network. A prototype build was based on web technology (of that time, i.e. the mid-nineties). Quite remarkably, the description of how the HTTP protocol and its request methods (GET, POST, etc.) is reminiscent of what would nowadays be called a REST architecture [Fielding and Taylor, 2002]. Other technology choices did not age as well, e.g. using CGI (*Common Gateway Interface*) to invoke tools. CGI is rarely used today due to its limited scalability. However, while Finnigan et al. do acknowledge scalability issues themselves, a software bookshelf workbench would not have the performance requirements many modern web applications have (e.g. the number of simultaneous users to be expected is clearly limited).

Although Finnigan et al. mention reengineering and migration, such projects are only supported insofar as patrons can use the bookshelf to gain knowledge about the original system; actually performing reengineering and migration activities are out of the scope of the bookshelf. In this regard, the bookshelf is best classified as a workbench (Section 4.3.5). The *tools* building block of the bookshelf can also be considered separately, if they are integrated by loosely-coupled means. In the terms of Fuggetta [1993], the result could be considered a toolkit (though still with the scope of a workbench, as only reverse engineering, not all software evolution activities, are covered).

This thesis has introduced the roles of tool developers and domain experts in Chapter 3, and will further expand on that in Chapter 9. These roles do not quite align: tool developers build the tools, but not the infrastructure, like the bookshelf's builders. Modeling processes is not really covered by the bookshelf; builders are responsible to integrate tools, while this thesis aims at supporting domain experts to model their processes, which are then turned into appropriate, integrated toolchains automatically (*Automatic Coordination* requirement). The distinction between librarians and patrons is not made in this thesis – both roles' responsibilities fall to domain experts.

The Software Bookshelf is implemented by the *Portable Bookshelf* (PBS). This system seems to have become obsolete by now, though; according to Kraft [2007, p. 22], it has evolved into *SWAGKit* (see Section 5.3).

### 5.9.2 Reference Model for Frameworks of Software Engineering Environments

The *Reference Model for Frameworks of Software Engineering Environments* [Martin, 1993] defines and organizes a comprehensive list of commonly required or useful framework *services*. A framework, in this context, is defined as the infrastructure part of a software engineering environment, i.e. interoperability means as opposed to the tools built upon them, which provide the central functionalities of environments to support concrete software engineering activities.

The reference model provides a very detailed scheme for comparison of environments, but it does not put forward a tool integration approach itself. In fact, it could have been used to classify and compare the existing approaches described in this chapter, but this was considered overkill, due to the reference model's comprehensiveness, and the fact that it is aimed at software engineering in general, not at software evolution environments.

The reference model is described as a *conceptual framework*. Remarkably, its use of the term "service" is, in many regards, closer to the service concept used in this thesis than most of the definitions put forward in SOA literature a decade later (see Chapter 7). The distinction between conceptual services and their realizations in concrete tool is made even clearer in the related *Reference Model for Project Support Environments* [Brown et al., 1993], which is briefly discussed in Section 4.1.1.

### 5.9.3 Component-based Tool-Building Lessons

In his dissertation, Kienle [2006] performed an extensive literature study on reverse engineering, to elicit tool-building requirements from it. Using these requirements, Kienle evaluated five component-based tools. Finally, a set of ten lessons learned were distilled from the case studies (a summary is provided by Kienle [2007]). The overall study differs from the requirements described in Chapter 3 in terms of aim, scope, and extent:

- Kienle's study looked at building individual tools, while this thesis is aimed at tool*chain*-building.
- Kienle focuses on reverse engineering, while this thesis considers the whole domain of software evolution.
- Kienle's requirements analysis is much more extensive, as its results were a major deliverable of his work, while this thesis aims at creating a toolchain-building support framework, which is why requirements elicitation is less broad and more focused towards this specific objective.

Still, there is significant overlap to warrant a comparison of his findings with the assumptions, requirements, and conclusions drawn so far in this thesis. The five most important requirements for reverse engineering tools according to Kienle [2006] are *scalability*, *interoperability*, *customizability*, *usability*, and *adoptability*.

#### Tool- and Toolchain-Building Requirements

Scalability is a concern because of the large size of many legacy systems. Performance optimization on the toolchain level can occur in terms of data integration (e.g. incremental updates) and control integration (e.g. concurrency). While this thesis is not aimed at providing ready-made solutions to address scalability concerns, the approach

has to make provisions that ensure it can be combined with techniques appropriate for project-specific scalability needs.

Interoperability is a requirement that is perfectly in line with the requirements elicited in Chapter 3, particularly the *Tool Interoperability* requirement.

Customizability demands tools be adaptable to different usage scenarios. This thesis recognizes this need, but takes a different view. Maybe the most important factors for enabling customizability and flexibility is the separation of abstract software evolution tasks, and their support and implementation by concrete tools (*Separation of Concerns* requirement). This allows to model custom needs and specific tool provisions (requirements *Task Description* and *Tool Description*), and break down large tools into the individual tasks they support, yielding smaller elementary units for easier composition and reuse.

Usability concerns are outside of the scope of this thesis. Its importance, both for individual tools and integrated toolchains as a whole is not questioned, though. Usability of integrated solutions is mostly affected by presentation integration. To some extent, this can be addressed by choosing software evolution environments as target platforms to build upon, which provide means for presentation integration. This has the drawback that tools need to be specifically designed for the chosen platform and technology. An approach towards more flexible ways to provide well-integrated user interfaces is described in Section 16.4.2.

Adoptability, when applied to the toolchain-building support framework to be constructed, is mainly affected by the ease of integration with existing tools. This is reflected by the *Tool Interoperability* requirement, demanding specific support for tool developers when adapting their tools to conform to the approaches' provisions. The adoption barrier can be lowered further by choosing appropriate framework technology equipped with aids that allow easy and light-weight interfacing.

#### Lessons Learned

Kienle has distilled the experiences from his case studies in a set of ten lessons learned, not all of which can be transferred from tool- to toolchain-building. Therefore, Lessons 1 and 2 (concerning scalability, which has been discussed regarding toolchains in Section 5.9.3) are omitted here.

With Lesson 3, Kienle [2006, p. 244] observes that "[p]resentation integration is only feasible if the host components support the same wiring standard", i.e. for tools build for different, incompatible platforms presentation integration cannot be achieved, or only with disproportionate effort. This thesis aims at enabling the integration of vastly diverse tools, but it does not cover presentation integration. The experience expressed by this lesson may be taken as an indicator that there is need for further, dedicated research (see Section 16.4.2 for a brief outline of what this might entail), as presentation integration is clearly desirable.

Lesson 4 states that file-based interoperability can be an effective approach [Kienle, 2006, pp. 244-245]. The observation is limited to the integration of extractors and visualizers, though. In general, many authors consider file-based data integration to be cumbersome, e.g. Sim [2000] (see also Section 1.1 and Section 4.3.2), as it hinders experimentation and iterative processes. However, Kienle also continues to say that simple file-based data integration should be chosen because the means for higher-level (control) integration means are inadequate or immature. This is an aspect this thesis wishes to address and improve upon.

Lesson 5, 6, and 7 deal with customizability, specifically the benefits of scriptable "host" tools, impediments caused by lack of tool customizability, and the value of existing customizations as learning examples, respectively. As previously discussed in Section 5.9.3, this thesis is aimed at making the toolchain as a whole as flexible and customizable as possible, and tries to make as few assumptions about the nature of individual tools as possible for universal applicability. When integrating many different tools, this allows to specify the toolchain in a single, bespoke language, instead of having to deal with various, tool-specific scripting languages.

Lesson 8 describes using a tool familiar to users as host for integration (by customizing and extending it) as beneficial for usability. Clearly, there is a tradeoff, though, between the usability gain, and the restraints imposed by using a particular software evolution tool, workbench, or environment as target platform. Striving to support the integration of arbitrary tools, this thesis values universal applicability over usability.

Finally, Lessons 9 and 10 deal with adoptability, particularly the need for quickly observable benefits of using a tool, and the need to factor in practical experience and feedback from domain experts. The former is definitely an issue to look out for when applied to a toolchain-building support framework built on separation of concerns (*Separation of Concerns* requirement), which can give the impression of having to do more work than before, even if it is actually the same amount of effort performed in a more structured manner (which in turn will save effort in the long run). The same goes for reusability (Objective 2, Section 1.2), which also implies investing more effort up front, to considerably reduce it over time. Since most domain experts can also be expected to be experienced software engineers, it is reasonable to assume that they will recognize the long-term benefits. Also, the objectives and requirements of this thesis have, to a large degree, been derived from the Q-MIG project, which had toolchain-building at its center, and involved an industry partner, so this work is well-grounded in practice.

## 5.10 Summary

Table 5.1 provides an overview of the support offered by the existing approaches discussed in this chapter, with regard to the requirements of this thesis. Because of the diversity of the approaches presented in Section 5.5, they are instead represented by the underlying paradigms they are based on: component-based, service-oriented, and

| | Exchange File Formats | Common Data Models | Workbenches | Environments | CBSE | SOSE | MDSE | SOFAS | TIL | Workflow Systems | Bio-jETI |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Separation of Concerns | ○ | ○ | ○ | ◒ | ◒ | ◒ | ◓ | ◒ | ◒ | ◒ | ◒ |
| Comprehensiveness | ○ | ◒ | ○ | ◓ | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| Task Discovery | ○ | ◒ | ◒ | ○ | ○ | ◓ | ○ | ● | ○ | ○ | ● |
| Task Description | ○ | ○ | ○ | ○ | ○ | ◓ | ◒ | ○ | ○ | ○ | ○ |
| Data Flow | ○ | ○ | ○ | ◒ | ◒ | ◓ | ◒ | ● | ● | ● | ● |
| Control Flow | ○ | ○ | ○ | ◒ | ◒ | ◓ | ◒ | ● | ◒ | ○ | ● |
| Tool Discovery | ○ | ○ | ○ | ○ | ○ | ◒ | ○ | ○ | ○ | ○ | ◒ |
| Tool Description | ○ | ○ | ○ | ◒ | ◒ | ◓ | ◒ | ○ | ○ | ○ | ○ |
| Tool Interoperability | ○ | ○ | ○ | ● | ● | ● | ○ | ○ | ● | ○ | ● |
| Uniform Interfaces | ○ | ○ | ○ | ● | ● | ● | ○ | ● | ● | ◓ | ◓ |
| Reusability | ◓ | ◓ | ◓ | ◓ | ◓ | ● | ○ | ○ | ◒ | ○ | ○ |
| Automatic Coordination | ○ | ○ | ○ | ◒ | ◒ | ◒ | ◓ | ● | ● | ● | ● |

Table 5.1: Scope and extent of support of the related work described in this chapter for the requirements defined in Chapter 3. An empty circle means no support and no influence on the requirement: ○. Foundational support: ◒. Substantial support: ◓. Full support: ●.

model-driven software engineering (abbreviated CBSE, SOSE, and MDSE, respectively). Each of these will be evaluated in detail in Part III. Bio-jETI is listed separately from the other workflow systems, as Section 5.8 has shown that it is substantially closer to the requirements of this thesis than other scientific workflow systems.

As is clearly evident from the table, no existing approach comes close to satisfying all the requirements of this thesis. Due to its focus on presentation and control integration, many approaches that focus on data integration are almost completely orthogonal. For example, SENSEI can be complemented by the use of exchange file formats and common data models.

Workbenches are integrated solutions, rather than solutions for integration, whereas software evolution environments offer support similar to using a general-purpose component framework, and may be suitable as target platforms. Their advantage is that they are made specifically for software evolution. This is usually reflected (only) by a set of available, pre-integrated software evolution tools. Their disadvantage is that they

are less flexible. They were not built with the same objectives as this thesis in mind, and impose restrictions (e.g. usage of a common data model, a certain operating system platform, or programming language) that would impede the realization of those requirements that they do not fully satisfy on their own. In particular, the integration of arbitrary, existing tools can be better facilitated with state-of-the-art, general-purpose component and service frameworks.

In fact, service orientation, which builds on component technology and incorporates its principles, offers the greatest degree of support for the given requirements. In addition to the structure-giving features of components, this is facilitated by its promotion of loose coupling, and the incorporation of workflow (or business process) automation technology in the form of orchestration languages and engines. A prerequisite is to adopt a high-level, implementation-agnostic view of the service concept, and not dilute it with low-level, technical concerns. Model-driven technology is an important complement to the service-oriented aspects, providing the necessary language-building and automation features. This is why these three software engineering paradigms are reviewed comprehensively in the following chapters, to form the foundation of the SENSEI approach.

Workflow systems are not generally thought of as a tool integration technology. While their support for the requirements of this thesis is low, it naturally peaks in those areas most directly concerned with process integration. Most "traditional" tool integration approaches ignore this dimension, although some exceptions have been mentioned. Bio-jETI goes far beyond what most other scientific workflow systems have to offer. While it is specifically aimed at the bioinformatics domain, its individual constituents (e.g., jETI and jABC) are viable candidates to serve as foundation to realize the objectives of this thesis. Still, more general-purpose frameworks have been deemed to be more amenable to be shaped for the full satisfaction of all requirements.

Both SOFAS and TIL are powerful tool integration, toolchain-building, and automation frameworks. Superficially, they may appear to be quite similar to this thesis, but closer analysis and comparison has shown that they are actually aimed at different objectives. SOFAS is a solution specifically focused on software analysis, with a strong focus on data integration, but less on the integration of existing tools. TIL addresses toolchain-building for embedded systems development tools, mainly, and is aimed at more interactive usage scenarios, as opposed to process-oriented automation.

In comparison, what seems to be the most unique feature of this thesis is the observation that a clear and strict separation of a conceptual layer for toolchain specification, and a technical layer for toolchain integration can offer substantial opportunities for supporting and automating the toolchain-building process. It enables simplified specification and automation of the mapping process between the two layers, from what is functionally needed to how it is technically provided. These aspects are expressed the strongest through requirements *Separation of Concerns*, *Task Description*, *Tool Discovery* and *Tool Description*, demanding degree of changeability and flexibility that cannot be achieved by any of the other of the approaches presented here.

# PART III

# Key Technologies

The previous chapters have elicited requirements for SENSEI, elaborating the objectives of this thesis (Chapter 3), reviewed the field of tool integration by providing an overview of its different aspects and placing this work within the presented classification frameworks (Chapter 4), and comprehensively surveyed existing tool integration approaches and other related work for comparison and delimitation (Chapter 5). This has distilled the unique characteristics and novelty of the approach: a focus on process integration that fills a void and complements existing approaches that are mostly aimed at data integration, with separate, tailored support for the two central stakeholders of toolchain building – domain experts and tool developers.

SENSEI strives to yield *flexible* toolchains built and integrated automatically from *reusable* parts to increase overall *productivity*. Therefore, it must be built on foundations that provide strong means for abstraction, encapsulation, expressiveness, and automation. The review of existing approaches has highlighted three engineering paradigms as the best candidates to draw from and built SENSEI upon: *service-oriented*, *component-based*, and *model-driven software engineering*. Their underlying concepts will be presented and evaluated for their possible contributions towards the requirements of SENSEI in Chapter 7, Chapter 6, and Chapter 8, respectively – components being introduced before services because the latter build upon the former.

As a preview to the detailed description of SENSEI in Part IV, Figure III.1 sketches its vision, and the role each of the key technologies presented here are meant to play: On the left, **services** will be used to abstract from concrete tools and implementation details, and represent tasks and techniques in the language of domain experts. On the right, **components** encapsulate tools, so tool developers can equip them with uniform interfaces for interoperability. In the center, **models and transformations** bridge these separated concerns by providing the means to express both sides' artifacts in with the necessary formality to automatically map processes expressed in terms of services onto an appropriate composition of components that implements the desired toolchain.


Figure III.1: The key ingredients of SENSEI.

CHAPTER 6

# Component-Based Software Engineering

Component-based software engineering (CBSE) arose as a software development paradigm in the '90s [Szyperski, 1997], though its roots arguably lie in the *software crisis* of the late 1960s, together with software engineering as a discipline itself [McIlroy, 1968]. At this time, software systems had grown so large and complex that it became increasingly harder to maintain and continue to develop them. To manage the complexity, Doug McIlroy advocated "Mass-produced Software Components"[1], i.e. producing software by assembling it from *reusable sub-components*. This divide-and-conquer approach lowers the size and complexity of individual components, as well as the complexity at the assembly level, by hiding the internals of individual components through encapsulation.

Means to structure software systems on more fine-grained levels were developed and adopted in the 1970s and 1980s, e.g. *structured programming* [Dahl, Dijkstra, and Hoare, 1972] and *object orientation* [Dahl, 2004; Kay, 1993]. Introducing "Software-ICs" (as in *integrated circuit*), Ledbetter and Cox [1985] use the term "component", but the concepts were arguably closer to the level of objects. In the early 1990s, the first component-based technology was developed, among them IBM's *System Object Model* (SOM) [Conner et al., 1992], Microsoft's *Component Object Model* (COM) [Kindel, 1997], and Object Management Group's *Common Object Request Broker Architecture* [*CORBA* 2020]. Especially the latter originally focused more on communication between parts of distributed systems than componentization, and the standard was only amended with a proper component model in 2002. Java got *JavaBeans* in

[1] McIlroy also notes the fallacy of the analogy to mass production, a process which is trivial for software, if taken literally.

1997 [Hamilton, 1997] and *Enterprise JavaBeans* [2019] in 1998. In more recent component technology, terms and (to a lesser extent) concepts from *service-orientation* have seeped in, as in the *Open Services Gateway Initiative* [*OSGi* 2020] and *Service Component Architecture* (SCA) [Edwards and Chapman, 2016]. This is discussed in Chapter 7.

In academia, component-based software engineering began to form as a research area in its own right in the 1990s: For example, the *Workshop on Component Oriented Programming* (WCOP) was founded in 1996, the *International Symposium on Component Based Software Engineering* began (also as a workshop) in 1998, and first comprehensive books appeared around the millennium, e.g. Crnkovic and Larsson [2002], Heineman and Councill [2001], and Szyperski [1997].

This chapter provides an overview of component-based software engineering concepts. In the context of this thesis, components are a key factor towards its reusability objective (Section 1.2), and are mainly reviewed for their potential to contribute to the fulfillment of the following requirements in particular:

**Tool Interoperability.** The support framework must aid tool developers in creating interoperable tools.

**Uniform Interfaces.** The support framework must impose standardized, uniform interfaces on tools.

**Reusability.** The support framework must promote reusability of data transformers.

The outline is as follows: Section 6.1 gives a broad overview of the terminology, key concepts, and their interrelations. The subsequent chapters will then describe those concepts in more detail: Section 6.2 defines components and their properties in general, Section 6.3 introduces component models, and Section 6.4 discusses concrete component frameworks that each implement and conform to a specific component model. Section 6.5 provides a summary of the provisions and utility of component-based software engineering in the context of SENSEI.

## 6.1 Overview

Figure 6.1 depicts a model of the concepts of component-based software engineering and their interrelationships as UML class diagram. The concepts represented by classes in the figure will be explained in the following.

The left-hand side, representing the central concept of *components*, their constituents, and their assembly from *atomic components* into *composite components* is described in Section 6.2. However, the term *service* will admittedly remain a bit vague for now, as a closer examination of the concept will only be made in Chapter 7. For

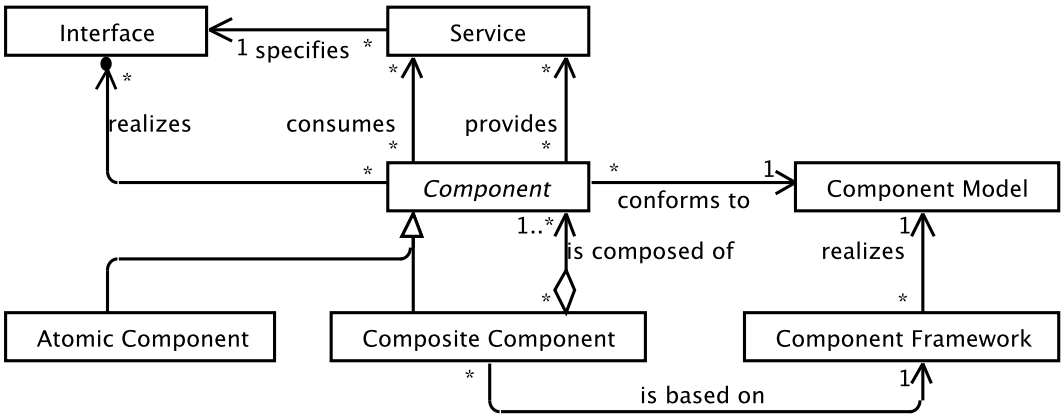



Figure 6.1: Core concepts of component-based software engineering.

now, it shall be sufficient to understand that software components have a purpose, and as such provide some functionality (and may also require some additional functionality to achieve a more complex goal): services represent such required or provided *units of functionality*.

Another concept explicitly depicted in Figure 6.1, because it is essential to the idea of components, is the *interface*. Interfaces define how the functionality represented by services is being provided, how it can be accessed, what information has to be provided, and what kind of information will be returned as a result of its invocation. They can be on a conceptual (implementation-agnostic) level, or on a technical level. Interfaces do *not* include the implementation itself, though, following the general software engineering principle of separation of interface and implementation [Bourque and Fairley, 2014, p.2-3].

On the right-hand side of Figure 6.1, the *component model* and the *component framework* are depicted. The former defines standards on how to describe and use components, while the latter implements such standards and provides a platform for components to run on. They will be defined in Section 6.3 and Section 6.4, respectively.

## 6.2 Components

There is no perfect consensus on what exactly a component is: many different definitions can be found in the literature, each emphasizing different aspects. For this thesis, a clear and binding definition will be derived, after reviewing existing definitions put forward by experts of the field.

A frequently cited definition of software components is by Szyperski [1997, p. 34]:

> “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

To paraphrase, components are assembled to form an application, or more generally, a *component-based software system*. To be of use for other components, the functionality offered to the outside world is specified via *interfaces*, along with the specification of how to invoke it. Components are only used through these interfaces – otherwise, they appear as black boxes. It is via these interfaces that a component offers *services*. Context dependencies refer to the environment the component needs to be deployed to, i.e. for and with which technology it has been built. On the one hand, the definition does not allow for the declaration of dependencies on other components, as this would violate the requirement for independent deployability. This requirement is softened in modern component frameworks like *OSGi* [2020], where components explicitly declare dependencies on other components. On the other hand, as Szyperski [1997, p. 3] writes, “Components are for composition”: along these lines, dependencies can be interpreted as sub-components. Therefore, Figure 6.1 depicts a composite pattern made out of components in general, which can either be *atomic components*, or *composite components*. The latter are assembled of more basic components using *bindings* [Crnkovic et al., 2011], which wire consumed services to services provided by other components with matching interfaces.

Booch, Rumbaugh, and Jacobson [1999, p. 459] define components – in the context of UML – as follows:

> “A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.”

Here, components are physical entities as opposed to *conceptual* – they manifest in concrete, usable, binary form. One component may be swapped for another and deployed in its place, as long as it conforms to the same interfaces.

Heineman and Councill [2001, p. 7] emphasize the *component model*:

> “A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”

The introduction of the concept of a *component model*, which “defines specific interaction and composition standards” [Heineman and Councill, 2001, p. 7], allows to formalize the definition of a component: if it conforms to the component model, it is a component. This concept is useful to argue in terms of basic interoperability of components, and allows to distinguish standard specifications and their implementations (*component frameworks*; see Section 6.3 and Section 6.4).

In this thesis, a component is defined by its properties. The three definitions cited before all contribute different aspects. There are, of course, many more definitions

in the literature. The ones selected here were chosen to provide coverage over all aspects deemed important in the context of this thesis. For the sake of completeness, the definition of the IEEE should be mentioned [Bourque and Fairley, 2014, p.2-10]. It is more or less in line with both Szyperski and Booch, Rumbaugh, and Jacobson, but does not mention the concept of a component model. The definition that will be used for the remainder of this thesis, summed up from the cited literature, is as follows:

**Definition 6.1: Software Component**

A software component is a software system that exhibits the following properties:

1. It is a *self-contained black box*, i.e. it is packaged together with all dependencies or sub-components, and is not modified for use or deployment.
2. It offers all its functionalities – its *services* – only via well-defined *interfaces*, and is therefore *replaceable* with any other components that provide the same set of interfaces.
3. It is described in terms of and conforms to a *component model*.

This definition drops the requirement for binary delivery, as this seems ambiguous and unnecessary. Defining a component as a software system should be sufficient, and also takes care of the *physical* aspect: components actually implement the functionality offered via its interfaces. That is what sets them apart from *services*, which in this thesis are conceptual entities providing only specifications, and not their realization. This distinction is further elaborated on in Chapter 7. Being a black box, components may consume external services, as long as their resolution and use is completely transparent (invisible) to outside observers.

Furthermore, the definition only calls for *provided interfaces*, and omits *required interfaces*. The latter would be conceptually redundant with having explicit dependencies, which are also equated with sub-components.

The notion of *executability*, which some authors use (e.g. Sommerville [2011, pp. 452], and also Szyperski [1997, p. 4]), is explicitly avoided here, though. This, too, can be an ambiguous term: for example, a component depends on a *component framework* as execution environment, so it is usually not executable on its own. Also, executability might be misunderstood to be executable and directly usable by a (human) user. The interfaces a component provides do not have to be user interfaces, though (e.g. command line interfaces or graphical user interfaces); rather, they are usually application programmer interfaces. A component that provides functionality through interfaces implies an ability to be executable in some form, so an additional requirement is not necessary.

Lastly, a component is required to conform to a specific standard that provides concrete rules on how to (technically) describe components. This is the *component model*, which will be defined next.

## 6.3 Component Model

Weinreich and Sametinger [2001, p. 37] provide a detailed overview of the elements of component models. They offer the following definition:

> “A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment. A component model also defines standards for an associated component model implementation, the dedicated set of executable software entities required to support the execution of components that conform to the model.”

Component models provide a standardization of both components and the environment into which they can be deployed and will be executed in. Such an environment is called a *component framework* in this thesis; Weinreich and Sametinger, in the above definition, refer to *component model implementations*, meaning the same thing.

Crnkovic et al. [2011] develop a very comprehensive classification framework for component models. They offer the following definition:

> “A component model defines standards for (i) properties that individual components must satisfy, and (ii) methods for composing components.”

This is a fairly concise definition, but it does not explicitly refer to component frameworks, subsuming their role in the somewhat vague “methods for composing components”. For this thesis, a definition is used that avoids the breakdown into several covered aspects [Weinreich and Sametinger, 2001], which is not needed for further discussion, while explicitly highlighting the relationship to component frameworks:

**Definition 6.2: Component Model**

Component Models provide rigorous standards to govern

1. component description and implementation,
2. component composition and interaction, as well as
3. component framework construction and operation.

The standardization offered by component models enables components (and their developers) to safely make certain assumptions about the environment in which they will be executed in, and vice versa allows component frameworks (which are implementations of such execution environments) to access and interact with components deployed to them in a uniform manner. Component models dictate how components need to be described, how they will be accessed (i.e. how and when data and messages will be exchanged), as well as what kind of services conforming component frameworks have to provide, how they have to be provided, and how components are composed into applications or more complex, composite components. In short, component models ensure basic *interoperability* of components conforming to it.

Without standardization through component models, components could not achieve much. They would, in principle, be reusable, but not *exchangeable*: one component could not be swapped for another offering the same services, because they would be technically incompatible. For the same reason, they would also not be *composable*.

This can be likened to (national) standards for power plugs and sockets: they ensure a general compatibility of electric household appliances with the electrical wiring of private homes. Without it, one would have to adapt each new desk lamp, refrigerator, or hair dryer to the house installation on a much more basic level, e.g. connecting individual wires, adding transformers to adapt to differences in expected voltage or amperage, etc. – instead of just plugging it in.

The analogy also highlights another aspect: when traveling to a country with a different standard, adapters are needed to be able to use, for example, the phone charger that was brought along, but which was built for a market with different socket outlets. Software components are only compatible within the confines of a component framework that adheres to the standards put forward by the used component model.

Prominent component models are (or have been) *COM: Component Object Model Technologies* [2018], the *CORBA Component Model (CCM) 4.0* [2006], *Enterprise JavaBeans* [2019] by Oracle (formerly Sun Microsystems), *OSGi* [2020], and the *Service Component Architecture (SCA)* [2015], which was first developed at IBM, then transitioned into the industry consortium OSOA, and later to OASIS. These component models, along with several component frameworks implementing them, were the subject of a two-staged comparison study to find a suitable infrastructure basis for software evolution tool integration, performed by Ringe [2013][2]. This has led to the selection of SCA for the SENSEI demonstrator SCAffolder (Section 14.2.3). Although SENSEI does not rely on any SCA-specific features that are not also provided by other component models, the inclusion of service-oriented principles can provide higher flexibility in terms of platform and technology independence. An alternative to SCAffolder was provided by Küpker [2015], which uses *WSO2* [2020] as target platform (see Section 14.5).

## 6.4 Component Framework

Having introduced components and component models, the definition of *component frameworks* is straight-forward:

**Definition 6.3: Component Framework**

Component frameworks provide an execution environment that manages components and their interactions, in accordance with a component model.

[2]Microsoft COM was excluded very early on due to its tight coupling with the Microsoft Windows platform, and a requirement for platform independence in software evolution tool integration.

In the literature, they are also referred to as *component model implementations* [Heineman and Councill, 2001]. Crnkovic et al. [2011] seem to use component framework and *component execution platform* (or just *platform*) interchangeably.

The distinction between component models and conforming frameworks means that, in a perfect world, components conforming to a certain component model can be deployed into any component framework also conforming to the selfsame component model, independent of the vendor of the framework. Slight deviations of framework vendors from the component model standard, or component developers' (possibly inadvertent) reliance on non-standardized, vendor-specific framework features may impede this in practice. In case of SCA, the whole attempt at standardization has been blocked for unclear, possibly corporate-political reasons (judging from the publicly accessible *Mailing List Archives of the SCA-Bindings Technical Committee* [2013]), leaving the status of SCA as an industry standard in abeyance (see also Section 14.2.3).

There are several alternative component frameworks for each of the above-mentioned component models, e.g. the Oracle-sponsored *GlassFish Server* [2020] and Red Hat *JBoss Developer* [2020] for Enterprise JavaBeans, Eclipse *Equinox* [2020] and *Apache Felix* [2020] for OSGi, and *Apache Tuscany* [2016] and *Fabric3* [2016] for SCA.

Ringe [2013] suggested Tuscany as the most suitable framework to use as basis for the SENSEI prototype SCAffolder (Chapter 14), based, among other criteria, on the facts that it is a well-established open source project with comparatively good documentation. Being the basis of the SCA implementation included in IBM's WebSphere also secured tool support (SCA modeling tools are available, for example, for Rational Software Architect). The comparatively young framework *SwitchYard* [2020] was released after the study by Ringe. Although based on SCA, it does not claim standard conformance, and is not really advertised as SCA-compatible.

## 6.5 Summary

As was sketched at the beginning of Part III, components are meant to support tool developers in SENSEI: they are expressed in technical terms and wrap around concrete tool implementations. Figure 6.2 refines the right-hand side of Figure III.1 (page 97) to sketch how component-based software engineering concepts fit into the overall architecture of SENSEI: Component models establish standards that enable a basic level of interoperability, as users of a conforming component framework can rely on all components describing their interface and offering their services in the same, well-defined manner (*Uniform Interfaces* requirement). From the perspective of tool developers, this provides the foundation for supporting the creation of interoperable tools, adapters, and transformers during the toolchain-building process (*Tool Interoperability* and *Reusability* requirements; see also Section 3.2.4 and Section 3.2.5).

In the terminology introduced in this chapter, the actual integration of tools into a toolchain becomes a composition of components. The necessary glue logic respon-

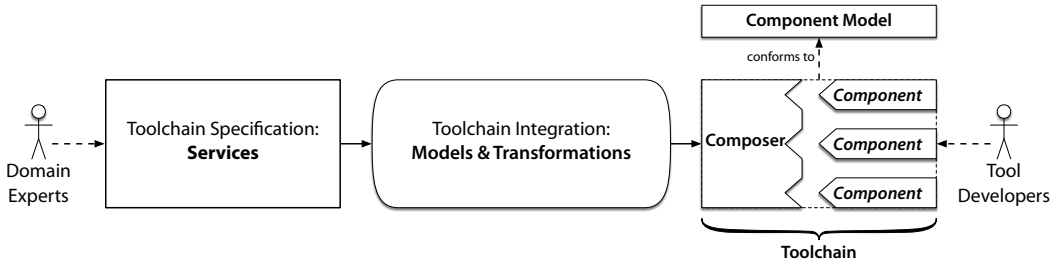



Figure 6.2: Components contributing to SENSEI.

sible for transporting data between components and managing control flow, is itself encapsulated as component, referred to as *composer*. Its inner workings will become substantially simpler by being able to treat all tools in a uniform manner. This is an important prerequisite for automating the integration step of the toolchain-building process, i.e. auto-generating the composer.

Having a component framework as common platform for all components (tools) to be integrated allows to automate the deployment, as well. This is useful for software evolution toolchains, which, due to confidentiality concerns regarding the data being processed (the legacy software systems being modernized), will usually be self-hosted. In this case, tool developers deliver components as their products. In other domains it may be more acceptable to integrate functionality hosted by external parties, shifting the role of tool developers to that of a *service provider*. Such a *software-as-a-service* model (see e.g. Mell and Grance [2011]) relaxes the requirements for uniformity: only the interfaces and modes of communications have to conform to a common standard, while the implementation behind those interfaces would not be restricted to a particular platform defined by a component model. With respect to achieving reusability, in particular that of adapters and transformers, a fully component-based approach would still be expedient, though. SENSEI has been applied in both scenarios – they are described in detail in Chapter 15 and Chapter 16, respectively.

While the term "service" appears as part of some component models, its semantics in this context are still technical in nature and close to the implementation. This is appropriate for tool developers, but for domain experts, SENSEI intends to offer abstractions that are implementation- and technology-agnostic, able to capture the tasks and techniques of any particular application domain. The next chapter will review service-oriented software engineering for potential candidates to fill this role.

CHAPTER 7

# Service-Oriented Software Engineering

Service orientation, and in particular service-oriented architecture (SOA), can be traced back to the middle to late 1990s. The first mentioning of the term "service-oriented architecture" is usually attributed to Gartner Group's Schulte and Natis [1996] (Josuttis [2007, p. 7] provides some details). Service orientation, and SOA in particular, has been motivated by business needs of corporations, and so has always incorporated this "business layer" above the purely technical concerns and concepts, to "bridge the business-IT gap" [Josuttis, 2007, p. 2]. The same cannot be said for component-based software engineering discussed in Chapter 6, which is more clearly rooted in engineering, as opposed to management.

SOA went through a massive hype in the 2000s (cf. *Gartner Hype Cycle* [2015]), which blurred the definition and concepts of SOA, and was also closely associated with a hype of web service technology [Vaughan-Nichols, 2002]. Disillusionment followed – Manes [2009] pronounced SOA dead. As a response to the perceived complexity and over-standardization of the web services technology stack, RESTful web services have gained momentum [Richardson and Ruby, 2008]. On a more conceptual level, the notion of *microservices* has emerged, which may be considered a successor, or a concrete approach to executing SOA [Lewis and Fowler, 2014].

The concepts of service orientation, and service-oriented architecture, are mainly aimed at building and integrating large enterprise applications, or even applications that span multiple corporations. Among its goals is the ability to reuse functionality that previously may have been developed redundantly, side-by-side in different divisions of a company. Furthermore, it should yield applications build for adaptability in the face of changing and expanding business requirements.

Service orientation has, to some degree, evolved from the component-based paradigm. Where reusability is one of the main motivations for component-based software engineering, for service-orientation it is *flexibility*. At its core lies the idea to abstract from concrete implementations, platforms, and technologies, and describe *what* units of functionality are doing, but not *how* they are doing it. These "units of functionality" are referred to as *services*. The abstraction facilitates reuse, which is already covered by components, but beyond that, services can be assembled into loosely coupled applications without referring to how and where they are implemented. The binding to concrete implementations can happen dynamically, possibly at runtime. This, in theory, yields scalable, distributed applications able to evolve easier and faster than monolithic applications, or applications build from more tightly-coupled components.

The way services are assembled as opposed to components is a key difference between the two paradigms, noted e.g. by Sommerville [2011, p. 455]: in a component-based application, its components are *internal*, i.e. part of the application itself, whereas in service orientation, services are referred to but remain *external*.

In SENSEI, services and service-oriented principles take on the most prominent part of the overall approach, and are the key to achieving Objective 1 – increasing toolchain flexibility to retain agility within projects. Adopting services in SENSEI is a prerequisite for almost all requirements elicited in Chapter 3. The following requirements are most directly supported by service-orientation:

**Separation of Concerns.** The support framework must establish a clear separation between toolchain specification and toolchain integration.

**Task Discovery.** The support framework must aid domain experts in finding existing techniques relevant to a given task.

**Task Description.** The support framework must provide domain experts with a means to describe required properties of tasks in a standardized way.

**Data Flow.** The support framework must aid domain experts in specifying the data flow between tasks.

**Control Flow.** The support framework must aid domain experts in specifying the control flow between tasks.

## 7.1 Overview

Figure 7.1 shows the core concepts of service-orientation, and their interrelations, which are explained in the remainder of this section. The central term is, of course, the *service*, which provides its functionality through an *interface* that defines how it is accessed, technically (*signature*), as well as what it is meant to do (the *functional* and

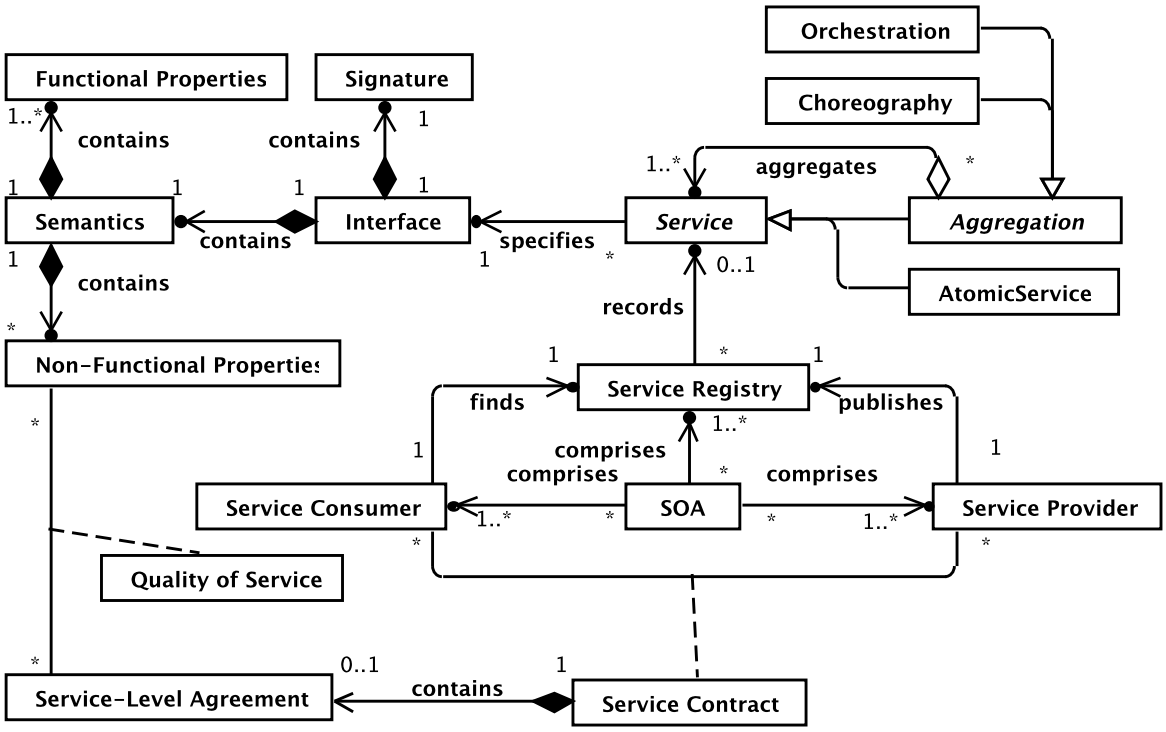



Figure 7.1: Core concepts of service-oriented software engineering.

*non-functional properties* of its *semantics*). A definition for services in these terms will be derived in Section 7.2. This will be further discussed in Section 7.3, which summarizes desirable service properties to be taken into account when designing services, particularly non-functional properties that define the *quality of service* at runtime, and classification schemes that can help with determining appropriate service granularity.

Section 7.4 introduces *service-oriented architecture* (*SOA*), an architecture pattern dealing mainly with *service consumers* dynamically discovering services using a *service registry* filled by *service providers*. SOA initially popularized service-orientation, but is now often considered obsolete [Manes, 2009]. It is discussed to provide context, and learn from mistakes made during its rise and fall. Also, the concept of service *orchestration*, which does have great relevance for this work (particularly in terms of requirements *Data Flow* and *Control Flow*), is closely linked with SOA, and is introduced in Section 7.5.

## 7.2 Services

The term *service* has already been used in previous chapters (in fact, it appears in the title of this thesis); Section 6.1 introduced a preliminary and minimal working definition: services are units of functionality. In the context of enterprise software systems, the service abstraction is used to describe functionality needed as part of business processes to be automated by appropriate software support. It is meant to provide high degrees of flexibility to account for shifting business requirements, and interoperability to integrate processes across departments and organizations.

Component-based and service-oriented software engineering have many commonalities [Breivold and Larsson, 2007], so it is of particular interest to differentiate the central building blocks of these paradigms, components and services, respectively. Both can be considered black-box abstractions, whose functionality is described in terms of, and accessed exclusively through, explicit interfaces. A relationship has already been established in Section 6.1, and is depicted in Figure 6.1: components *provide* and *consume* services. Some important aspects to further distinguish services and components include the following:

- Services *specify* functionality [MacKenzie et al., 2006, p. 12], components *implement* functionality.
- Services remain *external* to systems that use them, components become *internal* parts of systems they are composed into [Sommerville, 2011, p. 455].
- Services are modeled with respect to business functionality, components are also structured by considering technical aspects [Vogel et al., 2011, pp. 206-207].

In literature on service orientation, this distinction is rarely made so clearly, however. In particular, many authors define services in terms of software programs or as special kind of components [Melzer and Eberhard, 2010, pp. 14-15, Erl, 2007, p. 39, Josuttis, 2007, p. 300]. In this thesis, this view is expressly rejected: identifying services with the implementation of functionality, rather than their specification, or considering services to comprise both specification and implementation, is confusing at best. Blurring the boundaries between services and components unnecessarily imposes technical restrictions and prematurely curtails the solution space. In the process, much of the intended flexibility may be lost.

In contrast, MacKenzie et al. [2006, p. 12] leave the implementation completely open, going so far as to expressly allowing manual realization (i.e. being carried out by humans instead of software) of the defined functionality. A high-level of abstraction with services that are completely technology-agnostic enable *(vendor-)diversity*, i.e. a highly heterogeneous application or tool landscape is hidden behind the interfaces of services. This is a key prerequisite for interoperability, flexibility, and agility.

Definitions of the term *service* vary widely, and are hard to reconcile. Instead of reviewing literature definitions individually, common aspects [Josuttis, 2007, pp.25-34, Erl, 2005, p. 40ff] are listed in the following, and then summarized into a coherent definition.

The functionality to be provided by a service is described completely in terms of an **interface**. The interface is described in terms of syntax by its **signature**. The **se-**

**mantics** define what the service is supposed to do [Josuttis, 2007, p. 28; Melzer and Eberhard, 2010, p. 15]. Service semantics can be further subdivided into **functional properties** and **non-functional properties**. Josuttis refers to service interfaces defining both signature and semantics as *well-defined*.

Melzer and Eberhard require the interfaces (or *service descriptions*) to be machine-readable. A related requirement is the property of being *self-descriptive* [Vogel et al., 2011, pp. 206-207]. This is aimed at making services *discoverable*: services are advertised in central repositories for late binding over a network. *Network transparency* is another common attribute of services, though few authors (among them Melzer and Eberhard) mention it explicitly. Arguably, it is a technical feature that needs to be realized by software implementing services, rather than a property of the services themselves. The fact that services are implementation-agnostic ensures that they can be mapped onto technologies that realize network transparency, if it is needed.

Interfaces are sometimes described as enabling access to the *capabilities* of services; this term is avoided here, for now, as it has a very particular meaning in the context of SENSEI (see Chapter 9). Josuttis views services as consisting of one or more *operations*, that provide different parts of the service's functionality and are invoked by service consumers, i.e. they correspond to functions or methods. This is not adopted in this thesis: an additional means of hierarchical decomposition is considered unnecessary, as services can already be aggregated by means of orchestrations (see Figure 7.1 and Section 7.5). A means of grouping units of functionality, which would then be represented by operations, degrading services to be mere containers, may be counteractive to the goals of service-orientation. It seems to encourage mixing behavior and state (as in object-orientation), and designing stateful services with conversational interfaces, which are more complex to specify, implement, and use (but see Marino and Rowley [2009, pp. 109ff] for an opposing view).

Another common ideal is that services should be abstractions of *business* or *domain functionality*, only, as opposed to functions needed to support the overall system on a technical level [Josuttis, 2007, p. 26]. On the one hand, keeping to this code can be viewed as an instance of *separation of concerns*, and simplifies arranging complex behavior from basic services, exclusively using the language of the domain. In practice, "helper services", e.g. for data manipulation, may often creep in, leading to a mix of domain and technical aspects on the same level of abstraction. On the other hand, since the building blocks of both the application domain as well as of technical issues are units of functionality, modeling both as services arguably serves consistency: introducing a separate concept for technical functionality also adds a degree of complexity. Ideally, all technical aspects should be hidden behind the interfaces of services representing domain functionality.

Finally, services should strive to be *stateless* and *idempotent*. The former means that an invocation of a service's functionality will yield the same result every time, i.e. the result only depends on the explicitly provided input data. The latter, related

concept demands that invoking a service's functionality multiple times with the same input data has the same effect as only invoking it once. Statelessness makes using services more predictable, and their semantics are simpler to express and understand. Also, implementations are considered more scalable as they do not have to keep track of state and therefore have a smaller memory footprint. Idempotence simplifies inter-service communication: if implementations must use a potentially unreliable means of communication, invocation messages can safely be repeated in case the original message is suspected to have been lost. In practice, these requirements for services are usually substantially relaxed. The reliance on the state of an external database, for example, is not considered to be part of the service state [Josuttis, 2007, pp. 191ff; Erl, 2007, pp. 325ff].

In summary, the following definition is adopted for the remainder of this thesis:

**Definition 7.1: Service**

A service specifies a unit of domain functionality, described in terms of an interface on a conceptual, technology- and implementation-independent level. Service interfaces consist of 1) a *signature*, which defines an abstract usage syntax, i.e. input and output parameters, 2) a semantic description that relates input and resulting output data, and expresses what the service is supposed to do (functional properties), and optionally operating ranges for the quality of service (non-functional properties).

Due to the prescribed, abstract nature of services according to this definition, aspects like information hiding, network transparency and loose coupling are considered emergent properties, and are therefore not mentioned, explicitly.

## 7.3 Service Design

Some desirable service properties have already been mentioned in Section 7.2, e.g. statelessness. Providing more comprehensive guidelines on how to design services properly, i.e. in a way that maximizes their intended benefits such as increased flexibility through loose coupling, is out of the scope of this thesis. Still, service design is a necessary task in any service-oriented approach, and so this section briefly sums up some common service properties to strive for. However, there is probably not a single right way to model services, and on a very general level, the trend has swung from advocating large and coarse-grained services in the context of SOA, to small, fine-grained *microservices* [Lewis and Fowler, 2014; Newman, 2015]. Two means to aid service design are reviewed in this section: desirable *properties* derived from practical experience, and service *classification* schemes that help describe services by narrowing down their scope and relate them to similar ones.

Josuttis [2007, pp.25-34] discusses a quite comprehensive set of concepts and properties surrounding the service term, comprising *service signature*, *well-defined interfaces*, *pre- and postconditions*, and *quality-of-service (QoS) and service-level agreement (SLA) capability* (jointly also referred to as *service description* or *service contract* by other authors), *self-containedness*, *granularity*, *reusability*, *discoverability*, *statelessness*, *idempotency*, *composability*, *technicality*, *vendor-diversity*, *interoperability*, and *web service implementation*.

Erl [2005] also covers most of these terms, with the exception of idempotency (many of these aspects are found under "common characteristics of [...] SOA" [Erl, 2005, p. 40ff], as opposed to properties of services). Erl [2007] postulates eight core "principles of service orientation", namely *contracts*, *reusability*, *autonomy* (mostly synonymous with self-containedness), *statelessness*, *discoverability*, *composability*, as well as *coupling* and *abstraction*. The latter two properties are not covered by Josuttis. All these properties are interrelated, as well; Erl [2005, pp. 311-321] provides a detailed discussion of how they affect each other.

Three of these properties are summarized in the following, while other aspects will be discussed elsewhere (or have already been covered): *Composability*, the ability to combine several "smaller" services into a single, new service to provide a more complex functionality, is discussed in detail in Section 7.5. Service discoverability, service contracts, and other concepts related to service-oriented architecture are described in Section 7.4, which also touches on issues of technology fixation, e.g. implementation using Web Services [Booth et al., 2004]. *Technicality* refers to supporting infrastructure services, that do not provide a functionality of the application domain, and will be picked up later on, when discussing service classification.

**Abstraction in Support of Diversity and Interoperability.** A key prerequisite for most other service properties is that no assumptions should be made about *how* services are implemented; they only define *what* they are providing in terms of functionality. This enables *(vendor-)diversity*, i.e. a highly heterogeneous application or tool landscape (cmp. Section 7.4) is hidden behind the interfaces of services, which in turn facilitates *interoperability*, flexibility, and agility.

**Self-Containedness.** This property advocates that services should be in full control of the functionality they are providing (also compare service *autonomy* demanded by Erl [2005, pp. 303-307]). Considering that services are merely the specification, this seems mostly a concern for their implementations, however, it also implies that service semantics should not be described in terms of other services, to minimize the impact of service changes. Another implication is that service interfaces should be narrow and simple: Josuttis [2007, p. 29] warns of complex data types used in interface descriptions, which may lead to service interdependencies, negatively affect universality and reusability, and might indicate poorly chosen service granularity.

Furthermore, Josuttis [2007, p. 38-39] also cautions against attempting to fully harmonize the data model of a business or domain: there are to many different concerns of different stakeholders to reconcile, leading to bad compromises, and extremely complex yet incomplete models, and the dependence on common data models impedes independent evolution. Several examples of intricate issues from industrial practice, that may arise when attempting such tight data integration, are found in Schmidt, Otto, and Österle [2010].

**Granularity and Reusability.** The question of granularity is on what level to stop decomposing services into smaller sub-services. Reusability can be assumed to increase, the more fine-grained the services are, although there is probably a lower bound, at which service functionality becomes so trivial that the overhead of assembling services exceeds reuse benefits. Many small services also incur a higher communication overhead than a few larger ones, which might be expensive, especially in a highly distributed environment, while coarse-grained services may exchange large data sets, of which only a fraction might be really required in certain scenarios [Josuttis, 2007, p. 164].

The choice of the right granularity depends on the context and the kind of functionality the service or process step is representing. Therefore, there are classification schemes for services (called *service models* by Erl, 2005), to help organize services for different uses on different granularity levels.

A common, *hierarchical* classification, distilled by Josuttis [2007, pp. 61-73], and apparently based on Krafzig, Banke, and Slama [2005, pp. 69ff] (which has additional classes and slightly differs regarding terminology) and Erl [2005], is depicted as class diagram in Figure 7.2. A hierarchy of three levels is introduced, with *basic services* at the bottom, *composed services* above that, and *process services* at the top. With each service type, a "style" of SOA (see Section 7.4) is established, as well: *Fundamental SOA*, *Federated SOA*, and *Process-Enabled SOA*.

Basic services provide an elemental (business) functionality and are atomic, i.e. they should not, or cannot reasonably be decomposed into more primitive sub-services. Their classification, however, *is* broken down further into *data services*, which act as abstractions from databases and provide the ability to read and write data, and *logic services*, which perform computations and implement elementary business rules.

*Composed services* are assembled from basic services, as well as other composed services to realize more complex tasks. A special kind of composed services are *adapter services*, which wrap around existing services to achieve a degree of interface compatibility, e.g. to accommodate for functionality offered by legacy systems. Composed services are further characterized as being short-running and stateless – workflows with these properties are referred to as *micro flows* (see also Hentrich and Zdun [2009] and Manolescu [2000]).

Long-running, interruptible business processes, in contrast, are considered *macro flows*, and are modeled as *process services*. Because these processes may run for a

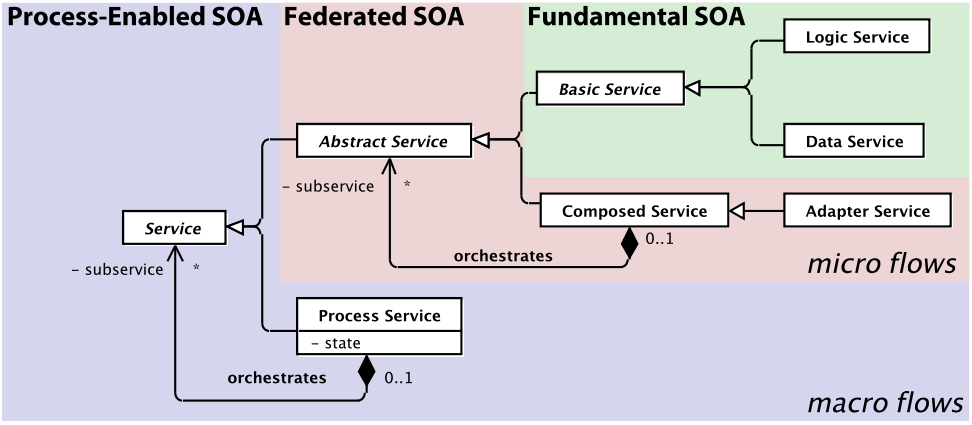



Figure 7.2: Hierarchical service classification represented as class diagram, based on the description given by Josuttis [2007, pp. 61-73].

long duration, may be paused, and may involve human interaction, they are usually stateful, to allow for their suspension and continuation at a later time. This distinction between micro and macro flows makes sense for enterprise applications, but is less pronounced when applied to software evolution toolchains, which are often expected to run tasks in batches.

Another example of a classification scheme is one by Cohen [2007], which distinguishes services according to their *mode of use* (also see Jelschen [2013]). Also worth mentioning, although targeting components rather than services, are the four major, generic software categories, referred to as *software blood types*, of the *Quasar* methodology ("quality software architecture") introduced by Siedersleben [2004, pp. 74ff]. To a degree, these principles can be applied to services just as well:

**Blood type 0** refers to very common, globally available utility functionality,
**Blood type T** represents *technical* application-independent infrastructure,
**Blood type A** subsumes *application*-specific business logic, and
**Blood type R** transforms data from one *representation* to another.

Mixing these blood types is possible, but discouraged. In particular, mixing aspects of A and T in a single unit of functionality (service) is problematic, as it makes them more complex and thus less maintainable, and reduces reusability, as it binds the application-specific functionality to purely technical aspects. The blood types are ordered from generic (Type 0) to specific (Type R), and interfacing services should only expose concepts of the more generic type to each other. The R Blood Type, a separate category for transformers, is another reason why Quasar's software blood types

are considered here, since the *Reusability* requirement explicitly identifies the need for reusable data transformers. Siedersleben emphasizes this requirement: if the necessary translations needed to get two services to communicate was handled internally by one of them, it would be "polluted" with external concepts from the other service, impeding reuse.

## 7.4 Service-Oriented Architecture

An overview over service orientation cannot be complete without mentioning SOA. SOA was befallen by a babylonian confusion regarding its terminology and concepts in the wake of a massive hype during the early 2000s. Many readers will therefore have preconceptions and will make implicit assumptions that would potentially lead to confusion in the remainder of this thesis, so it is imperative to discuss the different facets and incarnations of SOA, and contrast them with the view towards service orientation adopted here. In many cases, the SOA hype could not live up to expectations, but while SOA has been declared "dead" [Fowler, 2005b; Manes, 2009], the underlying principles are considered sound. It is rather that they seem to have been widely misunderstood or ignored, and upstaged by tool vendors and consultants, who were trying to sell SOA-branded infrastructure and expertise, respectively [Newman, 2015, p. 8].

Josuttis [2007, p. 11], who offers one of the better treatises of the topic, states that there are too many different definitions. These definitions usually do not provide sufficiently clear conditions to decide whether a given architecture is service-oriented or not. In fact, there is not even consensus about whether SOA actually is an architecture – according to Josuttis [2007, p. 12] it rather is a *paradigm*, not a concrete architecture. It seems like it could have been intended to be a *reference architecture* (in the sense of Winter [2000, pp. 106, 110]), and there are standards establishing SOA as such [ISO/IEC 18384, 2016; Laskey et al., 2012; MacKenzie et al., 2006]. The following is the definition Josuttis [2007, p. 24] arrives at:

> "SOA is an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners."

This establishes a certain scenario in which SOA can, should, or must be employed, but does nothing to exclude alternative approaches. For example, CORBA [*CORBA* 2020] was meant to fill this need in the nineties (and failed, according, for example, to Henning [2006], and Sommerville [2011, p. 482]). It seems to fit this definition, yet it is probably not an instance of SOA, at all, as Josuttis [2007, p. 23] explicitly delimitates SOA and CORBA, saying they are "exact opposites".

A definition offered by Erl [2005, p. 54] names some properties against which an architecture could be tested to see if it is indeed a SOA. However, it does not name any constituents of a SOA – it is more a description of a problem to be solved, than of a

concrete solution. Another issue is that it directly contradicts Josuttis [2007, p. 22], by defining SOA in terms of a specific technology stack, namely web services, instead of being technology-neutral.

Blending SOA with a particular implementation technology, especially web services, is common. Zimmermann, Tomlinson, and Peuser [2005, p. 161], who focus on web services exclusively, simply define a SOA to be:

> “ [...] any product and project architecture conforming to the W3C WSA base architecture [...]. ”

Here, *W3C WSA* refers to the *Web Service Architecture* standard [Booth et al., 2004]. It mentions some aspects that focus more on individual services than the overall architecture, such as technology-independent specification (“logical view”), “description orientation”, and coarse granularity. These aspects are discussed in Section 7.2. The concepts relating more directly to the architecture describe it as *message-oriented*, *network-oriented*, and *platform-neutral*.

The asynchronous passing of *messages* is meant to provide the desired low degree of coupling, as opposed to synchronous, remote procedure calls [Menascé, 2005]. However, Richardson and Ruby [2008, p. 19] assert that many web service-/SOAP-based systems are in essence using remote procedure calls, wrapped in messages.

The infrastructure required for message-oriented information exchange is called *message-oriented middleware*. More concretely, the necessary facilities are often provided by an *enterprise service bus* (ESB), which is sometimes referred to as an architecture, and sometimes considered a piece of software – there are lots of products advertised as ESB, but no universal notion of what such a framework needs to provide seems to exist. Josuttis [2007, p. 18] views an ESB as a fundamental ingredient, and therefore a required part of a SOA.

*Network orientation* is actually a bit redundant, since the definition also establishes *distribution* as a key property of SOA-based systems. *Platform neutrality* is an important aspect – Josuttis [2007, p. 14] highlights heterogeneity as a main driver for SOA.

All of these aspects have relevance for SENSEI: it strives for loose coupling to achieve its *flexibility* objective. However, low coupling can be achieved through means other than message passing, especially if the process to be automated is known and made explicit, so that it can be enacted by a central controller. The potential need for distribution was identified in the Q-MIG project (Section 2.3.3). The same is true for the need for platform neutrality, though the heterogeneity of software evolution tools was pointed out even earlier as a fundamental reason for the integration challenges (Section 1.1), and it is reflected, for example in the *Comprehensiveness* requirement.

Another view of SOA is of particular interest with respect to SENSEI’s requirements for discovering available tasks and tools (*Task Discovery* and *Tool Discovery* requirements), and mapping tools to tasks (*Tool Description* requirement): this “classical” definition is in terms of the *SOA triangle* [Marx Gómez, 2019; Melzer and Eberhard, 2010], sometimes also referred to as the *publish-find-bind-execute* interaction pattern (or some-

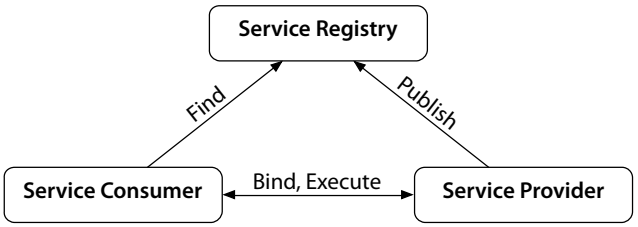



Figure 7.3: The *SOA triangle*, depicting the *publish-find-bind-execute* interaction pattern.

thing similar, e.g., see Ran [2003] and Zhu [2005]). A depiction is shown in Figure 7.3; the terms and their interrelations are also integrated in Figure 7.1.

*Service providers* **publish** the services they offer in a (possibly public) *service registry*. *Service consumers* can then search the registry for services, to **find** one suitable for a given task. Only then will the service consumer **bind** to the chosen service of a provider, using a specific communication technology, e.g. SOAP over HTTP in the case of web services. The service can then finally be **executed**, which entails sending and receiving messages via the established binding.

It seems that this kind of definition has quietly fallen "out of style", probably because this pattern was seldom followed [Michlmayr et al., 2007], and the standard associated with the service registry, UDDI (Universal Description, Discovery, and Integration), was never widely adopted [Sommerville, 2011, p. 510], partly due to its complexity [Richardson and Ruby, 2008, p. 309].

Still, service discovery remains an important factor for service-orientation, and for SOA in particular, as it is a key ingredient in facilitating loose coupling between services, thus yielding highly flexible system architectures. Melzer and Eberhard [2010, pp. 16-17] likens service registries to a kind of "yellow pages" for services, and highlights the need for service classification (discussed in Section 7.3) to be able to easily find services. Erl [2007, p. 40] refers to them as *service inventories*, and stresses their role as a "independently standardized and governed collection of complementary services". These are two complementary aspects: service standardization is desired so services are reusable and their implementations are interchangeable, while the role as a registry is more of a lookup table to find concrete providers and implementations of particular services. This is why in SENSEI, there are two separate concepts for this: the *service catalog* and the *component registry* (see Part IV).

Service discovery as described by Marx Gómez [2019] is illustrated in Figure 7.4. It further elaborates the discovery process as follows: A service consumer ("Software Element A") specifies requirements which are sought to be fulfilled by a service, while service providers ("Software Element B") specify the capabilities they offer through services. Matching requirements and capabilities filters out those services and service

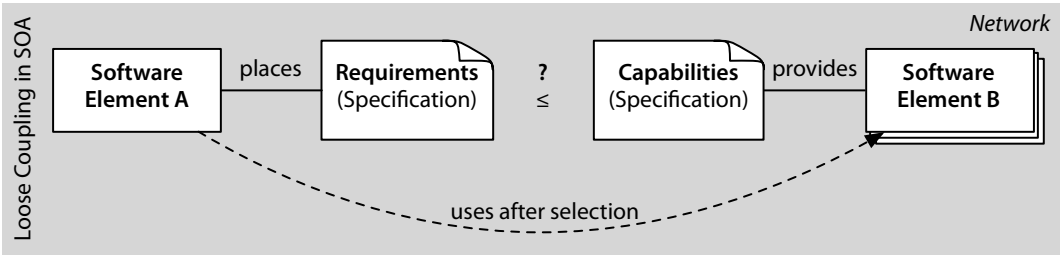



Figure 7.4: Service discovery and loose coupling in an SOA. This figure is a faithful reproduction and translation of the one presented by Marx Gómez [2019].

providers, which satisfy all requirements (at least).

As indicated in Figure 7.1, the two parties then enter into a *service contract*, which, amongst others, specifies *service-level agreements* (SLA). These regulate the required *quality of service* (QoS), i.e. specific, minimal conditions for non-functional service properties such as reliability, availability, security, etc. In this thesis, quality of service is therefore not considered a part of the service description, as it is negotiated and determined on a per-customer basis.

Discovery mechanisms like this have seldom been realized in actual SOA implementations, leading to tightly coupled, inflexible systems in practice [Michlmayr et al., 2007]. Erl [2007, p. 367] stresses the importance of service discovery, yet admits that its inclusion "represents an early incarnation of SOA that has long been surpassed". Usually now, discovery is carried out by humans at design time [Erl, 2007, p. 371], which relinquishes dynamic binding and tightens coupling, moving this form of service-orientation close to what can already be achieved with component-based approaches. This means that for SENSEI, which needs such a functionality (*Tool Discovery* requirement), there is no off-the-shelf solution offered in the SOA context.

Summing up, the following definition is adopted, relying on the understanding of services established in Definition 7.1:

**Definition 7.2: Service-Oriented Architecture**

A service-oriented architecture (SOA) describes a distributed software system in terms of coarse-grained services, which are coordinated externally, and which communicate indirectly via messages over a network.
In *classical SOA*, services are published by their *providers* in *service registries*, which service *consumers* can browse and choose from, potentially using late (runtime, dynamic) binding to maximize flexibility.

By 2009, the hype around SOA had passed. It had caused such a significant level of confusion [Erl, 2005, p. 2] that it had lost much of its popularity. While there were some success stories of implementing SOA, for example at Amazon [Gray, 2006], many companies failed in that attempt. In an empirical study, Kokko, Antikainen, and Systä [2009] report non-technical challenges as the main impediments for SOA adoption, namely resistance to changes by the workforce and lack of (business process) modeling experience. Another frustration was with the web services technology stack that was perceived to be overly complex [Sommerville, 2011, p. 483]. As a reaction, RESTful web services [Richardson and Ruby, 2008] were taken up as an alternative, which has its critics, as well [Nurkiewicz, 2015].

More recently, *microservices* have been proposed [Lewis and Fowler, 2014], sometimes considered to be a successor of, or alternative to, SOA. Among other issues, Newman [2015, p. 8] criticizes the lack of guidelines regarding service size, granularity, and how and where to split existing system functionality into services appropriately. Microservices focus on system decomposition and a significantly reduced service size, sometimes citing the UNIX philosophy as an inspiration or analogy, which strives to create small, reusable tools which do exactly one thing well (also compare the *single responsibility principle* by Martin [2003, p. 95]).

Manes [2009], in an "obituary" to SOA, describes how it has not been able to deliver on its promised benefits for most organizations that tried to implement it. While SOA as a word is declared "dead", the underlying concepts of service-orientation are still considered valid. One reason given for the failure or misunderstanding of SOA is the focus having been on technologies like web services, RESTful services, and enterprise services busses, as opposed to the basic, architectural principles. Fowler [2005b] came to a similar conclusion four years earlier:

> "[SOA is] beyond saving – so the concrete ideas that do have some substance need to get an independent life."

In Section 7.6, these underlying ideas, together with lessons to be learned from the perceived failure of SOA, will be summarized, to base SENSEI on sound concepts, and not repeat previous mistakes.

## 7.5 Service Orchestration

In the context of the toolchain-building support framework to be created, service orientation is considered as one cornerstone, mainly due to its attributed ability to deliver the desired, high degree of flexibility. In particular, means to coordinate tools into a toolchain in terms of data and control flow (requirements *Data Flow* and *Control Flow*, respectively) are wanted, which allow for easy and fast adaptation. In service-oriented software engineering, arranging individual services in such a way to jointly achieve a more complex goal, is called *orchestration*. As with the whole field, the terminology is not used consistently by different authors and practitioners.

This section gives an overview of service orchestration and related concepts, and establishes a coherent definition. First, origins, closely related terms, and influences are reviewed, e.g. *workflow technology*, *business process automation*, and *enterprise application integration* (Section 7.5.1). Then, orchestration is defined in contrast to the closely related concept of *service choreography* (Section 7.5.2).

### 7.5.1 Origins

Service orientation originates from the contact points of business concerns and IT. An influence of the former is thinking in (business) processes. Workflow management systems appeared in the 1990s [Hofstede et al., 2010, p. 23] to monitor and partly or fully automate the execution of business processes.

Distinctions made between the terms *process* and *workflow* vary widely, and often they are used more or less synonymous. Historically, workflow management systems became business process management systems in the 2000s [Hofstede et al., 2010, p. 23], so the terminology shifted towards "process". Hofstede et al. [2010, p. 3] basically equates business process automation with workflow management. In works focused on technical aspects, as opposed to business concerns, a common distinction is that the process is the more abstract representation, while a workflow is a technical realization of a process, i.e. can be automatically executed; this view is also in line with the *workflow management coalition reference model* [Hollingsworth, 1995]. That original standard does not seem to be of great relevance anymore in modern business process automation [Hofstede et al., 2010, p. 7]. A comprehensive discussion of the two terms is provided by Draheim [2010, pp. 75ff]. To highlight either the abstract or technical nature, the terms process and workflow, respectively, are used in this way for this thesis. If the distinction is secondary, the term process will be favored, though.

*Orchestration* is another term that can be considered mostly synonymous with either workflow or process, but is used only in the context of services. The term is sometimes also used to refer to the practice of defining processes or workflows of services, and as a general design principle to integrate existing services and business processes (e.g., see Erl [2005, pp. 200-201]). When used to mean a specific kind of process, the distinguishing elements of orchestrations are:

- The basic abstractions are services, as opposed to more generic concepts like tasks or activities.

- An orchestration represents a service itself [Erl, 2005, p. 201] – orchestrations and services form a composite pattern, with the former being the composites, and the latter the leafs.

In short, an orchestration can be defined as a service-oriented process:

**Definition 7.3: Service Orchestration**

A service orchestration is a process specification in which individual steps are identified with services. Orchestrations define

- the sequence of service processing, including conditional branching, loops, and concurrency (*control flow*), and
- the data exchanged between services, as well as consumed and produced by the orchestration as a whole (*data flow*),

in a manner that can be enacted by a central controller. An orchestration is a service, with the described behavior defining its semantics, and the consumed and produced data defining its signature.

In enterprise application integration, orchestration is considered as one integration pattern amongst many: Hohpe and Woolf [2004, pp. 312ff] refer to it as *process manager*. It is characterized by introducing a central controlling unit that executes workflows, invokes services, and routes data as needed.

On a technical level, several standards for orchestration languages have been developed. The most widespread one is the XML-based *Business Process Execution Language* (WS-BPEL 2.0, or simply BPEL), standardized through OASIS [Jordan et al., 2007]. It relies on WSDL service definitions to refer to services being orchestrated. BPEL has execution semantics that allows its instances to be interpreted by *orchestration engines*, but it lacks a graphical notation. OMG's *Business Process Model and Notation (BPMN) Version 2.0* [2011] is just the opposite in this regard. The two standards can thus be used complementary, for example using a mapping from BPMN language features to corresponding ones in BPEL, which is part of the OMG standard. UML activity diagrams are another option to depict orchestrations. Although BPMN possesses language concepts specifically aimed at business process modeling, in terms of general expressiveness, the languages are quite similar, as shown by Russell et al. [2006] and Wohed et al. [2006], evaluating the languages against their well-established set of workflow patterns [Aalst et al., 2003]. These are also the authors of *Yet Another Workflow Language* (YAWL; Hofstede et al. [2010]).

Integrating services using orchestrations can deliver an increase in flexibility (Objective 1), as processes governing toolchains are defined and maintained in a single place, as opposed to being distributed among all the tools being used (Erl [2005, p. 205]; Hohpe and Woolf [2004, p. 312]). The central logic also facilitates abstraction from tools and their technical interoperability issues to services.

The centrality of orchestrations can become a drawback in terms of scalability [Josuttis, 2007, p. 96], representing a possible single point of failure [Hohpe and Woolf, 2004, p. 320]. An alternative or complementary approach to which service orches-

tration is often contrasted is service *choreography*. Recent reinterpretations of service orientation like *microservices* [Newman, 2015], favor such a decentralized, and usually event-based approach.

### 7.5.2 Orchestration and Choreography

As always, use of terminology varies: Zimmermann, Tomlinson, and Peuser [2005, p. 392] make no distinction between orchestrations, choreographies, aggregations, and compositions of services. Both Zimmermann, Tomlinson, and Peuser [2005, p. 184] and Bieberstein et al. [2008, pp. 96-97] use the term "choreography" with respect to BPEL, while Josuttis [2007, p. 97] posits BPEL as "a pure orchestration language". The latter is the much more common notion, though there exist approaches to extend BPEL with concepts to also describe service choreographies [Decker et al., 2007].

The concept of service choreographies is usually introduced in contrast to service orchestrations. For example, Josuttis [2007, p. 295] offers the following definition:

> "A way of aggregating services to business processes. In contrast to orchestration, choreography does not compose services to a new service that has central control over the whole process. Instead, it defines rules and policies that enable different services to collaborate to form a business process. Each service involved in the process sees and contributes only a part of it."

Erl [2005, pp. 208ff] presents a similar view. Both authors also map orchestrations and choreographies to different use cases: the former is used in the context of single businesses to define and automate its processes, the latter is used to align the processes of multiple corporations (or align processes after corporate mergers). Hollingsworth [2004] refers to this as *internal* and *external* process behavior specification, respectively. In this view, orchestrations reside on a lower level than choreographies, which define interaction points and patterns between (pre-existing) orchestrations.

In practice, classical SOA has favored the orchestration-style definition of processes, as it is well-understood, and there is widespread support for it, mainly based on the BPEL standard. The same is not true for choreographies [Kopp and Leymann, 2008], although there are standardized languages, as well, most notably the *Web Services Choreography Description Language* (WS-CDL; Kavantzas et al. [2005]). According to Decker, Kopp, and Barros [2008], choreographies are not actually executed, but provide a means for standardization across corporate borders, and impose constraints on services and orchestrations that participate in them.

A slightly different perspective towards service choreography sometimes arises in the context of *event-driven* SOA, a variant which highlights publish-subscribe-based interaction patterns to minimize coupling [Levina and Stantchev, 2009; Michelson, 2006]. Although asynchronous messaging is part of many SOA definitions (cmp. Section 7.4), in practice it has often been implemented using remote procedure calls [Richardson and Ruby, 2008, p. 19]. In an event-driven architecture, processes may "arise" rather

than being explicitly defined. As in service choreography, there is no need for central control. In a "classical" approach, services are ignorant of the context in which they are used (which is important for reusability). Instead, in an event-driven approach, services must know which events they should react to, but remain agnostic of how other services react to their actions (events raised). The event-driven approach can be considered one way to realize choreographies [Josuttis, 2007, p. 97, pp. 137ff].

Microservices are very much rooted in this reactive, event-driven approach. Newman [2015, pp. 43ff], however, presents orchestration and choreography not as complementary, but competing, clearly preferring technologically light-weight, event-based service choreographies. Orchestrated systems are critized for being resistant to change, because of a tendency of devolving into a single, massive orchestration service containing the major part of the application logic (and breaking modularity), calling extremly primitive, low-level data manipulation services.

It can be argued that this view on microservices promotes evolution of individual, application-specific, context-aware services very well. This seems appropriate for the development and evolution of software systems, e.g. within a company, with most of the services implemented in-house. With software evolution toolchains, individual services are assumed to be more static (representing standardized, reusable tools). Instead, the toolchain as a whole must be very flexible, i.e. it should be easy to adapt the process. This is hard when there is no central process definition; rather, the services themselves would have to be adapted. For reusable, "off-the-shelf" services this might not even be possible.

## 7.6 Summary

Service orientation has been reviewed in this chapter due to SENSEI's need to address domain experts on an appropriate level of abstraction. In Figure 7.5 (refining Figure III.1, page 97), this corresponds to the left-hand side, where domain experts use SENSEI to specify toolchains. While the understanding of what a service is differs substantially in literature, the definition adopted in this thesis (Definition 7.1) is well-suited for describing functionality in a domain-centric, technology-agnostic manner, while also providing a level of rigor and structure needed to facilitate automatic processing.

Two central requirements that SENSEI aims to satisfy with service-oriented means are *Task Discovery* and *Task Description*. As indicated in Figure 7.5, SENSEI introduces a *service catalog*, that establishes a service description template to formalizes what constitutes a SENSEI service, and that can be filled and browsed by domain experts to create and find services needed for toolchain specification. The service properties presented in Section 7.2 provide the foundation for this. When creating new services, domain experts can look to service design principles as described in Section 7.3 for guidance, while for finding services, the briefly sketched service classification schemes can help organize the catalog.

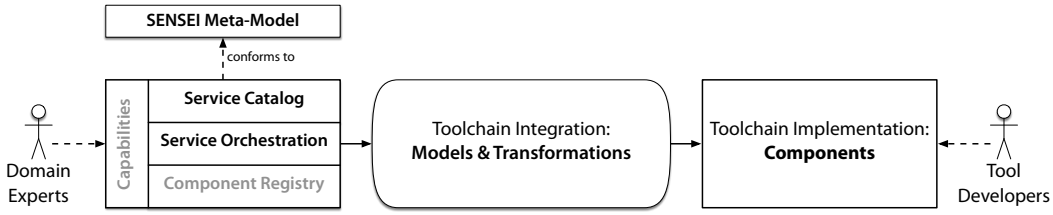



Figure 7.5: Services contributing to SENSEI.

Another pair of requirements considered is for service coordination, namely to model *Data Flow* and *Control Flow*. Service *orchestration* addresses both of these aspects, and goes well with SENSEI's process-centric approach. In contrast to service choreography, orchestrations explicitly manifest the processes, prioritizing flexibility to change processes over changing individual services. Section 7.5 named a couple of existing orchestration languages, none of which were chosen for SENSEI in the end, though, opting instead to create a custom language. Reasons for this design decision are given in Chapter 11, one of which is that SENSEI contains some unique features that would have been cumbersome to incorporate in an existing language.

Service-oriented Architecture envisioned a scheme to dynamically find and bind to services listed in a service registry. This registry addresses a different purpose than SENSEI's service catalog, more in line with the *Tool Discovery* requirement. SENSEI features a *component registry*, which is basically a lookup table to find components for a given service, and is filled by tool developers. Despite this, it is depicted on the left-hand side of Figure 7.5, because it is on the abstraction level of services (only containing references to components, not components themselves), and is part of the overall model used to derive toolchain implementations from. It is grayed out to signify that this concept is less directly derived from service orientation.

Regarding service discovery, the concepts and technologies that sprang from SOA, apart from the general idea, were not adopted by SENSEI, as they have largely failed in practice (see Section 7.4). SENSEI introduces *service capabilities*, meant to control service granularity and enable precise matching between services and implementing components. They represent a kind of connective tissue for its overall architecture and therefore appear in different forms in different places throughout SENSEI – the details of which will be explained in Part IV. Figure 7.5 indicates this role by depicting capabilities crosswise. They are grayed out, as they are unique to SENSEI, and not directly derived from service orientation. One takeaway from SOA is, however, that the service discovery mechanisms (UDDI in particular) were considered too complex. Capabilities expose a simple mechanic to both domain experts and tool developers, to support modeling required service specifics and provided component facilities, respectively.

In SENSEI, services complement components to jointly satisfy the *Separation of Concerns* requirement. This leaves a gap between the two that needs to be bridged to automatically derive component-based toolchain implementations from service-oriented toolchain specifications. To this end, specifications need to be machine-readable, which is why Figure 7.5 depicts that the service-oriented artifacts on the left-hand side of the diagram collectively conform to the SENSEI metamodel. The following chapter will review model-driven software engineering as a foundation for this essential link to complete the SENSEI architecture.

CHAPTER 8

# Model-Driven Software Engineering

A long-standing issue in software engineering is the detachment of documentation and realization of software systems. Documentation is needed both when initially developing software systems, for conception, design, planning, and specification, as well as during the remaining life time of systems, to facilitate maintenance and evolution activities. Developed separately, which incurs double effort, while only the work on the actual system creates an immediate benefit, documentation is often left unsynchronized with the software systems its supposed to describe, rendering it useless.

The core idea of *model-driven software engineering* is to make the *models* of software systems the central artifact of software development and evolution, rather than the code implementing the systems. Furthermore, to avoid the redundancies involved in manually implementing in code what has already been *modeled*, model-driven approaches strive to generate low level artifacts from higher-level models by means of *model transformations*, as far as possible. This leads to the four main benefits of model-driven software engineering according to Kleppe, Warmer, and Bast [2003, pp. 9ff]: besides the *productivity* boost, models become a single point of truth, and are automatically kept up-to-date with the code they *document* (as it is generated from them). Keeping models clean of technical and platform-specific details, by moving them into transformations, model-driven software engineering can also facilitate *portability* (targeting different platforms using appropriate transformations), and, by additionally generating adapters as needed, *interoperability*.

The intention behind model-driven software engineering is often compared to earlier efforts to raise the abstraction level of computer programming, e.g. with the advent of the first compilers, and programming languages above the assembler level. Atkinson and Kühne [2003] say this allowed to program more in terms of *what* the machine should do, rather then explicitly instructing it *how* to do it. And this is true to a degree, as high-level language constructs like loops and subroutine calls leave some room for

the compiler to decide how these will be realized in concrete machine code. However, modern programming languages common today, e.g. Java, which is certainly not considered "close to the metal", are conceptually still very much established in the technical realm of the underlying hardware and software systems.

The "grand vision" of model-driven software engineering is to raise the abstraction level from the solution into the problem domain. This would complement the *small step* of abstraction compilers and high-level programming languages achieved with a *giant leap* over the semantic gap in software engineering.

For SENSEI, model-driven software engineering is expected to provide support for the separation of problem domain and implementation, and for automatic tool integration into toolchains:

**Separation of Concerns.** The support framework must establish a clear separation between toolchain specification and toolchain integration.

**Automatic Coordination.** The support framework must provide automatic tool coordination and toolchain execution in conformance with its specification.

Furthermore, model-driven software engineering facilitates the creation of *domain-specific languages*, which should provide foundational support for all requirements concerned with enabling domain experts or tool developers to express their concerns or needs in a way that is close to their domain and realm of knowledge:

**Task Description.** The support framework must provide domain experts with a means to describe required properties of tasks in a standardized way.

**Data Flow.** The support framework must aid domain experts in specifying the data flow between tasks.

**Control Flow.** The support framework must aid domain experts in specifying the control flow between tasks.

**Tool Description.** The support framework must aid tool developers in specifying which tasks their tools can support.

The outline of this chapter is detailed in Section 8.1, giving an overview of the core concepts of model-driven software engineering that will be introduced in the subsequent sections. A summary is given in Section 8.8.

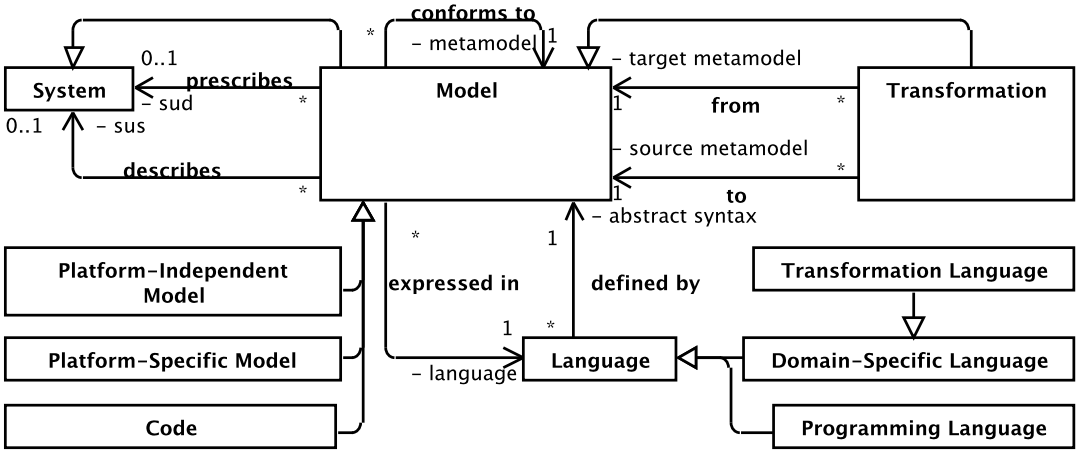



Figure 8.1: Core concepts of model-driven software engineering.

## 8.1 Overview

Figure 8.1 provides an overview over the core terms and concepts to be introduced in this chapter. At its core is the *model*. A definition of the term, and a model's relationship with the system it is representing, is given in Section 8.2. A very central concept is represented in Figure 8.1 only as a self-association on the model class: *metamodels* define *languages* to be used to express models in, and are described in Section 8.3.

With these concepts and terms introduced, Section 8.4 proceeds with presenting the principles and practices of *model-driven development*. In particular, the manifestation standardized by the Object Management Group, *model-driven architecture*, is introduced, characterized by a development process starting with *platform-independent models*, which are refined to *platform-specific models*, and then to *code*. A key part of that are model *transformations* (Section 8.5), to produce new models from existing ones. Another important sub-field of model-driven engineering is the design of *domain-specific languages* (Section 8.6), to enable the expression of (machine-processable) models using the specialist terminology of a particular domain. Finally, Section 8.7 introduces the concept of *technical spaces*, which group compatible implementation standards and development tools that are needed to actually practice model-driven development.

## 8.2 Models

As opposed to other fields of study reviewed in this thesis, the terminology in general seems to be more established and less fuzzy. A characterization of the definition of a model widely accepted in software modeling and model-driven software engineering, is given by Stachowiak [1973, pp. 131-133][1] in terms of three basic properties: *representation*, *reduction*, and *pragmatism*[2].

**Representation.** A model is always related to something in the real world, the *system under study*.

**Reduction.** It provides an incomplete description of it, possibly focussing only on select aspects, and leaving out details, e.g. by means of abstraction.

**Pragmatism** A model always serves a purpose for a certain time or occasion, and addresses certain stakeholders.

While this is certainly a simplified account of the original definition, it defines the concept of a model quite well. This particular rendition also introduces the term *system*, which Stachowiak avoids[3] due to his broader context.

Since an in-depth discussion of what a system is would be inexpedient, as well, leading away from the actual topic of the chapter, a common definition is adopted from systems theory without detailed discussion.

**Definition 8.1: System**

Skyttner [2005, p. 58] summarizes several definitions found in the literature as follows:

> “ [A system is] an organized whole in which parts are related together, which generates emergent properties and has some purpose. ”

An observation is that models are also systems (conceptual or abstract systems in terms of Skyttner [2005, pp. 61f]). The opposite is not true though, as systems have a purpose (pragmatism), but need not represent and reduce another system.

The initial, informal notion to think of a system as “something in the real world” is thus not entirely consistent, but serves as a guideline when to use the term system, and when to use model. In the context of (model-driven) software engineering, the real-world systems often are software, but may, for example, also refer to businesses or software evolution processes.

[1] This is a work in philosophy, mathematical logic, and cybernetics. It predates model-driven software engineering, and probably any common practice of software modeling for that matter, by several decades.

[2] This is rather freely translated from the original german “Abbildungsmerkmal”, “Verkürzungsmerkmal”, and “Pragmatisches Merkmal”, respectively.

[3] Also pointed out by Hesse and Mayr [2008]; the more generic term “original” is used instead.

**Definition 8.2: Model**

A model represents a part of a real system, reduced to serve a particular purpose.

There are two further additions to the definition of the term model. The first, owed to Kleppe, Warmer, and Bast [2003, p. 16], is that it is always expressed in a *language*. The second is that a (well-defined) model always *conforms to a metamodel*. The concepts of both languages and metamodels will be explored in Section 8.3, to extend the definition of "model" appropriately.

Models can be classified according to a number of criteria:

- Models are either *descriptive*, i.e. they have been derived from a *system under study* (SUS), or they are *prescriptive*, which means a *system under development* (SUD) is meant to be derived from them [Aßmann, Zschaler, and Wagner, 2006; Seidewitz, 2003]. According to Hesse and Mayr [2008], models can also exhibit both properties, i.e. be derived from an original system to then be used to derive a new system from it, which is reflected in Figure 8.1.
- Models are either *static* or *dynamic*: Static models contain only time-invariant aspects of the original system, constraining its structure, or describe a single state of a system frozen in time. They are expressed in terms of a system's elements and their interrelationships. Dynamic models describe rules for the behavior of their associated systems, either viewed in terms of the processes that can take place, or how the system changes states as reaction to external or internal stimuli.
- Models describe aspects of systems either as *tokens* or as *types* [Hesse and Mayr, 2008; Kühne, 2006; Peirce, 1906]: Token models match each element within the covered area of the corresponding system with a corresponding model element (token). Type models abstract from a system by grouping its elements (*instances*) according to certain characteristics and thereby establish a common model element for all of them (types).

A kind of model fundamental to model-driven software engineering are prescriptive static type models, a category which gives rise to the concepts of *metamodels* and *metamodeling*.

## 8.3 Metamodels

As expressed in Figure 8.1, a model is always expressed in a language. In the context of software engineering, the languages used should follow well-defined rules, so that the models are universally understood in the same way. To be useful in *model-driven* software engineering, models must be expressed in machine-processable languages.

**Definition 8.3: Language**

A *well-defined language*, i.e. one that can be interpreted algorithmically, is defined by [Kleppe, Warmer, and Bast, 2003, p. 16; Kelly and Tolvanen, 2008, p. 68]:

- its *abstract syntax* (the concepts of the language, and their interrelationships),
- its *concrete syntax* (the symbols used to depict these concepts to form expressions in the language), and
- its *semantics* (the meaning conveyed by the language's concepts and expressions).

For model-driven techniques, the abstract syntax is of principal importance (the importance of concrete syntax and semantics of languages is discussed, for example, by Kleppe [2009]). The abstract syntax of textual languages (e.g. programming languages like Java) can be formalized by grammars. Such grammars can themselves be considered type models. For example, a grammar of the Java programming language would be a model of all programs expressed in Java. Generalized to also include visual models and modeling languages, this gives rise to the concept of *metamodels*.

A metamodel is still just a model, not an entirely new concept, as reflected by Figure 8.1, where the term only appears as a role label. Metamodels provide the abstract syntax of languages used to express models. Figure 8.1 shows that a language is *defined by* a model, which makes it a metamodel by definition. Models expressed in this language are said to *conform to* this metamodel.

**Definition 8.4: Metamodel**

A metamodel is a static prescriptive type model used to define the abstract syntax of a language.

When considering a metamodel as "just another model", it must follow that it too conforms to a metamodel – depending on the point of view, also referred to as *meta-metamodel*. Followed through, this would lead to an infinite sequence of models and their meta-(meta-meta-...)models. In practice, it can be capped at four levels as follows (cmp. Figure 8.2):

**$M_0$** This is the bottom layer, where *systems of the real world* are situated.

**$M_1$** At this layer, *models* are formed describing the systems below.

**$M_2$** The *metamodels* at this layer define the languages used to express models at the layer below.

**$M_3$** *Meta-metamodels* in turn define languages to express metamodels in. Meta-metamodels can be made to *conform to themselves*, which is why no further levels are needed.

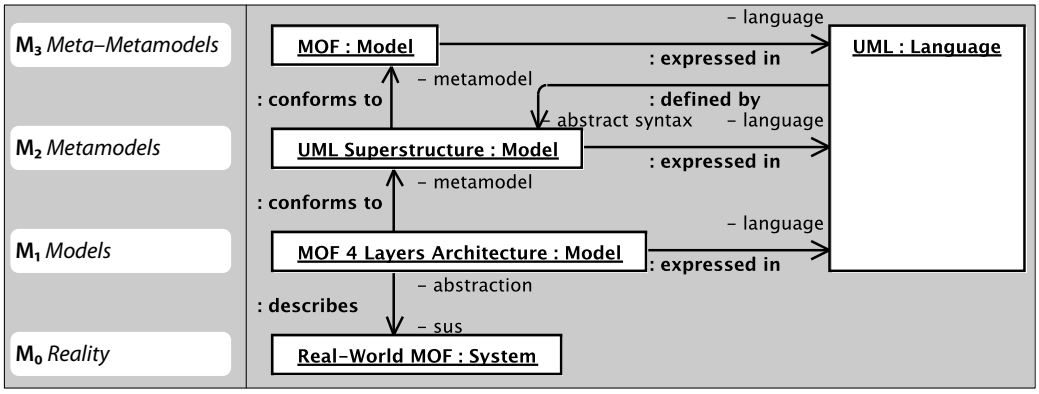



Figure 8.2: The basic, four-layered framework of MOF, depicted as object diagram conforming to the model shown in Figure 8.1.

This four level structure of meta-modeling is standardized as part of *MOF*, the *Meta Object Facility* [2013], which constitutes a $M_3$ meta-metamodel, and thus a metamodeling framework, which in turn is the basis for OMG's model-driven architecture (see Section 8.4). Figure 8.2 depicts these four levels as object diagram – a model which conforms to the (meta-)model that was used to introduce the basic terminology, the class diagram shown in Figure 8.1. Because it is a model of MOF, the figure is *highly* self-referential: at the $M_0$ level, there is the system *Real-World MOF*. Above that, a model of the *MOF 4 Layers Architecture*; this element in the figure therefore refers to the model depicted in the figure as a whole! The language used in this figure is that of UML object diagrams. This language is defined on level $M_2$ in the *UML Superstructure*[4]. Finally, at the meta-meta-level $M_3$, there is *MOF*.

It is a peculiarity of MOF that all modeling levels are expressed in the same language, UML. Moreover, this language is not really defined at the highest level, but rather at $M_2$. MOF is expressed in a subset of UML (basically, class diagrams), at least this is how the standards set it up – possibly a historical remnant due to the fact that the original UML actually came before MOF was standardized. An equivalent view would be to assume that MOF defines the required UML subset, which is then used to redefine itself as part of UML.

The strict four-layered design of MOF is not without critics, as it can pose problems when using a metamodel to create a model, which itself should contain concepts that need to be instantiated. As a metamodel user, one can only operate within the $M_1$ level, leading to the seemingly paradoxical situation (e.g., see Stahl et al. [2006, pp. 116f]) of

[4]As of UML 2.5, the distinction between infrastructure and superstructure was dropped.

model elements conforming to more than one (meta-)class. Atkinson and Kühne [2003] propose to distinguish between *linguistic* metamodeling, which dictates the modeling languages, and *ontological* metamodeling, an orthogonal metamodeling dimension to describe class-instance relationships within the same linguistic layer. This establishes awareness of the underlying issues, but does not solve them – the strict metamodeling hierarchy established through MOF constitutes real modeling limitations [Atkinson and Kühne, 2001]. Further solution approaches are presented by Goldstein and Storey [1994], Gonzalez-Perez and Henderson-Sellers [2008], Laarman and Kurtev [2010], and Neumayr, Grün, and Schrefl [2009].

In the context of model-driven development, models defined by means of metamodeling are a fundamental prerequisite, so they can be processed automatically. Specifically, it facilitates *transformations* between models conforming to different metamodels, by expressing transformation rules in terms of the concepts defined by the metamodels. The concept of metamodeling is further discussed in the following section, with regard to their place in the model-driven software development process.

## 8.4 Model-Driven Development

There are several terms all beginning with "model-driven", all coming with their own abbreviation. While some authors use them interchangeably, Brambilla, Cabot, and Wimmer [2012, p. 9] offer the following distinction:

**MDE** *Model-driven (software) engineering* is an umbrella term for model-driven techniques applied to different areas of software engineering, for example *model-driven software migration* (e.g., see Wagner [2014, p. 47]).

**MDD** *Model-driven (software) development* is a subset of MDE, focused on supporting and partly automating the forward engineering path of development activities.

**MDA** *Model-driven architecture* is a particular flavor of MDD. Based on UML, it is a standardization effort by the [*Model Driven Architecture* 2020], and is sometimes cited to have a narrower set of objectives than general MDD, focusing in particular on interoperability and portability [Stahl et al., 2006, pp. 4, 14].

Collectively, they are referred to as *MD**. Brambilla, Cabot, and Wimmer include another surrounding shell in this taxonomy, called *model-based engineering*, which they describe as weaker than model-*driven* approaches: in model-based engineering, the model is an important artifact, but not necessarily the central one. More importantly, models may not be subject to automatic processing. Stahl et al. [2006, p. 3] agree, but Pastor and Molina [2007, p. 41] seem to argue the opposite. In their view, a *model-driven* process is characterized by a phase of modeling, followed by a phase of (manual) implementation, possibly based on code skeletons generated from the models, whereas in their *model-based* approach, the system is fully specified in problem domain terminology, in what they call a *conceptual schema*, which is used to gener-

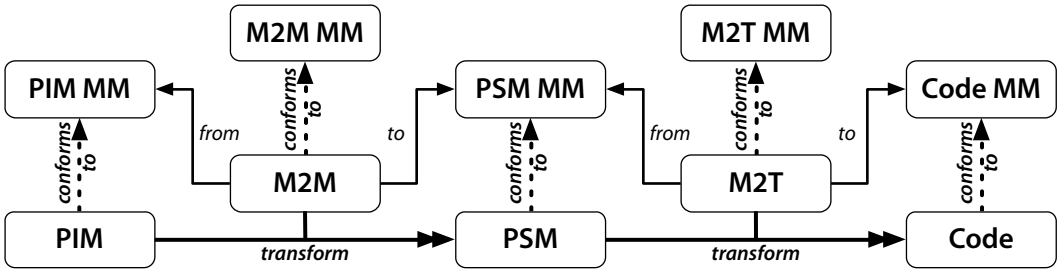



Figure 8.3: The basic model-driven development process.

ate resulting software systems without a need for further manual work. It gets even more confusing if neighboring disciplines are considered, e.g. the embedded systems community talks of *model-based design* [Karsai et al., 2003], which seems to mostly correspond to the above definition of MDE. Therefore, in this thesis, the term "model-based" is avoided.

Although this section describes MDD, the terminology used is that of OMG-standardized MDA, which has caught on in the field as a whole [Stahl et al., 2006, p. 11]. Figure 8.3 depicts the three main artifacts of MDD at the bottom (similar figures can be found in Kleppe, Warmer, and Bast [2003, pp. 8, 26] and Zeppenfeld and Wolters [2005, p. 64]): the *platform-independent model* (PIM), the *platform-specific model* (PSM), and *code*, sometimes referred to as *implementation-specific model* (ISM).

**PIM** The platform-independent model describes a system in generic terms, i.e. without referring to a particular platform like hardware, operating systems, virtual machines, programming languages, middleware, libraries, or any other implementation techniques. Since in MDA (as opposed to MDD in general), the PIM is expected to be expressed in terms of UML, there may be a bias towards object-orientation. This can be avoided by using specialized languages (see domain-specific modeling in Section 8.6), which UML enables to create through stereotypes and profiles.

**PSM** The platform-specific model is ideally derived fully automatically from the PIM, by mapping its generic concepts to concrete ones from a selected platforms. As indicated above, there can be a whole stack of platforms. PSMs can therefore also be refined iteratively, by mapping different aspects to different selected platform technologies [Stahl et al., 2006, p. 16f].

**Code** The final PSM is expected to be on an abstraction level very close to the actual implementation, so that code can be generated in a straight-forward manner. In ideal MDD, the code represents the complete software system as specified in the PIM, without need for further manual programming work.

Relating these artifacts to SENSEI, it seeks to establish a platform-independent metamodel for the specification of toolchains, expressed in the languages of services. A component framework serves as target platform, thereby dictating the platform-specific and code metamodels. What is needed, both by SENSEI, as well as in general, is a means to encode the mapping of platform-independent concepts to platform-specific ones, so it can happen automatically.

MDD uses *transformations* for this purpose, in particular *model-to-model* (M2M) transformations. A transformation can be thought of as a special program, though more generally, it is also a model, expressed in a transformation language (a more detailed definition will be given in Section 8.5). Of course, transformations have to be written by someone, first. The benefit of this approach is derived from the fact that it is written independently of any particular PIM, so it only has to be written *once*, and then its mapping rules can be reused to map arbitrary other PIMs to the same platform(s). This is where metamodeling becomes essential: to be able to specify such mapping rules in transformations, the concepts, and their interrelations, of both the platform-independent as well as the platform-specific domain have to be known. Those are formalized in their metamodels – again, this has to be done only once, to then specify arbitrary many systems as conforming models. If an existing language and metamodel is used, such as the UML, this step is not needed at all.

The transformation from PSM to code happens in much the same way. Only now, the language of the target model will be text-based, which makes a difference at a technical level, i.e. different kind of transformation strategies, languages, and tools will be used. This transformation step is thus called *model-to-text* (M2T).

A fourth artifact is sometimes introduced [Zeppenfeld and Wolters, 2005, p. 63], representing an abstraction level above the PIM: the *computation-independent model* (CIM). The Object Management Group equates this with a *domain model* [Siegel, 2014], i.e. a *descriptive* model of the problem domain, while also implying that the term as such is obsolete. The automatic derivation of software systems starting from this high level of abstraction therefore does not seem to be among the ambitions of MDA anymore, if it ever was; Guttman and Parodi [2006, pp. 54, 68] report on practical issues and lack of MDA tool support for CIM-to-PIM transformation. After briefly introducing them, Zeppenfeld and Wolters [2005, p. 63] dismiss CIMs as inconsequential. Kleppe, Warmer, and Bast [2003, p. 19] even argue that it is impossible to derive PIMs from CIMs automatically, as it involves deciding which parts of a business (modeled in the CIM) should be supported by the software system to be developed. In contrast, Stahl et al. [2006, p. 16] define PIMs as "domain-related specifications". Approaches in the realm of MDE less close to MDA, like domain-specific modeling as defined by Kelly and Tolvanen [2008, pp. 47-49], and the OO-method by Pastor and Molina [2007, p. 8], clearly aspire to bridge the semantic gap between problem domain and solution domain. This thesis adopts the view that PIMs are part of the problem domain, aligning them with the service-oriented specification side of SENSEI.

## 8.5 Transformations

Transformations are a key ingredient in model-driven development, used to map PIMs to PSMs and PSMs to code. The term has already been used, but has not been formally introduced. Therefore, a common definition from literature is adopted:

**Definition 8.5: Model Transformation**

Kleppe, Warmer, and Bast [2003, pp. 23ff] distinguish between *transformation*, *transformation rule*, *transformation definition*, and *transformation tool* as follows:

- *Transformation* refers to the act of producing one model out of another.
- *Transformation rules* specify how individual elements of the original model affect the generation of elements and aspects of the resulting model.
- *Transformation definitions* are aggregations of transformation rules to fully specify the transformation process of models conforming to the source metamodel into models conforming to the target metamodel. As such, transformations can also be understood in terms of (mathematical) *functions* [Mens and Van Gorp, 2006].
- *Transformation tools* take a model to be transformed, and a compatible transformation definition, to then execute the transformation process and output the resulting new model.

Previously, when referring to transformations, what was usually meant (according to these definitions), were transformation definitions. This distinction is usually obvious from the context, so that "transformation" will continued to be used. Another important concept is that of *transformation language* (which Kleppe, Warmer, and Bast [2003, p. 95] consequently term "transformation *definition* language"), used to express transformations in. Considering that transformations are models, transformation languages are implied by the concepts already introduced, i.e. every model needs a metamodel, which also constitutes a language to express the model in (see Figure 8.1).

To get a better understanding of transformations, and to be able to distinguish and decide between different transformation approaches, they can be categorized in various ways. For a comprehensive overview, the reader is referred to Czarnecki and Helsen [2003, 2006], who offer a detailed taxonomy based on feature models. In the context of this thesis, a relevant distinction is into *model-to-model* (M2M) and *model-to-code* or *model-to-text*[5] (M2T) transformations. This might seem artificial, since in model-driven engineering, code should be considered to be "just another model" (as expressed in

[5]Czarnecki and Helsen [2003] note that "model-to-text" is the more appropriate term, yet still they adopt the latter terminology.

Figure 8.1). In fact, transforming from PSM to code, only for a compiler to parse that code into an abstract syntax tree (which is a PSM again), seems particularly wasteful. However, taking this indirection allows to remain using existing compilers [Czarnecki and Helsen, 2003]. Technically, M2T transformations are usually untyped on the target model side, as there is no target metamodel to refer to and check against.

Mens and Van Gorp [2006] also provide a taxonomy of model transformations, which is broader in scope, but less detailed than those by Czarnecki and Helsen.

A plethora of model transformation tools exist, some of which will be mentioned in Section 8.7. Czarnecki and Helsen [2006] compares existing tools within their feature model-based classification framework. A recent, comprehensive overview and comparison of transformation tools is given by Jakumeit et al. [2014].

## 8.6 Domain-Specific Languages and Modeling

*Domain-specific languages* (DSLs) and *domain-specific modeling* (DSM) are fields that predate the standardization efforts towards MDA, and have initially not been a part of the agenda [Jouault, Bézivin, and Barbero, 2009]. According to the recently updated, official documentation of the Object Management Group [Siegel, 2014], they still are not. In the research communities, and under the more general MDE moniker, however, a shift towards domain-specific techniques, and a convergence of the fields can be observed [Kurtev et al., 2006; Jouault, Bézivin, and Barbero, 2009; Kelly and Tolvanen, 2008, pp. 6ff; see also Cook, 2004]. Early work on domain-specific languages was performed by Bentley [1986], referring to them as *little languages*. A comprehensive overview over the research field at the turn of the millennium (and thus predating MDA) is provided by Deursen, Klint, and Visser [2000].

Domain-specific languages address the previously raised question regarding the abstraction level of the high-level models in a model-driven engineering setting, i.e. the PIMs (or possibly CIMs). As the name suggests, DSM focuses on the *problem domain* [Czarnecki, 2004; Deursen, Klint, and Visser, 2000], whereas "official" MDA seems to endorse a view that sees PIMs on a high abstraction level, but described in technical, solution space terminology, using the various sub-languages of UML.

**Definition 8.6: Domain-Specific Language**

> "[A] *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."
>
> Deursen, Klint, and Visser [2000]

The key difference between DSLs and general-purpose modeling or programming languages (GPL) is their scope: a GPL like, for example, Java, can be used to describe and realize basically any arbitrary kind of application, but it has to be done in the technical terms of classes, methods, variables, loops, and so on. DSLs, in contrast, are intentionally designed with a (often significantly) narrowed scope, e.g. to describe order processing for a stock market system [Ghosh, 2010], query databases (SQL), or describe hyperlinked documents (HTML).

While these languages are not as widely applicable as GPLs, they allow to model aspects of the problem domain in its inherent terms and concepts: users of a DSL will ideally "perceive themselves as working directly with domain concepts" [Sprinkle et al., 2009]. Of course, a mapping onto concepts of the solution domain has still to occur, but with DSLs this can be factored out into transformations, i.e. it is done once, and then automated and reused in any further development projects in the same domain. In industrial practice, MDE seems to be adopted mostly for such narrow, specialized application areas, using tailor-made DSLs, as opposed to building complete applications in a model-driven approach [Whittle, Hutchinson, and Rouncefield, 2014].

In SENSEI, individual domain-specific languages may address the needs of its different user roles and their tasks in the toolchain-building process, corresponding to the requirements for *Task Description* and *Data Flow*, as well as for *Control Flow* and *Tool Description* specification.

Domain-specific languages can be created in a variety of ways. In the MDA framework, DSLs can be defined by means of UML profiles, which extends UML with domain-specific concepts by mapping them onto existing, generic ones. This can be contrasted with using special tools and environments for DSL engineering [Abouzahra et al., 2005], referred to as *language workbenches* [Erdweg et al., 2013; Fowler, 2005a]. Fowler [2010] further distinguishes between *internal* and *external* DSLs: internal DSLs are created inside of a host GPL. Many modern programming languages offer some form of meta-programming to describe a new language within a language. Ghosh [2010, pp. 128ff], for example, uses Ruby, Groovy, Clojure, and Scala to build internal DSLs. External DSLs are created, for example, using classical parser generators, or the aforementioned language workbenches.

Czarnecki [2004] discusses DSLs in the context of generative software *product-line engineering* (called "system-family engineering" here; see also Clements and Northrop [2002]), which aims to manage *variability* in the feature sets of software product families by explicitly modeling those features (in terms of appropriate DSLs), to then configure and generate different software systems from those descriptions. MD* is delimitated from generative software development by emphasizing the focus on system families of the latter. The mapping from problem to solution space is highlighted as key concept of generative software development, and likened to the transformations of PIMs into PSMs in MDA.

Expressing variability can be of interest with respect to SENSEI's requirements for *Tool Discovery* and *Tool Description*. These demand support for the description of required properties to properly support a task, and of provided properties of individual tools, respectively. Though the domain is very different, properties of (generic) software evolution techniques and tools can be likened to features of software product lines, i.e. the underlying mechanics are, to a certain extent, similar. A DSL to express variability in product lines is constituted by *feature diagrams*, which are part of *feature-oriented domain analysis* (FODA, Kang et al. [1990] and Kang, Lee, and Donohoe [2002]).

## 8.7 Technical Spaces

Model-driven engineering depends on proper development tool support for modeling, meta-modeling and language design, as well as to define and execute transformations. There are a variety of tools and solutions to choose from, but to interoperate, all tools should be chosen from the same *technical space*. The concept of technical spaces has been introduced by Kurtev, Bézivin, and Aksit [2002][6], to compare MDE with similar technical frameworks, such as XML and relational databases, and discuss common concepts on a generic, abstract level.

**Definition 8.7: Technical Space**

Kurtev, Bézivin, and Aksit [2002] state:

> "[a technical] space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities."

Bézivin and Kurtev [2006] add:

> "A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework."

Furthermore, technical spaces are characterized by exhibiting a three-level arrangement of meta-metamodel, metamodels, and models [Bézivin, 2006]. Generically, all three levels can be considered and described in terms of graphs.

The determining factor for the scope and boundaries of technical spaces are its meta-metamodels. A technical space of programming languages can be defined by means of EBNF [Bézivin, 2006; Klint, Lämmel, and Verhoef, 2005], which resides on the $M_3$ level, i.e. constitutes meta-metamodel. For XML, it would be the *schema-schema* that is part of the XML Schema standard [Thompson et al., 2004], defining its language in its own terms (as a $M_3$ level model, it conforms to itself).

[6] Here, they are referred to as *technological* spaces, but later publications change this, e.g. Bézivin [2006] and Bézivin and Kurtev [2006]

While SENSEI is a conceptual framework that can be implemented in any technical space, for its toolchain generator SCAffolder a design decision had to be made, which will be discussed in Section 14.2.1. Examples of technical spaces include:

- The **OMG technical space** based on MOF.
- The closely related, but separate [Bézivin and Kurtev, 2006] **Eclipse Modeling Framework (EMF) technical space** [Steinberg et al., 2008], based on a MOF-subset called *Ecore*. This is probably also the most widely used technical space.
- The **Microsoft DSL Tools technical space** [Microsoft, 2016], which supports the *software factories* approach [Greenfield and Short, 2004].
- The **TGraph technical space**, based on strong theoretical foundations and continuous research [Ebert, 1987, 2008].

The EMF technical space provides a plethora of tools, including, for example, the *Graphical Editing Framework* (GEF; *Graphical Editing Framework* [2020]), the *Graphical Modeling Framework* (GMF; *Graphical Modeling Framework* [2020]), *Graphiti* [*Graphiti* 2020], *EuGENia* [Kolovos et al., 2010], and *Sirius* [*Sirius* 2020], all graphical DSL tools, *Xtext* [Eysholdt and Behrens, 2010] for textual DSL creation, as well as the transformation languages ATL (*Atlas Transformation Language*, Jouault et al. [2006]), VIATRA, and *eMOFLON* [Hildebrandt et al., 2013]. The TGraph technical space, implemented in *JGraLab*, has a bridge into the EMF technical space. With *grUML*, a MOF-subset, it is possible to author TGraph metamodels using standard UML tools. JGraLab provides a Java TGraph API and code generation facilities, as well as support for model querying and transformation using GReQL and GReTL (the *Graph Repository Query/Transformation Languages*), respectively.

Several other tools exist that do not belong in any of the technical spaces mentioned so far, yet their comparatively smaller user community, lower visibility, and sometimes more limited scope means that they do not really span a technical space of their own, in conformance with Definition 8.7. To support a holistic model-driven development process, tooling is required for modeling, meta-modeling, and language engineering (unless a general-purpose modeling language such as UML is to be used), as well as for the creation and execution of transformations. With tools that provide only partial support, and are not part of a larger technical space, interoperability issues are to be expected.

Noteworthy tools include MetaEdit+ [Kelly, Lyytinen, and Rossi, 1996], the *Generic Modeling Environment* [Ledeczi et al., 2001] for the creation of graphical DSLs, and for the creation of textual DSLs there are, for example, *JetBrains MPS* [Pech, Shatalin, and Völter, 2013], *Spoofax* [Kats and Visser, 2010], and *Rascal* [Klint, Storm, and Vinju, 2011], the latter the successor of the *ASF+SDF Meta Environment* [Brand et al., 2001].

Most of these tools do not support the whole model-driven development process, providing only language engineering facilities, but no transformations. Apart from the transformation languages already mentioned, there are several more that do not fall in

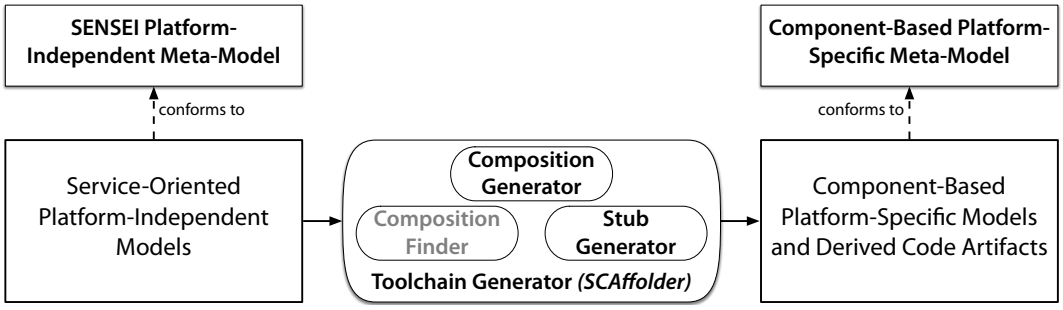



Figure 8.4: Model-driven concepts contributing to SENSEI.

any of the technical spaces presented here, for example *DMS* [Baxter, Pidgeon, and Mehlich, 2004], *StrategoXT* [Visser, 2004], and TXL [Cordy, 2004]. Again, these tools are originally aimed at software evolution, but can be used in model-driven and generative software development, as well [Czarnecki, 2004].

Overviews and comparisons of these and further tools are provided by Amyot, Farah, and Roy [2006], Czarnecki [2004], de Sousa Saraiva and Rodrigues da Silva [2008], Erdweg et al. [2013], and Jakumeit et al. [2014].

## 8.8 Summary

After having investigated service-oriented and component-based principles as the basis for specifying and realizing integrated toolchains, respectively, model-driven software engineering was reviewed to fill the gap between the two. Figure 8.4 once again recalls Figure III.1 (page 97), and refines it to express SENSEI's main building blocks in model-driven terminology, and locate the application of model-driven techniques in SENSEI.

In terms of the model-driven development process (Section 8.4), service-oriented specifications of SENSEI become *platform-independent models*, and component-based implementations become *platform-specific models*. Bridging the two are *model transformations* (Section 8.5). The model-driven frame ensures both a clear delimitation as demanded by the *Separation of Concerns* requirement, as well as satisfying the *Automatic Coordination* requirement by facilitating the mapping from toolchain specifications into concrete implementations, and automatically generating all the necessary artifacts.

SENSEI supports both domain experts and tool developers by providing two kinds of operations: the *Composition Finder* produces integrated toolchains from corresponding specifications. The *Stub Generator* allows tool developers to take descriptions of services to be implemented in a new tool (or to adapt an existing one to SENSEI), and create the necessary boilerplate code. Both are prototypically implemented by SCAffolder,

as a phase of model-to-model transformations followed by a model-to-text transformation phase, and relying on the TGraph technical space (Section 8.7).

One aspect of toolchain implementation is not directly covered by model-driven technology, namely the need to automatically map services to appropriate components (*Tool Discovery* requirement). Partly, this can be solved statically, but SENSEI offers domain experts more control over particular properties needed of services during specification, while also shielding them from implementation details, so that they do not have to figure out which concrete components work well with each other, have the necessary adapters and transformers, etc. To automatically find a set of components that satisfies all such constraints for a given toolchain specification, SENSEI provides the *Composition Finder* (shown grayed out in Figure 8.4), which pre-processes specifications and augments them with information about which concrete component to use for each service. This process is described in Chapter 12.

In addition to its role of linking toolchain specification to implementation, model-driven software engineering provides technical underpinnings on both sides, as the model transformations require them to be described in terms of appropriate metamodels. On the specification side, modeling plays a particular important role in defining the *languages* (Section 8.6) to express services and their orchestrations in terms suitable for domain experts. In that regard, model-driven software engineering also supports the requirements for *Task Description*, *Data Flow*, *Control Flow*, and *Tool Description*.

# PART IV

# Solution

So far, this thesis has introduced the challenges of toolchain-building and integration, and has defined the core objectives of the work (Part I). These were then refined, and delineated against existing approaches, painting a comprehensive picture of the problem domain and clearly placing SENSEI within it (Part II). Next, guided by a first high-level vision of what the approach should look like, software engineering paradigms were reviewed for design principles and concepts to serve as the technological foundations of SENSEI (Part III). Now, all the pieces are available and ready to be put together to present the solution that has been developed for this thesis.

SENSEI is a conceptual framework for *Software EvolutioN SErvices Integration*. It stands on the pillars of component-based, service-oriented, and model-driven software engineering techniques, with the separation of specification and implementation as central guiding principle. The following chapters describe all aspects of SENSEI in detail, covering all its conceptual layers, as well as the design and implementation of tools to support and partly automate the toolchain-building process.

Chapter 9 gives a brief, high-level overview of SENSEI, introducing its basic concepts and principles, and describing how it is used to support the toolchain-building process.

SENSEI is built around a central metamodel, which is divided into three layers: Chapter 10 describes the *service catalog*, used to define functional building blocks of software evolution tasks and activities. Chapter 11 describes how these building blocks are combined to describe processes in the form of *service orchestrations*. Chapter 12 introduces the *component registry*, which provides a bridge between toolchain specification and implementation, and describes how this bridge can be traversed by tools to automatically generate integrated toolchains.

In addition, the SENSEI approach towards toolchain-building is supported by two kinds of tools, for each of which an implementation was created. First, the integrated SENSEI *Editors* are presented in Chapter 13, used to create SENSEI models, i.e. assist domain experts and tool developers in modeling toolchain behavior and registering tools for use within the SENSEI framework, respectively. Then, there are tools which process such models to automatically derive fully integrated toolchains; one such tool is SCAffolder, which is described in Chapter 14, along with additional, related and alternative tools. Using these tools, SENSEI has been successfully applied in practice, as will be described in Part V.

CHAPTER 9

# SENSEI at a Glance

SENSEI provides a reference model for *software evolution toolchain-building support frameworks*. It is designed to increase the **flexibility** of toolchains, the **reusability** of its parts and integrating logic, and therefore also yields increased **productivity** in software evolution projects, as per the objectives of this thesis (see Section 1.2). Thus, implementations of SENSEI are intended for use by domain experts, to assist them in performing some steps of the toolchain-building process (Section 3.1), and relieve them of having to perform other steps, altogether, by automating them.

The requirements elicited in Chapter 3 are met by SENSEI using approaches and techniques from *component-based* (Chapter 6), *service-oriented* (Chapter 7), and *model-driven software engineering* (Chapter 8), combined with novel concepts. These key technologies that SENSEI is based on have been introduced in the previous chapters guided by a rudimentary, high-level sketch of the target architecture, and the contributions of each software engineering paradigm have been identified and placed within this picture. This chapter takes all the individual parts and presents a consolidated, broad overview of the central concepts of SENSEI from different angles: Section 9.1 provides a high-level view of the overall architecture, Section 9.2 introduces the SENSEI metamodel, Section 9.3 summarizes SENSEI's concept of *capabilities*, and Section 9.4 recalls the toolchain-building process and maps its steps to the responsible roles and SENSEI's supporting tools. A summary is provided by Section 9.5.

All these aspects will be described further and in more detail in the subsequent chapters, providing closer examinations of SENSEI's *service catalog* (Chapter 10), its *orchestrations* (Chapter 11), and its process of mapping between services and components (Chapter 12). Note that SENSEI does not constitute a software framework itself; it can rather be considered to provide "blueprints" in terms of a reference architecture that largely, and intentionally, leaves mappings to particular technical spaces or implementation technologies open. Practicability is demonstrated by concrete realizations of

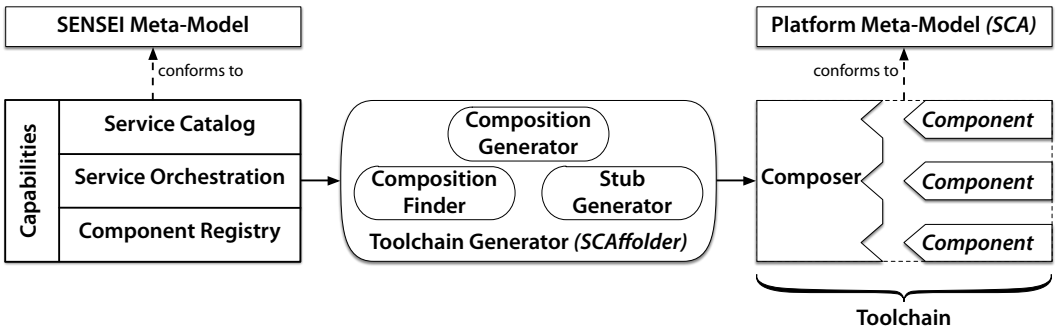



Figure 9.1: The basic SENSEI architecture.

SENSEI's core concepts. These implementations are presented in Chapter 13 (integrated editors for toolchain specification) and Chapter 14 (toolchain generator SCAffolder, and *Composition Finder*), respectively.

## 9.1 The SENSEI Architecture

Figure 9.1 shows a view of the SENSEI architecture that combines the parts that have been introduced throughout Part III. The following can be thought of as a summary of Section 6.5, Section 7.6, and Section 8.8, and provides a brief overview of all the essential parts that form the overall approach. The aim of this section is mainly to introduce or recall terminology and concepts – to give a high-level, static view of SENSEI. Complementing this is a dynamic view presented in Section 9.4, that demonstrates how SENSEI is actually used. In-depth details are provided in the subsequent chapters, each focusing on individual parts of SENSEI.

As previously depicted in Figure 7.5 (page 127), the left-hand side of Figure 9.1 contains the service-oriented aspects. These are also the main *artifacts* that are created by users of SENSEI: The **service catalog** holds software evolution services to choose from. **Service orchestrations** are created by domain experts using the catalog. The **component registry** is filled by tool developers with information about components and their services – again, using the catalog as a reference. While the service standardization facilitated by the catalog enforces a basic level of consistency, using **capabilities** on all levels allows for *concise, yet detailed* service descriptions (in the catalog), and automated matching of services with *required* capabilities (in orchestrations) to implementing components with *provided* capabilities (in the registry). Capabilities form an orthogonal, supportive mechanism, but have no meaning on their own. Conversely, the three horizontal layers each constitute an *artifact* that needs to be created – using appropriate editing tools like the **SENSEI editor**. Their structure, i.e. the abstract syntax

of their description languages, is defined in the **SENSEI metamodel**, an overview of which will be given in Section 9.2.

The center of Figure 9.1 is occupied by the model-driven **toolchain generator**, recalling Figure 8.4 (page 144). This is the automation machinery that takes a service catalog, and a service orchestration and component registry based on it, as input, and outputs a **composer**: a fully auto-generated piece of software that implements the behavior specified by the orchestration by coordinating components that implement the orchestrated services. SENSEI, as a conceptual framework, does not dictate any particular technology for the implementation of the toolchain generator (SCAffolder represents one possible instance), but rather demands *what* functionality it must offer. It is decomposed into three functional subsections, referred to as **processors**[1]: The **Composition Finder** and **Composition Generator** must resolve orchestrated services to implementing components, and map the modeled data and control flows onto corresponding code for a particular platform, respectively. These are supportive of the toolchain-building process as performed by domain experts, while the **Stub Generator** helps tool developers to create service-implementing components by generating boilerplate code and adapter interfaces.

Generated composers only contain the data routing and coordination logic necessary to actualize the process defined by the service orchestration it was derived from. They reference **components**, as depicted on the right-hand side of Figure 9.1, and first introduced in Figure 6.2 (page 107). Components implement orchestrated services to perform the actual computations. Their component model, and the component framework they are running on, is determined by, and bound to, a concrete toolchain generator implementation. The whole target platform must be described formally in a **platform metamodel** as a prerequisite for utilizing model-driven transformations. SCAffolder targets the *service component architecture* (SCA), and defines a corresponding target metamodel. Other implementations might choose techniques that do not make the target metamodel explicit. In fact, instead of a toolchain generator, an *interpreter* able to execute orchestrations directly might provide the same functionality. This alternative approach is briefly discussed in Section 14.5; otherwise, SENSEI assumes a model-driven, generative approach is used.

The most explicit support of the toolchain-building process is provided by SENSEI's editors (Chapter 13) and the toolchain generator (Chapter 14), as these are actual assistive software tools. To differentiate them from software evolution tools used as parts of toolchains, they are referred to as *meta-tools*, indicating the fact that they are tools used in the creation of tools and toolchains.

Before getting into actually using SENSEI and its meta-tools to build toolchains in Section 9.4, two aspects of the SENSEI architecture are given the spotlight, first: Section 9.2 zooms into the left-hand side of Figure 9.1 to provide an overall view of the

[1]The functionality could also be expressed in terms of a service orchestration, but such a self-referential notion might lead to confusion at this juncture.

SENSEI metamodel; the subsequent chapters will fill in more details into its respective layers. Section 9.3 summarizes the need for *capabilities*, and describes the different roles they play within SENSEI.

## 9.2 The SENSEI Metamodel

To leverage model-driven techniques, the main user-created artifacts of SENSEI are represented by models conforming to metamodels. A centerpiece of SENSEI is its integrated metamodel, whose main concepts are illustrated in simplified form in Figure 9.2, consisting of three main layers, each corresponding to one of SENSEI's central artifacts. Integrating these three layers are *capabilities* – these will be described in Section 9.3.

The top layer of the metamodel corresponds to the service catalog, defining the structure of its contents. The central concept is the *service*, which has a name and a description, as well as arbitrarily many input and output parameters. Apart from services, the catalog contains a type hierarchy of *data structures*, to which the services' *parameters* refer. Standardization of data structures is not a part of SENSEI, so the metamodel serves only to establish different types and specializations by unique names. *Restrictions* can be used to model relationships between capabilities and data structures, which allows to determine what specific kind of component implementing a service should be used for a given type of data at runtime. These mechanisms, as well as the detailed structure of the service catalog will be explained in Chapter 10.

The middle layer of the metamodel corresponds to service orchestrations, and allows domain experts to design processes in terms of services and their data and control flow interoperation. The central concept of this part of the SENSEI metamodel is the *service instance*: whereas the catalog contains service "blueprints", its instances represent concrete usages or invocations in particular scenarios, with certain capabilities selected. A service instance *conforms* to a service from the catalog. The service's parameters dictate their instance's *ports*, which can be connected by data *flows*. Control flow is dictated by the order in which service instances appear in an orchestration (the metamodel's corresponding association is *ordered*). Special kinds of orchestrations (subclasses of *ServiceContainer*) exist to represent conditional branching, concurrency, and loops, but are omitted from Figure 9.2 for clarity. The full details of this layer are presented in Chapter 11.

The bottom layer of the metamodel corresponds to component registries, and allows tool developers to register their tools with SENSEI, and describe their functionality. The central concept here is the *component* – its instances represent actual components available to SENSEI and contain meta-information about them, such as their name and location (e.g. a path in the local file system, a URL, or something similar – the concrete contents are implementation-specific). The component registry further requires *data definitions* to be specified for each parameter of each implemented service, to map the conceptual data structures defined in the service catalog to concrete technical

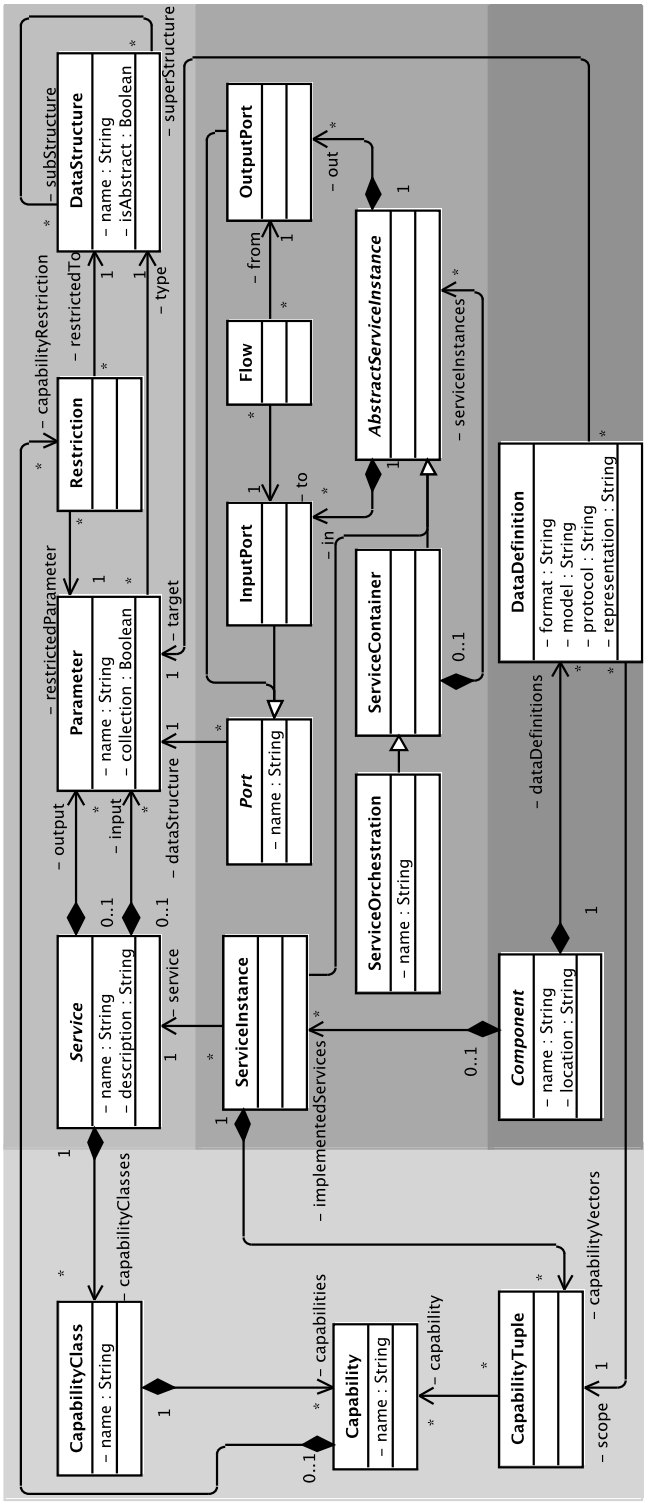



Figure 9.2: Excerpt of SENSEI's integrated metamodel, depicting its layers and their core concepts.

realizations. To indicate what services are offered by components, they are associated with one or more service instances. They do not directly refer to catalog services, because components generally do not implement a service's whole spectrum of possible functionality, but provide a subset of it. Service instances, along with capabilities, provide the necessary mechanism to narrow this down. Reusing the service instance *concept* that is also present on the service orchestration layer does not mean, however, that these layers overlap. Orchestrated service instances are always distinct from those used in the component registry – there can be instances that are equal in terms of their properties, but no single service instance is ever shared between the two layers. Finding matches between the two is one of the tasks of the *composition finder*. Details regarding the component registry and the process of service-component-matching will be given in Chapter 12.

The orthogonal layer depicted at the left-hand side of Figure 9.2 corresponds to *capabilities*, which are the topic of the following section. In terms of the metamodel, capabilities do not represent a separate artifact, but rather a connective fabric integrating its layers. *Capability classes* and *capabilities* are defined on the service catalog layer for associated services, to model variation points. They are referred to from within *capability tuples* used on both the service orchestration and the component registry layer, to refine the nature of required and provided services, respectively.

## 9.3 Service Capabilities

There are opposing requirements in SENSEI regarding the *granularity* of service description detail, as the higher service level demands more abstraction, while on the lower component level much more specificity and technical detail is needed[2].

The service catalog demands more general services to facilitate standardization, e.g. specify a *calculate metrics* service, not *calculate McCabe metric on Java code*. Such fine-grained descriptions would lead to a catalog cluttered with only marginally differing services, making it hard to identify the right services for a given task. In orchestrations, the high abstraction level hides interoperability issues.

In contrast, service orchestrations do need to be specific about the functionality required, e.g. here it is necessary after all to declare *McCabe* as the actual metric to be evaluated, and *Java* as the data to evaluate it on. Purely technical properties of particular implementations, e.g. that the input Java AST needs to be encoded in a specific XML format, should still remain hidden, though. In a component registry, both functional and technical properties need to be described rigorously, to enable automatic toolchain generation by matching up provided with required functionality, and be able to coordinate tools and accommodate for non-compatible data formats.

[2]This exposition is a slightly revised excerpt of a short, earlier publication [Jelschen and Winter, 2014].

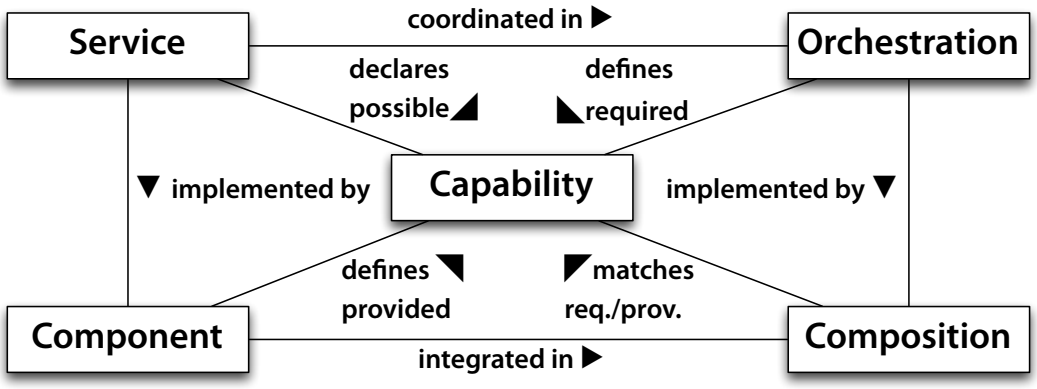



Figure 9.3: Capabilities as a central, integrating concept in SENSEI.

To bridge between these different abstraction levels, a means for synchronization is needed to map from services to components, and orchestrations to component compositions, representing an executable toolchain.

SENSEI introduces a simple model to explicitly represent the *capabilities* of services, supporting

- concise, generic service descriptions,
- implementation-agnostic, yet functionally precise orchestrations, and
- functionally and technically rigorous component descriptions and service mappings.

This allows SENSEI to keep service catalogs uncluttered, hide interoperability issues in orchestrations, and have sufficient technical detail on components to automatically compose them into toolchains.

Service capabilities play a central role in SENSEI, and are considered a distinguishing feature of the approach. Figure 9.3 depicts central concepts of SENSEI and their relationships with *capabilities*.

On the level of services, capabilities are introduced as a mechanism to keep them generic and only *declare possible capabilities* as variation points. This will be described in more detail in Chapter 10.

Services are selected and coordinated in an *orchestration* to model processes in need of tool support. Here, capabilities are used to instantiate generic services by *declaring required capabilities* specifically. Chapter 11 will provide insight into – and examples of – how this works.

*Components* implement the functionality defined by services. Capabilities allow to precisely *define provided capabilities*, which, in contrast to orchestrations, contain

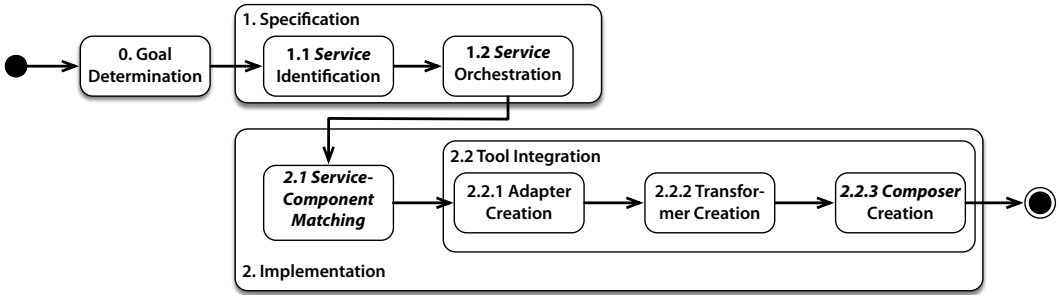



Figure 9.4: The toolchain-building process with SENSEI.

additional technical information regarding concrete data types. To finally create integrated toolchains as *compositions* of components, capabilities are leveraged to *constrain component mapping* to only match components which can provide the functionality required by orchestrated services. They are further used to *constrain component composition* to only select compatible components or add data transformers for services requiring direct interoperability. This will be elaborated in Chapter 12.

## 9.4 Building a Toolchain with SENSEI

To elicit requirements for SENSEI, Section 3.1 established the toolchain-building process – a sequence of activities aimed at creating integrated toolchains from individual tools that automatically perform a specified procedure. The purpose of SENSEI is to support and partly automate this process. This section aims at giving an overview of how SENSEI is used, by recalling the toolchain-building process and going through its steps, again – but this time with SENSEI as the sought toolchain-building support framework.

The toolchain-building process with its phases and steps is depicted in Figure 9.4, which corresponds to Figure 3.1 (page 27), except for the renaming of some steps to reflect the use of SENSEI. In particular, the word *task* was replaced by the word *service*, *task coordination* was renamed *service orchestration*, *task instantiation* became *service-component matching*, and *coordination logic creation* corresponds to *composer creation*. The changes are highlighted in italics.

SENSEI clearly assigns the different phases and steps to different roles, and supports them with two major meta-tools: the SENSEI *editor* and the *toolchain generator*. This is depicted in the use case diagram in Figure 9.5: it depicts all the steps of the toolchain-building process, assigned to the actors who perform them, and the meta-tools and their parts (as established in Section 9.1) which provide support. There are two additional activities in SENSEI, *service definition* and *component registration*, and the additional role of catalog maintainers.

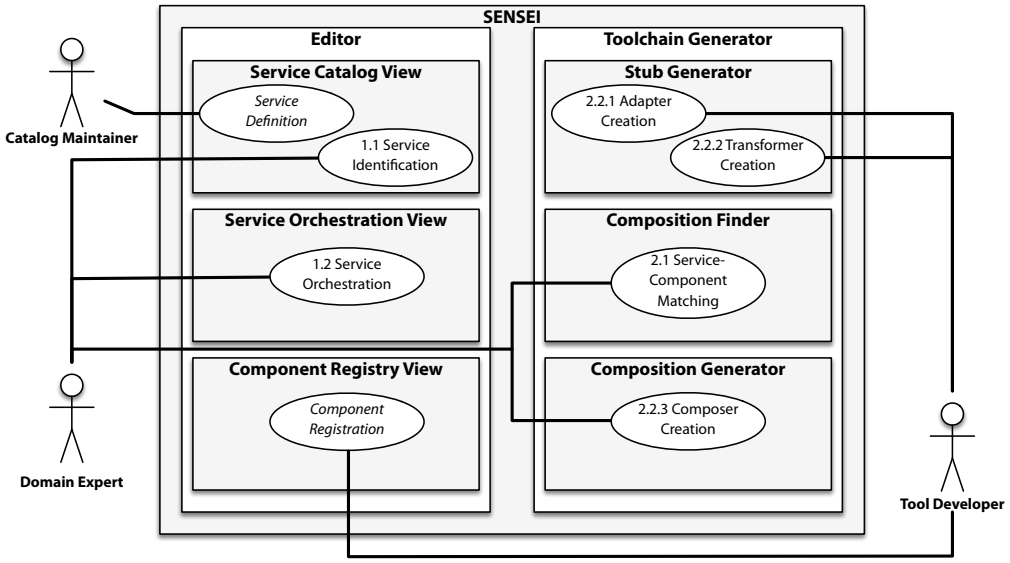



Figure 9.5: Toolchain-building process steps depicted as use cases and mapped to SENSEI's meta-tools (Meier, 2014b, p. 8 has a similar, but simpler diagram).

The job of catalog maintainers is to *define services*, i.e. initially fill, and then update the service catalog whenever necessary. Initially, this represents an overhead that comes with using SENSEI. When starting from scratch, without a service catalog, the need to define one is a real, additional effort on top of all other steps. Whether using SENSEI is economically sensible for a given project therefore comes down to whether this overhead can be offset by the productivity gains expected from using its overall environment and meta-tools. At the opposite end of the scale is the case of having a comprehensive catalog services readily available. A premise of SENSEI is that the overhead, besides being set off by automating other steps in the process, quickly becomes smaller as it is being used over time and in more projects, because the previously defined services can be reused, and only very problem-specific ones need to be added.

**Service Discovery and Description**

To fill the service catalog, either a *top-down* or a *bottom-up* approach can be used. The former approach identifies services from relevant publications and diverse software evolution projects, to create a catalog of generic, standardized services. Services can be picked from the catalog instead of being created for the project. A top-down process for service discovery and description, based on mining publication databases and clustering techniques, is sketched briefly by Jelschen [2013],

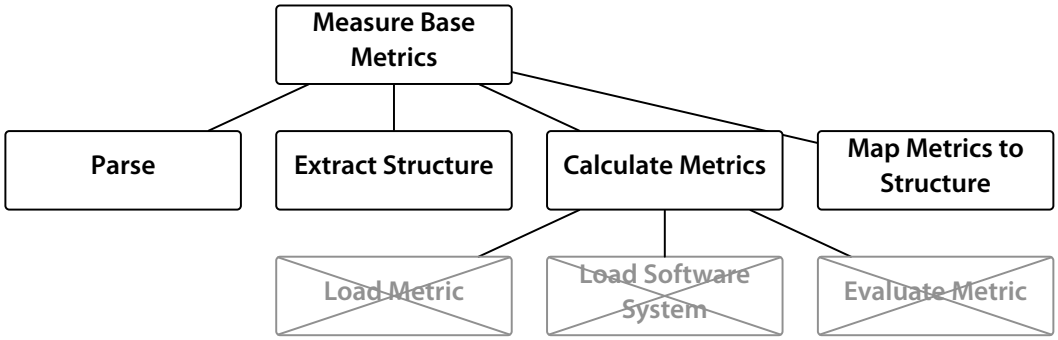



Figure 9.6: Decomposition of an activity to calculate base metrics.

**Service Discovery and Description (cont.)**

along with classification schemes and a corresponding description metamodel to complement the SENSEI metamodel described in this thesis.

Lacking a comprehensively filled catalog, services can be created *bottom-up* instead, only for a project's required functionalities, giving full control over service design, but potentially leading to project-specific services with lower reuse value. Still, this approach can be used to fill a catalog incrementally, and refine and generalize its services in the process.

Relevant to either approach is the issue of *service granularity*: when to stop breaking down an activity into sub-steps to discover *service candidates*. One heuristic is to keep in mind that services need to be *independent* and *self-sufficient*.

Figure 9.6 depicts a possible subdivision tree for the ongoing example of *base metric calculation* (Section 2.2): to calculate metrics, they, and the software systems under study, have to be *loaded into memory*. But this leads to an activity, *evaluate metric*, that cannot be executed *independent* of the other two.

Now consider that components implementing these services are strictly decoupled from each other by SENSEI – they might even be hosted on different machines distributed over a network. SENSEI will take care of data transport, and may (de-)serialize or transform it on the way. Data loading as a separate service is thus useless, and *evaluate metric* lacks *self-sufficiency*, because when provided with input data by SENSEI, any implementing component *has* to load that data itself.

Decomposition trees like the one shown in Figure 9.6 are a useful aid for service identification and orchestration, with atomic service candidates appearing on its leaves and potential orchestrations – which are also (compound) services – as root and inner nodes. Further examples are given in Chapter 15.

Ideally, the tasks of tool developers are also decoupled from the toolchain-building process: SENSEI supports them through its *stub generator*, so they can equip their tools with *adapters*, directly, and provide *transformers* up-front, as well. While SENSEI itself does not dictate any particular target platform, conforming implementations do have to decide for a (component) framework that establishes the necessary interoperability standards and enable generic tool adapters. To make them available to SENSEI-integrated toolchains, tool developers register their tools in the component registry, which represents another step that is added by SENSEI. This is a trivial task, though, and does not represent any meaningful overhead.

With these steps out of the way, domain experts are basically left with the steps of the specification phase, only: service identification is supported by the service catalog view of the SENSEI editor, and service orchestration by its orchestration view. Figure 9.5 also shows the remaining steps assigned to domain experts, but this is only to indicate that they trigger them – the actual work is fully automated: the composition finder of the toolchain generator performs *service-component matching*, which is followed by *composer creation* performed by the composition generator, resulting in a complete, executable, fully-integrated toolchain.

All participants in the toolchain-building process use the SENSEI editor, each focusing on another view. As a preview to Chapter 13, and to give a more tangible impression of what such a meta-tool can look like, Figure 9.7 shows a screenshot of the implementation built as part of this thesis. In the center, the orchestration view is visible, showing an orchestration that models the *base metric calculation* example first introduced in Section 2.2. The visual orchestration language, meaning both its abstract syntax defined by the corresponding layer of the SENSEI metamodel, as well as the concrete syntax created for the editor, will be described in Chapter 11. Without explaining the details, one can make out the four service instances shown as boxes with an encircled *S*-symbol in the top-left corner, connected by data flows (green) and control flow sequences (grey), and embedded in control structures (here, a concurrency and a map block are used).

While orchestrations are modeled graphically in this implementation, the service catalog and component registry views are each made up of a tree view (visible at the left-hand side), paired with the properties view (visible at the bottom). Here, the Q-MIG service catalog[3] has been opened, showing its data structure hierarchy, and its services, with the *Parse* service node expanded to reveal its parameters and capabilities. The component registry at the bottom left is basically a list of component entries, each of which contains one or more service instances with capabilities.

[3]The catalog contents are reduced to those used in the example orchestration. In the course of applying SENSEI to the Q-MIG project, a much more comprehensive catalog was built up, which will be described in Chapter 15.

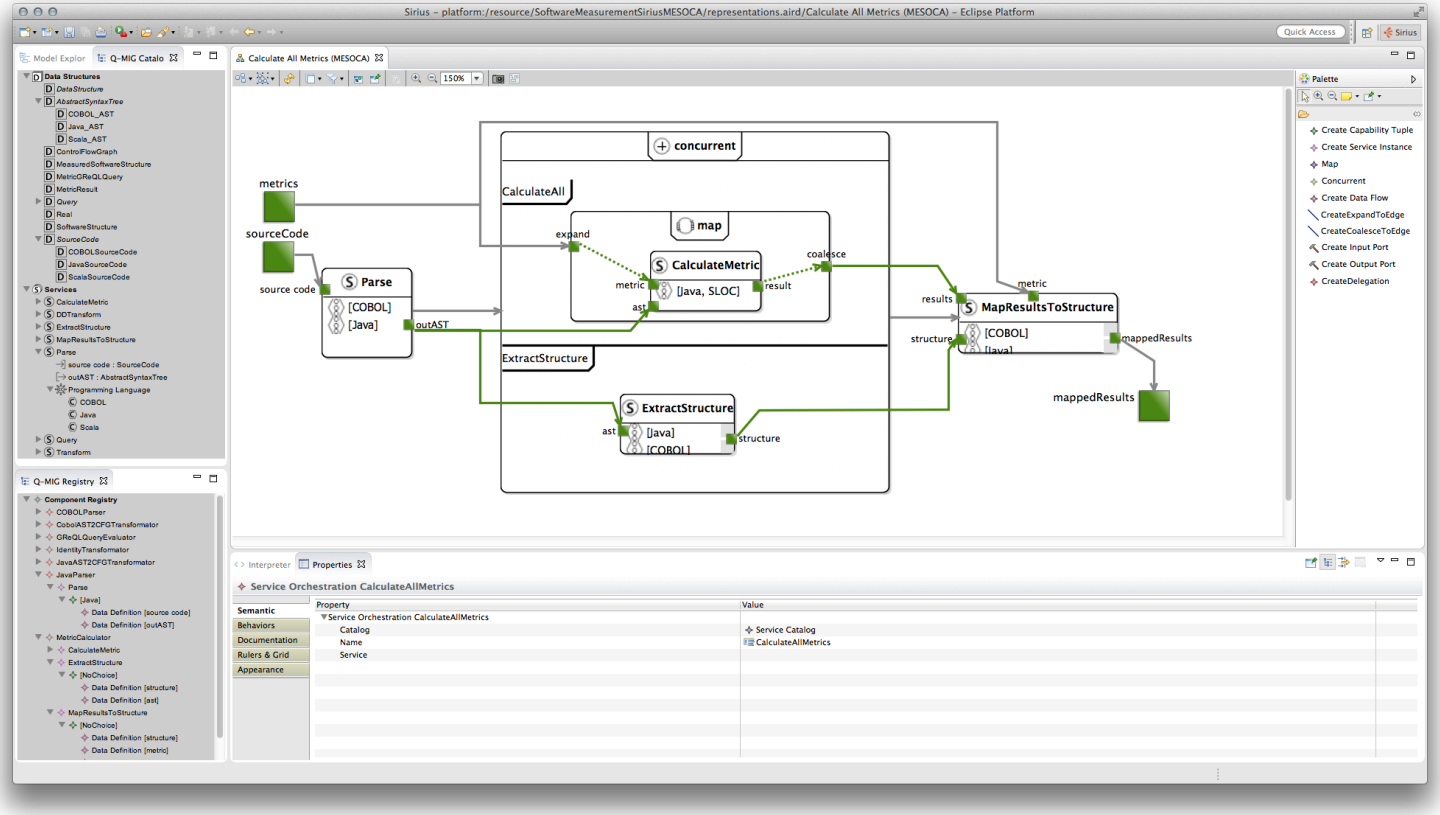



Figure 9.7: Screenshot of the SENSEI editor, showing a model of the *base metric calculation* example.

## 9.5 Summary

This chapter gave an overview of the basic structure of SENSEI. Its design decisions can be traced back to the objectives of this thesis, the refining requirements elicited through analysis of the toolchain-building process, and lessons learned from previous work and related approaches. Its principles are founded in well-established tenets of software engineering in general, and the component-based, service-oriented, and model-driven paradigms in particular. They are summarized as follows:

**Definition 9.1: Principles of SENSEI**

**Abstraction.** SENSEI establishes clear separation of concerns, particularly between *specification* and *implementation*, and defines correspondingly disjoint roles and responsibilities.

**Organization.** SENSEI demands specifications to be explicitly modeled in accordance with its layered metamodel, enforcing separation of concerns and definite, uniform structures.

**Correlation.** SENSEI introduces capabilities to bridge the gap between specification and implementation, and allow declarative specification of requirements and provisions.

**Automation.** SENSEI leverages separation of concerns to isolate mechanical tasks, machine-readable specification models and definite target structures to synthesize integrated solutions, and capabilities to automatically select appropriate implementations.

The SENSEI principles provide a very general distillation of its guiding philosophy. In fact, while they address the three major objectives of increased *flexibility*, *reusability*, and *productivity*, they do not actually refer to toolchain integration. SENSEI is a toolchain-building framework, but its underlying principles are more universal, and can serve as the foundation to create frameworks to solve similar problems. A concrete example is modeling and integrating *interactive applications* – this is out of scope for SENSEI, but a SENSEI-inspired approach to address this challenge has been developed in the context of the NEMo project, and will be briefly described in Section 16.4.2.

The following three chapters are dedicated to describing the three main layers of the SENSEI metamodel: the *service catalog* (Chapter 10), *orchestrations* (Chapter 11), and the *component registry* which enables automatic matchmaking between services and components (Chapter 12). Capabilities play a different role on each level, which will be explained in each of the corresponding chapters. After this exhaustive description of the SENSEI as a conceptual framework, Chapter 13 and Chapter 14 introduce the tools supporting the approach, SCAffolder and the SENSEI *Editors*, respectively.

CHAPTER 10

# Service Catalog

Standardized service specifications are a prerequisite for both using them in orchestrations, and for implementing them as components: both domain experts requiring services when designing orchestrations and tool developers providing the corresponding functionality in the form of components have to agree on what makes individual services, e.g. what a *Parse* service is expected to do, including its interface on a conceptual level. Without this, services cannot be matched to implementing components automatically, or at all: manual matching would require to adapt to the discrepancies between assumptions made by domain experts and tool developers, essentially breaking down the separation between services and components. The service catalog is therefore an indispensable part of the core principles of SENSEI.

The structure of the service catalog is defined by the SENSEI metamodel. This chapter introduces the modeling concepts used to represent services, starting in Section 10.1 with their most basic constituents that define their conceptual interfaces, and the data they consume and produce.

With regards to the fundamental structure of the catalog, an issue that arises is about service granularity. For example, a *Parse* service would be too generic for both domain experts and tool developers to precisely specify their needs and tool provisions, respectively. Domain experts may need a service to *Parse Java*, or even more specifically, a service to *Parse Java Version 5 trough 8*. However, a "flat" catalog of very fine-grained service definitions will quickly become disorderly and unmanageable.

SENSEI introduces *capabilities* as a means to control service granularity and model variants (Section 10.2). While ostensibly a very simple feature, capabilities are at the center of SENSEI's automatic service-component-matching. They permeate all three main layers of the metamodel, providing a measure of integration, and a means for concise, declarative specification of required or provided functionality. Service *restrictions* (Section 10.3) build on capabilities, defining how particular variants narrow the

service's interface. Restrictions can be leveraged when executing orchestrations, to select appropriate components for a given service, based on the kind of input data present at runtime. The details of this will be explored in the upcoming chapters.

Through its structure, the service catalog establishes a description template that determines what properties SENSEI services possess, and how they are defined and documented. Furthermore, Chapter 13 presents integrated editors, that allow to fill SENSEI models, including the definition of services for the catalog. A separate question arising is, which services a catalog for a particular domain (such as software evolution) should contain, and how catalog maintainers should go about finding these services.

Recalling their brief introduction in Section 9.4, two approaches can be distinguished: Either, a comprehensive catalog is standardized up-front, and in a *top-down* manner, for the whole company, industry, or domain. Or, services are defined in a more pragmatic, needs-based, *bottom-up* approach, during the course of a concrete project. While the top-down approach (depending on the actual scope) may require considerable amounts of time, effort, and resources, the bottom-up approach may yield services that are too confined to a particular application area, and thus lack reusability value (compare Kokko, Antikainen, and Systä [2009]).

This thesis is focused on the core principles of SENSEI and provisioning a working toolchain-building framework that confers the desired properties of increased flexibility, reusability, and productivity (recall Section 1.2). Therefore, these questions are not elaborated in greater detail here. For practically applying SENSEI (Part V), the bottom-up approach for service catalog creation has been used. However, a process for discovering software evolution services and describing them in a top-down fashion, based on text-mining relevant academic literature, has been developed and experimented with [Jelschen, 2013].

## 10.1 Services and Data Structures

Figure 10.1 depicts an excerpt of the SENSEI metamodel, showing the core concepts of the service catalog. Some concepts that are relevant only for the catalog's organization and service discovery, such as service classification means, are omitted here. The core concepts considered here provide the basic service description template, and are directly used by service orchestrations and component registries, as well as SENSEI's processors. The concept of *capabilities* and the related *capability classes* and *restrictions* will be explained in Section 10.2.

Unsurprisingly, the central concept of the service catalog is the *service*. A SENSEI service serves as an abstraction of some functionality, in line with the most basic definition of the term as "unit of functionality" (Section 7.2), but without referring to implementation aspects in any way, as opposed to some of the definitions used in SOA literature (see Chapter 7). Instead, mappings to available implementations are maintained separately in *component registries*, which will be introduced in Chapter 12.

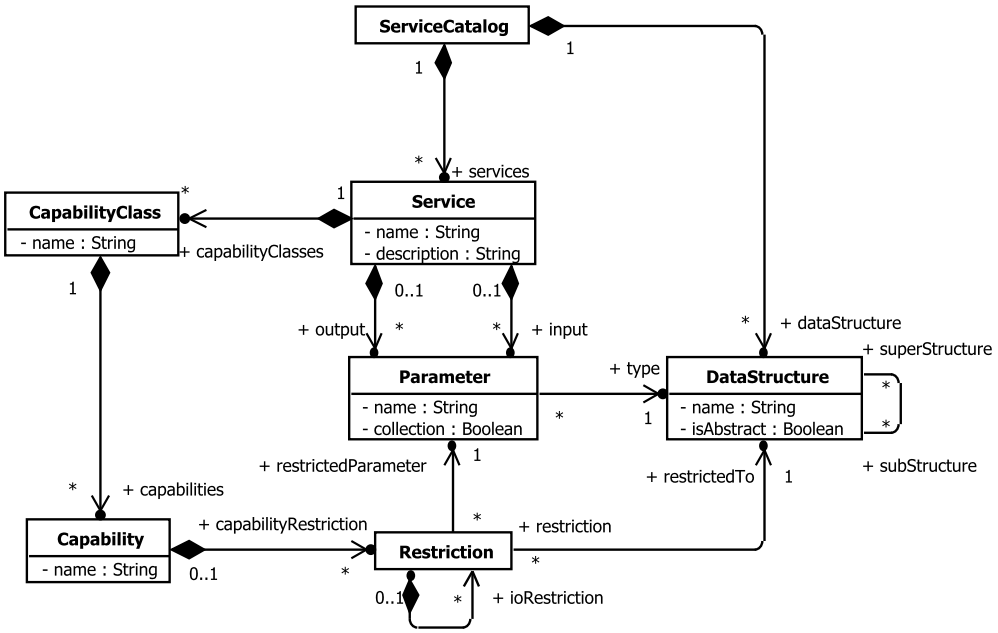



Figure 10.1: A view of the central service catalog concepts defined in the SENSEI meta-model.

For identification, a service bears a *name*. The *description* contains a non-formal specification of the semantics of a service, i.e. what functionality it represents. Since it is non-formal, it cannot be leveraged technically, and is only intended to aid domain experts in finding appropriate services.

A formal description of service semantics was considered inexpedient with respect to SENSEI's objectives: it would make service specification an extremely complex task, requiring proficiency in some formal specification language, which is a highly specialized skill. SENSEI tries to leverage existing skills of tool developers, instead, who are assumed to be proficient in one or more programming languages, and are not experts in formal specification. A complete, formal specification of a service would, in theory, allow the fully automatic derivation of an implementation, but it would in all likelihood also incur *at least* the same amount of effort required for a conventional implementation. The service model of SENSEI could be extended by means to allow partial specification in terms of formal requirements or constraints which would enable formal verification; this might be an interesting extension, but is simply out of scope with respect to the objectives of this thesis.

Since it is a unit of functionality, a service can also be thought of as a *function*. A function is defined in terms of its inputs and outputs, and how it relates those. Accordingly, to be able to perform anything of interest, services require *inputs* and *outputs*. Both are represented as *parameters*: services can have an arbitrary number of input and output parameters. Their *name* indicates the role the data provided on the parameter takes in the context of the corresponding service. SENSEI provides a rudimentary mechanism to represent *collections* of data items. The flag indicates that the data passed through a parameter needs to be interpreted as some kind of iterable collection (data series, stream, list, set, etc.).

Each parameter represents a different kind of data element fed to the service for processing, or returned by it as a result. The "kind of data" is described by *DataStructures*. In SENSEI, a data structure is identified by its *name* only. There are no means to *actually* describe the structure that data entities are expected to conform to. Analogous to services, a data structure only identifies *what* kind of data needs to be represented, but leaves the question of *how* to represent it open to implementations (see Chapter 12).

**A Note on Terminology: Data Structures and Types**

The terms *data structure* and *data type* seem to be used somewhat interchangeably; different authors may use the same term but with different meaning, and often the terms are used without explicit definition. For example, Wirth [1986, pp. 18, 157] seems to be using the term "data structure" to refer to a *kind of type* (as in arrays, linked lists, etc.), as well as to refer to an *instance of a type* (concrete data elements conforming to a specification expressed as data *type*).

In SENSEI, a distinction is needed between an *abstract* concept of exchanged data on the service level, and concrete data format specifications on the component implementation level. Note that *abstract data type* also is an already established term that focuses on the operations defined on an underlying data abstraction [Dale and Walker, 1996]. Therefore, SENSEI introduces its own definitions of both *data structures* and *data types*; the latter will be given in Chapter 12. In short, data structures are a technology-agnostic, service-level concept, whereas data types exist exclusively on the level of components and map data structures onto concrete technical implementations.

Still, standardized descriptions of data structures could be included on the service catalog level, e.g. in terms of implementation-agnostic metamodels or ontologies. While this may simplify data integration, a comprehensive standardization of data formats is an extensive and complex undertaking. Its feasibility is questionable when considering the scope of SENSEI, which aims at broad applicability. Existing approaches usually target narrower domains, like *SOFAS* (Section 5.6), which incorporates a data ontology, but is restricted to software analysis. This issue has come up earlier in this

thesis, e.g. Karsai, Lang, and Neema [2005] discuss it in the context of their integration patterns (see Section 4.3.3. The problem can also be likened to the creation of business-wide data models, briefly mentioned (and discouraged) in the context of service-orientation (Section 7.3), indicating that such a comprehensive, integrated data model would be hard to maintain and extend, and possibly lead to tight coupling.

The DataStructure class of the SENSEI metamodel can be thought of as an extension point for data integration approaches. On its own, SENSEI offers only its transformer concept; without a common data model, the number of transformers required to establish data interoperability between $n$ different data formats is $n^2$. Except for use cases with very diverse and highly interconnected services, in the long term, transformer reusability makes this a pragmatic and viable solution.

Data structures may be part of a specialization and generalization hierarchy (in terms of subtyping, not inheritance; data structures are pure entities without an attached definition of behavior). The specialization relation establishes a partial order on data structures (reflexive, transitive, and anti-symmetric), i.e. the hierarchy is really a directed, acyclic graph. A substructure is a subset of all its superstructures. A parameter with a particular data structure as type can therefore handle data entities conforming to either this data structure, or to any of its substructures. A data structure can be marked *abstract* to indicate that it is meant as a superstructure only. It has no direct instances, i.e. any data entities conforming to it, also conform to a subtype (implying that an abstract data structure should have at least one substructure defined). The data structure hierarchy plays an important part in deducing service interoperability on the data integration level, and is linked to service capabilities. The discussion on these issues is continued in Section 10.2 and Section 10.3.

### 10.1.1 Example Services

Back in Section 2.2, the base metric calculation process of the Q-MIG project was introduced. Its four basic steps can be translated directly into corresponding services: *Parse*, *ExtractStructure*, *CalculateMetric*, and *MapResultsToStructure*. They are shown in Figure 10.2 using a uniform, tabular *description template*, which lists name, a non-formal descriptive text, as well as input and output parameters. Parameters are listed with their associated data structure type. Also, an asterisk in square brackets following the parameter name is used to indicate that it is multi-valued (in terms of the SENSEI metamodel: the parameter's *collection* attribute is set to *true*). This notation of multiplicity and type is borrowed from UML class diagrams. One of the services, *CalculateMetric*, is also depicted as object diagram in Figure 10.3. This just serves as an example of how a service description conforms to the SENSEI metamodel – the object diagram notation shows this relation more immediate than the tables.

Notice that this is merely a list of service descriptions – an excerpt of a larger service catalog – and *not* an orchestration of services. Orchestrations instantiate services from

| Service | | |
|---|---|---|
| | Name | Parse |
| | Description | Parses code files and returns an abstract syntax tree / graph (AST/ASG) representation. |
| | Input | source code : SourceCode |
| | Output | outAST : AbstractSyntaxTree |

| Service | | |
|---|---|---|
| | Name | ExtractStructure |
| | Description | Extracts a basic, hierarchical decomposition of a software system's AST. |
| | Input | ast : AbstractSyntaxTree |
| | Output | structure : SoftwareStructure |

| Service | | |
|---|---|---|
| | Name | CalculateMetric |
| | Description | Evaluates the specified metric over a software system's abstract syntax tree and returns the result. |
| | Input | metric : Metric<br>ast : AbstractSyntaxTree |
| | Output | result : MetricResult |

| Service | | |
|---|---|---|
| | Name | MapResultsToStructure |
| | Description | Takes a hierarchical decomposition of a software system, as well as a series of metric values calculated on the same system, and combines the two by annotating the structural elements with their corresponding metric values. |
| | Input | results [*] : MetricResult<br>structure : SoftwareStructure<br>metric [*] : Metric |
| | Output | mappedResults : MeasuredSoftwareStructure |

Figure 10.2: Simplified service descriptions for the Q-MIG base metric calculation example (more details will be provided in Section 10.2 and Section 10.3).

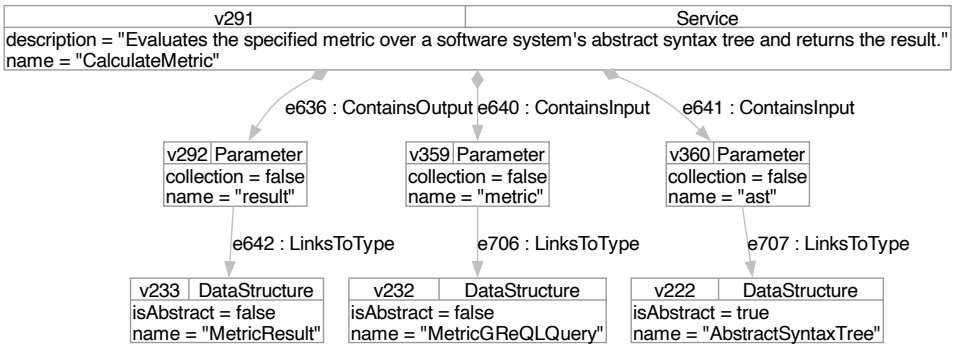



Figure 10.3: The service *CalculateMetric* depicted as object diagram-like graph, an instance of the SENSEI metamodel.

the catalog and establish relations between those service instances in terms of control and data flow. The mechanics of how this is done in SENSEI are described in Chapter 11.

Both *Parse* and *ExtractStructure* are fairly self-explanatory (though the reader might want to go back to Section 2.2 to recall the intended functionality of the process and its individual steps). Both take one input and return a single result. The associated data structures of their parameters also indicate that they can be chained.

*CalculateMetric* takes two inputs: the software system to be analyzed, and the metric to be calculated. It could be argued that the type of the *ast* parameter should be relaxed, as software systems may also be represented by their source code, or their binaries. This would give tools more freedom to choose a format appropriate for their implementation, and allow for more flexibility and better reuse by combining the service with transformers as needed. To keep this first example a little simpler, the input parameter's data structure has been chosen to match the output parameter of the *Parse* service. Conversely, the data structure of parameter *metric* is completely abstract (also simply named *Metric*). Recall that, on the level of services, data structures are merely symbolic names, for which implementing components may choose a concrete representation: The type *Metric* implies what the data fed to the service at the corresponding parameter should represent, but not how. Different implementations may require a simple, symbolic name to look up the corresponding metric internally, a file path or URL pointing to an external specification to load a metric from, or even have the parameter carry the metric algorithm itself (e.g. as expression in an appropriate query language). The output data structure, *MetricResult*, is similarly abstract.

Finally, there is *MapResultsToStructure*, a service specifically designed to combine

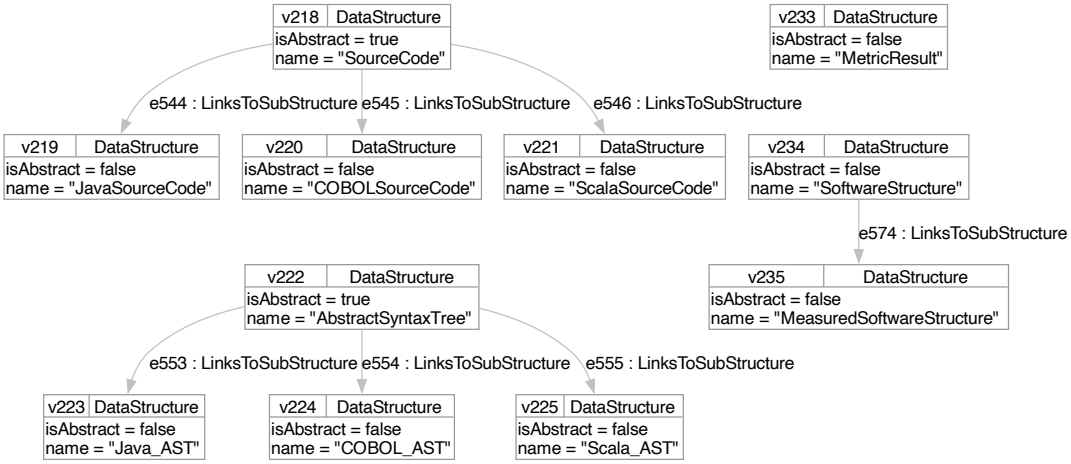



Figure 10.4: Data structures for the Q-MIG base metric calculation example.

the results of *ExtractStructure* and *CalculateMetric*. It takes collections of metrics and corresponding calculation results, as well as a software system structure to write the results to. While *Parse*, *CalculateMetric*, and to some extent *ExtractStructure*, are reusable for multiple purposes (the latter e.g. in software architecture reconstruction), potentially with off-the-shelf tools available, *MapResultsToStructure* is more clearly derived directly from the Q-MIG project's specific needs. This is indicative of two different kinds of services. It is part of SENSEI's philosophy – and natural in a service-oriented approach – to encapsulate any project-specific logic that cannot adequately be constructed by orchestrating existing services in a custom service. Apart from presumably having comparatively low reuse value, it still is "yet another service", conceptually no different from any other services. Given that the need for such project-specific functionality is generally unavoidable, i.e. domain experts in this case *will* have to double as tool developers, this is a clean and consistent approach. Also, in case at some point additional use cases for such services should emerge after all, they are readily available (along with their implementing components).

Also worth noting is that *MeasuredSoftwareStructure* is modeled as a sub-structure of *SoftwareStructure*. All data structures used in this example are depicted in Figure 10.4 in object diagram notation, including sub-structure relationships. Sub-structures are still purely symbolic, however, the relationship implies that data conforming to a sub-structure can be passed to or from parameters typed with their super-structures. Figure 10.4 also includes several exemplary sub-structures of *SourceCode* and *Abstract-*

| Service | | |
|---|---|---|
| | Name | CalculateMetric |
| | Description | Evaluates the specified metric over a software system's abstract syntax tree and returns the result. |
| | Input | metric : MetricGReQLQuery<br>ast : AbstractSyntaxTree |
| | Output | result : MetricResult |
| | Capability Classes | SupportedMetrics = {McCabe, SLOC, CloneLines, ...}<br>SupportedProgrammingLanguage = {COBOL, Java, ...} |

Figure 10.5: The *CalculateMetric* service again, now with capability classes.

*SyntaxTree* for representing software systems of particular programming languages. So far, the service descriptions do not make use of them, but will be refined in Section 10.3. Together with *capabilities*, sub-structure types are utilized by SENSEI's processors to find the most appropriate components to match particular service instances.

## 10.2 Service Capabilities

Services are characterized by their input and output parameters, and by what their general function is, i.e. what they *do*. Returning to the running example, it makes sense to have a service for *metric calculation*, as it represents a commonly required software evolution functionality. While such an abstract description is beneficial for finding required services in the catalog, to use them in concrete orchestrations, the functionality needed has to be specified more precisely. To find suitable, implementing components (automatically), it must be known, e.g. *which* metrics will have to be calculated, and on what *kinds* of software systems. There are hundreds of software metrics and programming languages, but actual metric calculation tools will usually only support a fraction of those. Cluttering the catalog with descriptions of fine-grained, specific services representing all possible combinations of supported programming languages and metrics for analysis would completely defeat its purpose of easing service discovery.

Therefore, service descriptions in SENSEI are kept generic. Each service in the catalog may define arbitrary many *capability classes*. As introduced in Chapter 9, capabilities are a central concept in SENSEI, running through all levels of its metamodel, and serving different purposes on each of them. In the catalog, they are a means of organizing it, enabling concise descriptions of coarse-grained, generic services.

Capability classes are used to declare the dimensions of variability that can exist – like supported *metrics* and *programming languages*. Figure 10.5 shows the service description of the *CalculateMetric* service again, this time also including capability classes. Each capability class defines a set of possible values – the actual *capabilities*.

For the sake of brevity, only a few examples are shown in Figure 10.5. When instantiating a service, either to use it in an orchestration, or to register a component, a subset of the available capabilities are chosen to specify required and provided capabilities, respectively. The exact mechanisms for this are detailed in Chapter 11 and Chapter 12.

**Capabilities Defined by Components**

Capability classes and their capabilities are part of service descriptions and the service catalog, as it makes sense to standardize them: using them for specifying service requirements and matching them to component provisions only works if all domain experts and tool developers have the same capabilities available to them, and a common understanding of their individual meaning. On the contrary, it might be hard or even impossible to foresee all potentially useful capabilities. For example, a particular parser might only support a few specific dialects of COBOL – and this might be sufficient for an orchestration for a particular software evolution scenario. While not manifested in the SENSEI metamodel, implementations of SENSEI processors and editors may choose to dynamically load capabilities declared by components through the registry and make them available to domain experts, even if they are not part of the corresponding, standardized catalog services. This could also be used by tool vendors to expose proprietary features as provided capabilities though, and relying on them would lead to vendor lock-in.

The prototype implementations of the SENSEI processors (SCAffolder, Chapter 14) and editors (Chapter 13) currently do not support such a feature, but the metamodel does not preclude it. When defining services needs-based, for a specific software evolution project (bottom-up), as opposed to using an already available catalog of standardized services (top-down), this does not become an issue, anyways, as capabilities can be added immediately when the need arises.

### 10.2.1 Capability Modeling Pragmatics

Capability classes are the main tool of SENSEI to control service granularity. Picking up on the sidebar discussion on service discovery and description given in Section 9.4, there are no hard rules on what is allowed, or what makes "proper" capability classes. However, a few rules of thumb can be given, based on the experience of practically applying SENSEI (see Part V):

One question that arises is how many capability classes should be modeled. Consider why *CalculateMetric* (Figure 10.5) has two classes. It seems straight-forward, because the two aspects being modeled – metrics and programming languages – are clearly distinct. But, if a metric like, for example, *average methods per class* is added, this view is arguably upset, because such a metric has only meaning for object-oriented

languages which have the concepts of both methods and classes. COBOL has neither, so the combination of that metric with this programming language makes no sense.

In and of itself this does not pose a problem for SENSEI; it merely means that there are combinations of capabilities which in all likelihood no actual component will ever support. However, if the capability classes produce many meaningless capability combinations, it might become hard for service catalog users to discern those from the useful ones. There may also be services for which the choice of most, if not all, capabilities of one class depend on the choice of another class. In such cases, it might be better to collapse them into a single capability class, instead, with capabilities defined by the cartesian product of the original classes (omitting the useless combinations). If this was done for *CalculateMetric*, the resulting, single capability class would contain capabilities like *SLOC_Java* and *SLOC_COBOL*.

Another aspect to consider during capability class design is the level of abstraction of the overall service. For example, the semantics of *CalculateMetric* could also be covered by a much more generic *CalculateFunction*, with appropriately generalized capability classes, parameters, and associated data structures.

This has upsides and downsides, e.g. it might lead to the identification of additional tools that can be applied to a given problem, but it can also make it hard to find appropriate services for particular problems when browsing the catalog. Avoiding such over-abstraction is a matter of applying domain knowledge and common sense: in the most extreme case, every service can be viewed as a function that maps an input into output. In fact, this is basically the definition of what a service is, but this abstraction level is obviously useless to solve specific problems in a given domain. Also note that it is perfectly valid to model the specific *as well* as the generic service, giving tool developers the option to support one or the other, or even both.

### 10.2.2 Capability Semantics

Like data structures, capabilities are purely symbolic, and merely represent certain aspects of services with an agreed upon meaning. For example, the capability *COBOL* of class *SupportedProgrammingLanguages*, defined for the *CalculateMetric* service, implies that an implementing component declaring its provision is actually able to calculate metrics on software systems written in COBOL. These semantics should be made clear in the service description – the examples shown in Figure 10.2 and Figure 10.5 only feature one-liner descriptions for the sake of brevity. When using the service catalog as a means for standardization, these descriptions would have to be much more detailed and precise to serve as design contracts.

Since these semantics are only specified in prose in SENSEI, such contracts cannot be technically enforced. Formal specifications were considered far too extensive and expensive to be of value, and thus were dismissed when designing SENSEI. A comprehensive formal specification of a service would probably amount to a (declarative)

implementation in terms of required effort and resulting complexity. In most practical cases, this should not pose a problem, as components "lying" about their capabilities can simply be removed and replaced, if such a case should ever occur. For (potentially expensive) commercial tools, compliance test suites might be defined, though often claimed conformance to already existing industry standards will take on this role. In any case, such issues are not within the scope of SENSEI.

SENSEI does feature a concept to define a certain kind of semantics, after all, to establish relationships between parameters and their associated data structure types, and capabilities. They are referred to as *restrictions* and are mainly utilized by SENSEI's Composition Finder to match orchestrated services to components in a way that observes data compatibility.

## 10.3 Service Restrictions

With capability classes and capabilities, it is now possible to characterize the required or provided functionality of service instances more precisely, based on generic catalog services. The data structures associated with the parameters of these services are usually, and intentionally, very abstract (see Section 10.1.1). Using *CalculateMetric* as example once more, if only *Java* is picked as capability from the *SupportedProgrammingLanguage* class, than this has implications for the kind of data it can accept. Such a service instance would not be able to accept COBOL abstract syntax trees, or anything other than Java ASTs, on the *ast* parameter, for example. This is important when chaining up services: if the input for the *ast* parameter is provided by a *Parse* service instance, then it must output Java ASTs, not arbitrary ones. While this may seem obvious and could be achieved by careful orchestration, SENSEI's automation features can assist, and make more sensible matches of components to services, if these interdependencies between capabilities and parameter data structures are formally specified.

The automation tooling becomes even more useful in cases where multiple capabilities are specified, as is the case in the base metric calculation example, which needs to support both COBOL *and* Java. Obviously, this cannot happen at the same time[1], but one at a time. Then the relationship between capabilities and data structures of parameters can be utilized in reverse: at runtime, the actual input data is introspected for its specific data structure type, and the appropriate components are invoked (either a Java or a COBOL parser, for example) based on which provides the associated capability (more on this in Chapter 11).

[1] Of course, COBOL and Java metric calculation can be performed concurrently, by invoking the corresponding toolchain once for each case, and running them in parallel. An input software system cannot be both programmed purely in Java and in COBOL, though, and that is what is meant here. It also does not mean that mixed-language systems cannot be handled – if appropriate tools exist, a capability like *COBOLJava* can be defined, so there is no limitation of SENSEI here.

SENSEI therefore provides *restrictions* of parameter types to sub-structures of their associated data structures, based on chosen capabilities. As shown in Figure 10.1, arbitrarily many restrictions can be defined for a capability in the SENSEI metamodel. A restriction only relates to a parameter and a data structure. The data structure must be a sub-structure of the data structure that is the type of the parameter the restriction links to[2]. A restriction can be written and understood as a logical implication:

$$\text{select}(\mathit{Java}) \Leftarrow \text{typeOf}(\mathit{ast}) \lhd \mathit{Java\_AST}$$

This can be read as "The capability *Java* needs to be selected when the type of the data at parameter *ast* is *Java_AST*". The symbol $\lhd$ (as opposed to an equality sign) indicates *assignment compatibility*, i.e. the type of the data at parameter *ast* must conform to the *Java_AST* data structure, or a substructure. Furthermore, two functions are informally introduced for the notation used in this example. The *selection* of a capability refers to a running toolchain. The controlling composer component needs to choose between capabilities, as it may need to invoke different components for different capabilities. The select operator therefore checks whether the specified capability has in fact been selected in this manner. The operator typeOf is also to be interpreted at toolchain runtime, and returns the data structure corresponding to the type of the actual data available at the specified parameter. The same restriction is shown in context of the associated service, *CalculateMetric*, as object diagram in Figure 10.6.

The "runtime semantics" are also the reason why the implication may seem reversed, as restrictions have been introduced from the perspective of catalog maintainers, earlier: if a capability is chosen, then a parameter is restricted to a particular sub-type of its base data structure. However, the selection expressed by the select operator does not refer to domain experts defining required capabilities during orchestration design. Rather, toolchain composers auto-generated by SENSEI select at runtime from *those* capabilities defined in the orchestration. The actual data available at parameters cannot be influenced by composers – it represents the basic facts available to it to deduce the capabilities that were implicitly requested by piping in that data.

With the example given, the question may arise whether capabilities and the ability to define subtypes of data structures essentially amounts to the same thing. The concept of restrictions is only needed if these two concepts are separate, and since the relation between the capability *Java* and the data structure *Java_AST* seems to be straight-forward and one-to-one, why not dispense with capabilities and restrictions altogether, and just use sub-structures directly? SENSEI has capabilities, because this question implicitly assumes that services are fully determined by their input and output parameters and their associated data structures.

[2]This kind of constraint cannot be modelled in UML alone, and is therefore not visible in Figure 10.1. It has to be expressed separately, e.g. using *OCL* [*Object Constraint Language* 2014].

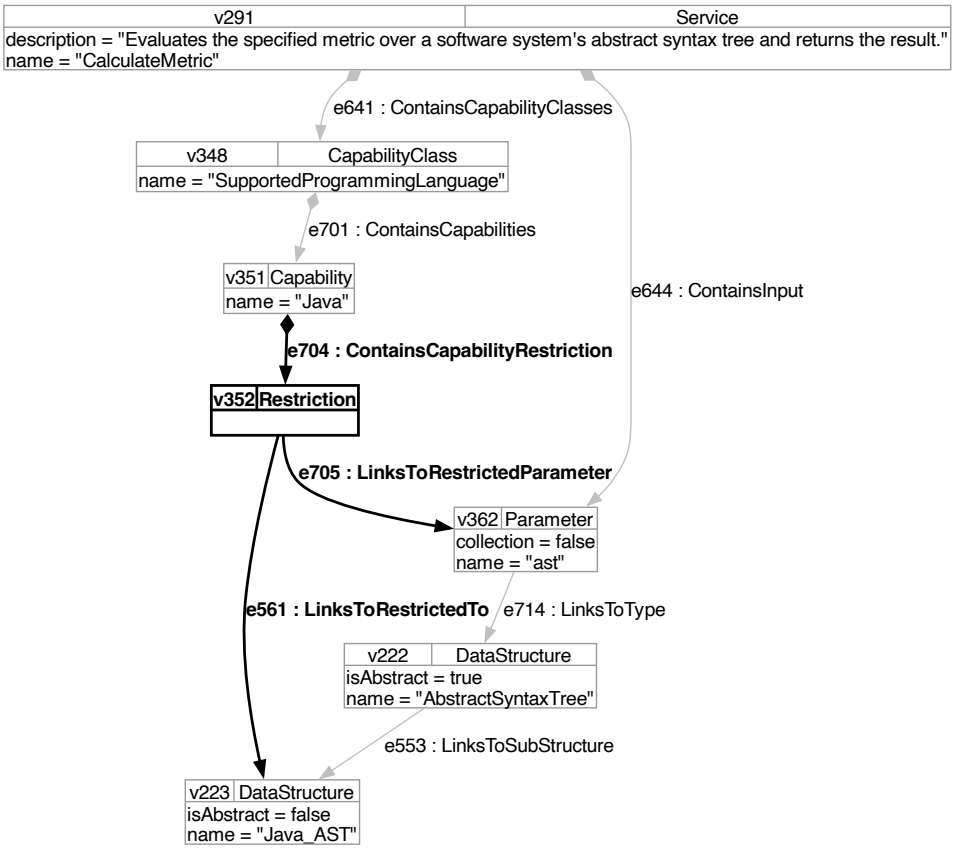



Figure 10.6: Example of a restriction as graph-based instance of the SENSEI metamodel.

Consider the *ExtractStructure* service: it could be fitted with a capability class modeling the supported depth and granularity of the extracted structure. In Java terms, capabilities could indicate the ability to extract structures down to class, method, or even statement level, for example. This would, intuitively, neither have an impact on the input nor on the output parameter. Still, it *could* actually be modeled only with subtypes of the output parameter's data structure, by defining a separate data structure for each granularity level. Since data structures in SENSEI are purely symbolic, they could even be mapped to the same technical data type representation in implementing components (Chapter 15 presents examples of this). Nonetheless, such data structures seem artificial, and defining capabilities like this is cumbersome.

Capabilities can also be used to represent non-functional properties of services. For instance, different traversal strategies could be modeled for the *CalculateMetric* service as capabilities. Many software evolution services, like clustering for architecture reconstruction (see Jelschen et al. [2013]) or clone detection, can be implemented using fundamentally different algorithms, where some might be more appropriate than others for a given task. Modeling this purely through parameters seems even more contrived, as changing such a non-functional capability would not only not change the types of the parameters, depending on the nature of the "strategy", the actual data returned would also be exactly the same (maybe delivered faster or slower).

Finally, use cases for capabilities can be considered which do not map to data structures at all, but would also require an extension of SENSEI. These are discussed briefly in Section 18.3. The utility of capabilities and restrictions will be examined further in the following chapters:

- Chapter 11 will show how to use capabilities to define required service properties declaratively and concisely, partly as an alternative to manually model more complex control flow logic.
- Chapter 12 will briefly explain how tool developers can use capabilities to register components, and provide an in-depth description of how restrictions can be leveraged to find the best combination of components for a given orchestration.
- Chapter 14 will present the automated toolchain generation as implemented in SCAffolder, and how succinct capability specifications are turned into considerably more complex, fully auto-generated control logic.

## 10.4 Summary

This chapter introduced the structure of the service catalog of SENSEI, as defined by the top level of its metamodel. As such, it provides the basis for both domain experts and tool developers to choose services from for designing toolchains as service orchestrations and map services to implementing components, respectively.

A distinguishing feature of SENSEI are its service capabilities. In the service catalog, they serve to provide a means of control over the granularity of services, allowing

to define generic services and model different dimensions of variability as capability classes. Chapter 11 will describe how domain experts make use of this feature to specify required capabilities within orchestrations, while Chapter 12 will show how tool developers do the same to map components with provided capabilities to services.

The service catalog also provides foundations for automatically matching orchestrated services to suitable components, using capabilities in conjunction with restrictions. These mechanisms can be used both during toolchain generation, as well as at toolchain runtime, to route between different components implementing the same service (but with different capabilities) based on actual input data. This, too, will be explained in detail in Chapter 12.

CHAPTER 11

# Service Orchestration

SENSEI aims to enable domain experts to specify toolchains independent from the actual tools, and the integration issues that would arise on a technical level. Services, defined in the service catalog, provide the basis for that. These services then need to be combined in a manner that forms the desired processes: arranging them in a meaningful running order and determining how input and output data is passed between them. The means to do this in SENSEI is by creating *service orchestrations*. This chapter describes the metamodel concepts used to represent orchestrations, and the graphical language based on them, to be used by domain experts to model orchestrations.

Even though there are already a plethora of workflow, process, and orchestration languages (e.g. BPEL, BPMN, YAWL, UML activity diagrams – see Section 5.8 and Section 7.5), creating a new one dedicated specifically to the purposes of SENSEI, has had several advantages: it allowed to directly use, and freely design, the orchestration layer of the SENSEI metamodel, instead of having to either compromise on it, or transform between it and the metamodel of a preexisting language. Specifically, this approach allowed for the seamless integration of capability specification into the language.

Otherwise, the language constructs could be kept minimal: only concepts specifically necessary for a meaningful evaluation of the overall approach have been included, especially in terms of control flow constructs. The *structured* control flow approach has been taken partly because of ease of implementation, but it also holds the benefit of being easily extended with additional control flow constructs. The strong separation between services and components, and the code generation approach allow for further extensions with minimal or no changes to the language at all, by implementing configurable, advanced semantics of existing language concepts as part of SENSEI processors.

Conversely, BPEL, which in the early stages of this thesis had been investigated to be used as orchestration language, is tied closely to web services, which expose very technical interfaces, thereby opposing a key requirement of SENSEI. In general, existing

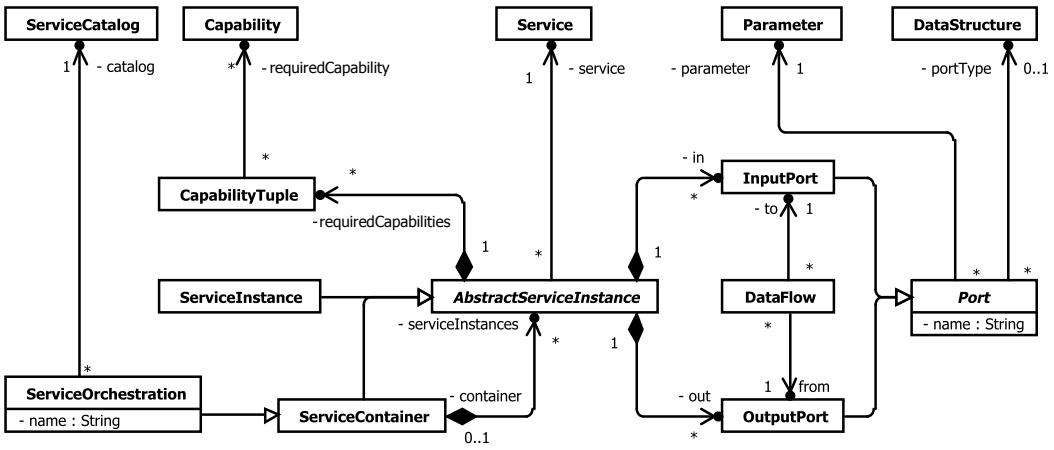



Figure 11.1: A view of the central service orchestration concepts defined in the SENSEI metamodel.

orchestration languages usually come with certain dependencies, use very particular language concepts, and make assumptions that are unsuitable for, or hard to align with, the concepts of SENSEI. In particular, the specification of required capabilities would have necessitated extending the chosen language.

An overview over the central concepts for service orchestration in the SENSEI metamodel is depicted in Figure 11.1. The top row of classes are actually concepts from the service catalog (cmp. Figure 10.1) that are being referenced from orchestrations. The service orchestration concepts can be divided into four areas, by which they will be described in more detail, in the following:

**Service Instances** Besides the class *Service Orchestration*, which provides a context for orchestrations and acts as container for its constituents, the base element for orchestration modeling is the *Service Instance*, representing concrete uses of catalog services to which they conform (Section 11.1).

**Required Capabilities** Instantiating a service mainly entails selecting the *required capabilities*, which are modeled as *Capability Tuples* (Section 11.2).

**Data Flow** Service instances possess *Input* and *Output Ports*, in accordance with the parameters defined by their corresponding catalog services. Ports can be connected by *Flows* to model data exchange (Section 11.3).

**Control Flow** SENSEI employs a fully structured control flow model, with concepts for conditional branching, looping, and concurrency. The corresponding metaclasses are mostly based on *Service Container*, but have been omitted from Figure 11.1 for better clarity. They will be presented in detail with a separate view of the SENSEI metamodel (see Figure 11.6 in Section 11.4).

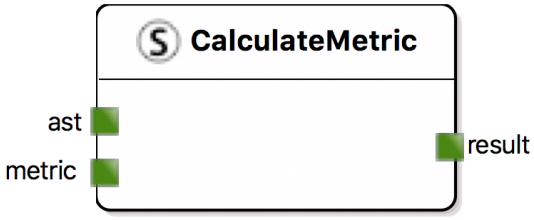



Figure 11.2: An instance of the *Calculate Metric* service as defined in Section 10.1.1 (without capabilities).

## 11.1 Service Instances

The title of this chapter is "Service Orchestration", but actually, "service *instance* orchestration" would be more precise. It is not the services described in the catalog themselves that get orchestrated. The catalog contains service definitions that act as templates, made generic by capabilities. Service instances conform to these services, but are representing a particular usage in an orchestration, with only a specific subset of the available capabilities indicated as required, and data and control flow relations to other service instances in the same orchestration. Obviously, the same service can be used in arbitrarily many orchestrations, but these instances are distinct - they do not share the same required capabilities or interrelations.

SENSEI service orchestrations are meant to be modeled graphically. Therefore, a graphical, concrete syntax is needed. An example instance of the *Calculate Metric* service is depicted in Figure 11.2. Service instances are represented by rounded rectangles. It is horizontally separated into two compartments, the upper one carrying the name of the corresponding service as label, preceded by an icon indicating the type of service instance. An encircled "S" is shown for standard service instances (other kinds of services will be introduced in Section 11.4). For each port of the service instance, a solid box is drawn on the border of the shape. Ports are labelled with the name of the corresponding service parameter, written next to the box. In this thesis, data flows and ports are always depicted in green.

Service instances, as represented by the class of that name, are the leafs of a composite pattern rooted in the class *Abstract Service Instance*. Nesting is done using *Service Containers*. In particular, *Service Orchestrations* are service containers, and thus service instances. This means that orchestrations can be used as service instances in other orchestrations to model even more complex processes. There is one caveat, though: service orchestrations being service instances means there needs to be a corresponding service in the catalog – but if there were such a service readily available, there would be no need to model an orchestration from more primitive service instances. For service orchestrations, the association is therefore interpreted inversely: orchestrations *induce* corresponding services. These services may then be included in the catalog.

Service instances *conform to* exactly one service, represented by an association in the SENSEI metamodel. This relation is further constrained, though:

- Required capabilities must refer to capabilities contained in capability classes of the instance's service.
- For each input (output) parameter of the service, the instance must contain an input (output) port referring to that parameter. A service instance may *not* define additional ports.

A comprehensive set of constraints to complement the SENSEI metamodel has been gathered and formalized as OCL expressions by Meier [2014b, pp. 63ff], including the ones defining the conformance relation between service instances and services, which are only given in prose, here. Both constraints can be supported and enforced by proper tooling, as implemented in the SENSEI editor (Chapter 13). Ports can, in fact, be generated completely automatically. The main task of domain experts when instantiating services for use in orchestrations is therefore the specification of required capabilities.

## 11.2 Required Capabilities

A major aspect of orchestration in SENSEI, that sets it apart from other approaches, is the ability to partly specify it *declaratively*, using capabilities. Instantiation of generic services from the catalog for concrete usage in orchestrations is mainly done by specifying *required capabilities*. The service catalog specifies which capabilities are available for any given service: Each service has one or more *capability classes*, each with a set of actual capabilities (Details are given in Section 10.2).

To specify required capabilities for service instances, at least one capability from each capability class of the corresponding service has to be chosen. If multiple capabilities are required from more than one class, the semantics of the selection may become ambiguous. Consider the example service *Calculate Metric* again. It has two capability classes (go back to Figure 10.5 for details): *Supported Metrics* and *Supported Programming Languages*. Lets assume the software evolution project in need of toolchain support requires the *McCabe* and *SLOC* metrics to be calculated, and both *COBOL* and *Java* need to be supported. The selection of these capabilities would express that. But, maybe SLOC is actually only needed for Java, while the McCabe metric should indeed be calculated for both Java and COBOL. This would lead to the same set of capabilities, even though implementing components might be chosen which support the latter, but not the former case.

To resolve this ambiguity, SENSEI introduces *Capability Tuples* to group selected capabilities. For a service with $n$ capability classes, an $n$-tuple, which contains exactly one capability chosen from each of the classes, forms such a capability tuple. Returning to the example, the capability tuples necessary to specify that the Calculate Metric

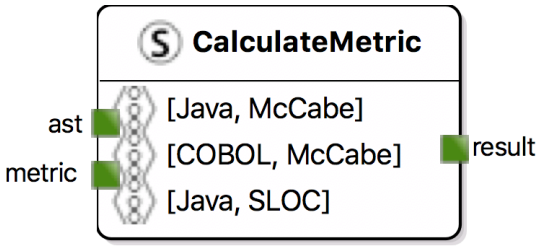



Figure 11.3: An instance of the *Calculate Metric* service as defined in Section 10.1.1, *with* capabilities.

service instance must be able to calculate the McCabe metric on both COBOL and Java, and SLOC on Java only, are as follows:

$$\begin{pmatrix} \text{Java} \\ \text{McCabe} \end{pmatrix}, \begin{pmatrix} \text{COBOL} \\ \text{McCabe} \end{pmatrix}, \begin{pmatrix} \text{Java} \\ \text{SLOC} \end{pmatrix}.$$

Each of these capability tuples represents a minimal functional unit, i.e. the service instance of Calculate Metric requires these three specific manifestations of the generic service to be supported. At runtime, a single invocation of the service instance's functionality (implemented in a component) will be required to support exactly one of these capability tuples. Which one depends on the actual type of the data available at the service instance's ports, and the restrictions (see Section 10.3) defined for the service.

A service instance with capability tuples is depicted graphically as in Figure 11.3. The tuples are listed in the lower compartment of the service instance shape, enclosed in square brackets, with the capabilities separated by commas. The SENSEI meta-tools interpreting this, or generating code from it, are responsible for expanding this into a case distinction. Figure 11.4 illustrates this, but intentionally does *not* use SENSEI syntax, because in SENSEI this is modeled much more concisely – the single service instance with capability tuples shown in Figure 11.3 is all that is needed[1].

So in essence, a service instance with $n$ capability tuples can be separated into $n$ *minimal* service instances with one capability tuple each. This is an important observation for service-component matching (Chapter 12), as matches need only be found for each of these minimal service instances, so that several components can be picked to satisfy the requirements of a single service instance. Conversely, minimal service instances are atomic, i.e. they cannot be realized by multiple components working in concert.

The guard expressions on the control flow branches in Figure 11.4 are partly derived from restrictions defined in the service catalog (Section 10.3). Where there are no restrictions, the fallback is to ask the components implementing the service instances

[1] SENSEI does allow to model control flow branching explicitly, though, too (see Section 11.4).

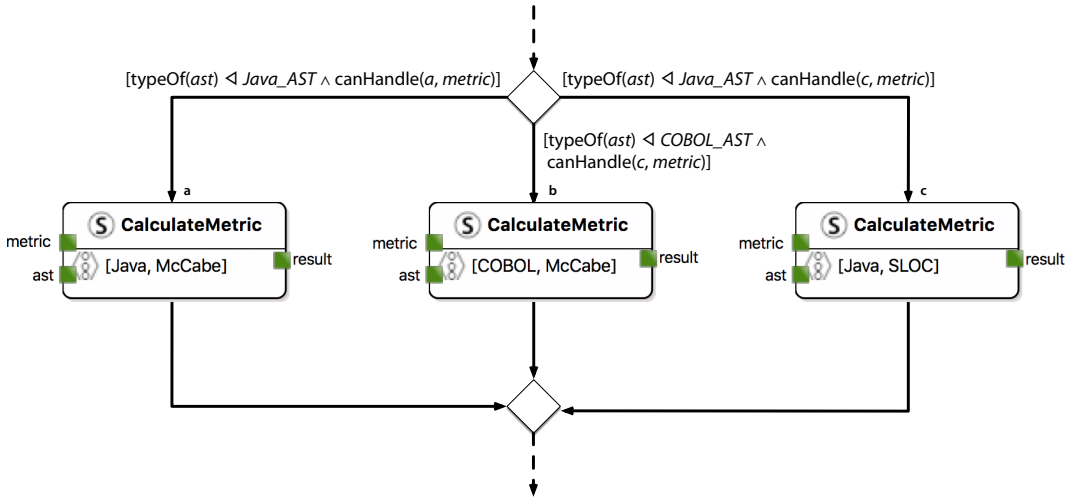



Figure 11.4: The semantics of specifying multiple capability tuples (Figure 11.3) corresponds to a control flow branch.

whether they can handle the data at their ports. SENSEI therefore requires components to provide appropriate facilities to do so; how this is done, technically, is an implementation detail of SENSEI *processors*. Here, this is indicated by the predicate symbol canHandle. Also, the service instances have been adorned with letters $a$, $b$, and $c$ to reference them. Again, all of this happens "under the hood" – domain experts do not have to worry about any of this, as they only have to declare their required capabilities to express this.

The example shows one way by which SENSEI, and its capability concept in particular, help achieve the overall objectives of this thesis: increasing *flexibility*, *reusability*, and *productivity* (Section 1.2). The branching pattern depicted in Figure 11.4 does not have to be modeled by domain experts, because the necessary work is performed by catalog maintainers (service definitions with capabilities and restrictions) and tool developers (uniform components with runtime self-description and introspection abilities), who only do this work *once*, enabling **reuse**. This shift in responsibilities takes much of the integration modeling and implementation off the shoulders of domain experts, reducing the required effort and thus opening up opportunities to increase **productivity**. Just by comparing Figure 11.3 and Figure 11.4, which do not even show the implementation details hidden behind both modeling alternatives, and by extrapolating from there, it is possible to imagine the relative conciseness and simplicity that SENSEI orchestrations can achieve. This makes makes them easier to understand and to be adapted more **flexibly**.

## 11.3 Data Flow

The data integration solution of SENSEI is designed to be simple and extensible, as the focus of the approach is on process integration (see Chapter 4). No common data formats or data models are dictated by the approach. Rather, technical data incompatibilities remain invisible on the service level. At the component level, tool developers can freely decide how to model and represent their input and output data. Mismatches must be resolved by transformers – corresponding services are inserted automatically into orchestrations at toolchain generation time, as necessary.

The specification of data flow is kept simple and straightforward, as well, with a focus on extensibility. *Input* and *Output Ports* represent service parameters on service instances. Ports are connected by *Flows* (see Figure 11.1), each piping the data provided at an output port into an input port. The main constraint is that the data structure returned from a service instance at an output port, must be assignment compatible (see Section 10.3) with the data structure expected at the input port targeted by a connecting flow. These conceptual *port types* are defined in the service catalog, i.e. through the data structures associated to the corresponding service parameters. Technical incompatibilities do not arise at the orchestration level: Concrete implementation types are defined by components and are only taken into consideration when the service instances are matched to components during toolchain generation. The matching process, described in more detail in Chapter 12, will try to avoid conflicts due to incompatible data formats, and will automatically insert transformers, if necessary and possible. Only if no solution of mapping the service instances to components and available transformers can be found, new transformers have to be created.

The semantics of data flows are as follows: after a service instance finishes execution, the data it returned on its output ports is copied to all input ports connected via flows. A single output port can be connected to multiple input ports, which will result in a copy of the data provided to each of them. The opposite is also possible, i.e. an input port being fed by multiple output ports. The data at the input port will simply be overwritten every time a service instance with output ports connected to it finishes execution. An output port may even feed an input port of the same service instance, which can be useful if the control flow contains loops. Multiple output ports of the same service instance connected to a single input port will lead to unspecified behavior, though, and is therefore prohibited.

Data flow semantics are strictly separate from control flow semantics in SENSEI, too. As opposed to UML activity diagrams, for example, where an action is invoked as soon as there are tokens available on all its incoming edges, irrespective of whether these are control or data flows, here data flow does not determine control flow. Service instance invocation is not triggered by the availability of input data, and control flow must therefore be designed to ensure that all required data is in fact available when service instances get invoked.

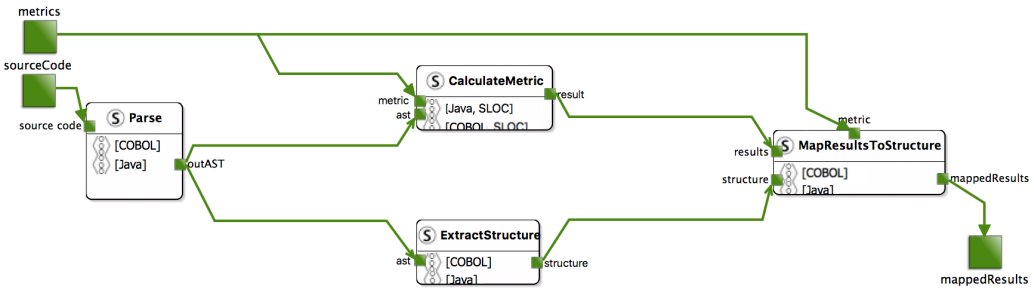



Figure 11.5: Service instances to model the Q-MIG base metric calculation example, with ports connected by data flows.

Figure 11.5 continues the Q-MIG base metric calculation, and shows the required service instances with ports connected by data flows. Ports and data flows are always drawn in green color, to clearly distinguish data flow from control flow arrows, which appear in gray. Control flow is not present here, yet – it will be added in Section 11.4. This will also require a slight modification to the data flow, because *CalculateMetric* service instance needs to be executed once for every metric in the list of metrics that is input into the orchestration. In Figure 11.5, the metrics list flows directly into the *CalculateMetric* service instance, which actually expects a single metric, only.

Orchestrations, being service instances themselves, also possess input and output ports. By default, all input ports of contained service instances without any incoming flows become input ports of the orchestration, and all output ports of contained service instances without any outgoing flows become output ports of the orchestration. If this is not the desired behavior, ports can be specified explicitly, represented by squares larger than the boxes drawn on service instances to depict their ports. This is useful, for example, if several service instances should be fed the same data, as is the case with the *metric* input ports of the *Calculate Metric* and *Map Results To Structure* service instances. Another use case would be if the output of a service instance is used internally, first (e.g. in a loop), and only the result of the final execution should also become the result of the orchestration returned at its own output port.

To specify how the data of service orchestration ports should be conveyed to and from ports of its service instances, *data delegations* are used[2]. These are different from data flows, because they either connect an external input port to an internal input port (*downwards delegation*), or an internal output port to an external output port (*upwards*

[2]Delegations are somewhat similar to the concept of *promotions* in the service component architecture (SCA) standard (see Laws et al. [2011, p. 98]). However, promotions apply to whole services there. Also, SCA is about assembling *components* rather than *services*, with a much more technical service notion than that of SENSEI.

*delegation*). Conversely, data flows are always on the same nesting level, and always connect an output port to an input port. Delegation relations are not depicted in the metamodel view in Figure 11.1, but are instead included in Figure 11.6 of Section 11.4, as they are only needed to bridge nested service instances, and nesting is the central modeling concept for specifying structured control flow.

## 11.4 Control Flow

Control flow in SENSEI is fully structured, i.e. as in structured programming [Dahl, Dijkstra, and Hoare, 1972]. This design decision is mainly a pragmatic one: it can be (meta-)modeled very elegantly and it is easy to implement. Structured control flow is modeled by nesting, which corresponds to a composite pattern on the metamodeling level. Such a structure is already established, anyways, with service instances nestable in service orchestrations. This also makes this approach very easily extensible by means of subclassing.

The control flow constructs currently provided by the SENSEI metamodel are depicted in Figure 11.6, along with advanced data flow means, that are described in more detail in the following. The complete orchestration for base metric calculation, depicted in Figure 11.7, serves as example.

*Service Container* serves as base class for the control flow classes. Only conditional branches, represented by *Alternative Service Instances*, constitute a special case, and are derived from *Abstract Service Instance* directly. Since base metric calculation does not require this construct, a separate example is given in Figure 11.8.

Plain **Service Orchestrations** represent *sequences*, the semantics being that all contained service instances get invoked, one after the other. The order of service invocation is determined by the *serviceInstances* association end on the association between *ServiceContainer* and *AbstractServiceInstance*, which is why it is explicitly annotated with the *ordered* modifier. In concrete syntax, sequences of service instances are indicated by connecting them with gray arrows. In Figure 11.7, the top-level orchestration contains three service instances in sequence: the *Parse* service instance is followed by a *concurrent* control flow block, which is followed by the *MapResultsToStructure* service instance.

**Concurrent Service Instances** allow for the specification of independent subprocesses. *All* nested service instances or orchestrations get invoked, in no particular order, and are potentially executed in parallel. Concurrent service instances are depicted as blocks with an encircled plus sign and the word *concurrent* at the top. The body is horizontally separated into lanes, each of which contain sub-orchestrations that are to be executed concurrently. An example of a concurrent service instance with two lanes can be seen in the center of Figure 11.7.

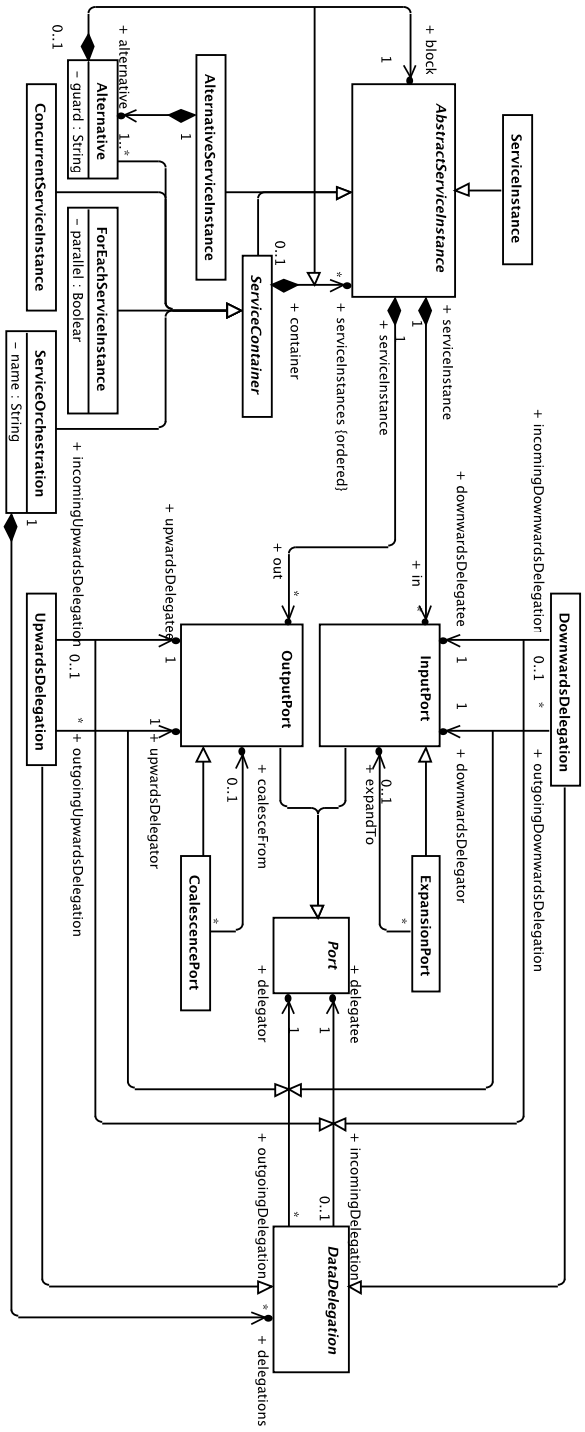



Figure 11.6: A view of service orchestration concepts for structured control flow defined in the SENSEI metamodel.

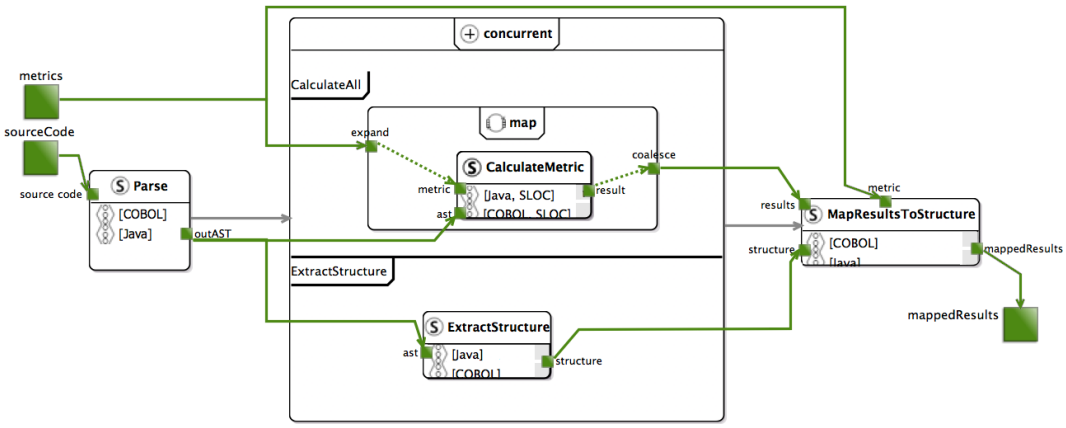



Figure 11.7: The complete SENSEI orchestration for the base metric calculation example, containing control flow sequences, concurrency, and a loop.

**ForEachServiceInstances** represent loops. They expect exactly one nested service instance or orchestration[3], which get invoked repeatedly. If set to true, the *parallel* attribute indicates that the loop iterations may be executed concurrently. *ForEachServiceInstances* possess two ports: the *Expansion Port*, a special kind of input port, determines the number of repititions. The data passed in is expected to be a collection, i.e. if there is an output port providing the data via a flow, its corresponding service parameter must have the *collection* attribute set to *true* (see Section 10.1). The collection will be iterated, and for each element, the nested service instance or orchestration will be invoked once. Furthermore, the current element can be passed into an input parameter of the nested service instance or orchestration. Conversely, an output port of the nested service instance or orchestration can be used to gather results from each execution, which will be gathered into a collection that is available at the *Coalescence Port* of the ForEachServiceInstance after it has finished execution.

The concrete syntax of ForEachServiceInstances uses a block labelled *map* (referring to the common collection operation available in many functional programming languages) at the top, and a circle with a stack of boxes at both sides, symbolizing the input and output collections. The expansion and coalescence ports are labelled *expand* and *coalesce*, respectively. Figure 11.7 has a ForEachServiceInstance with an embedded *CalculateMetric* service instance: it effectively *maps* the provided metrics onto a a corresponding list of metric results.

**Alternative Service Instances** represent branches in the control flow. There can be arbitrary many *Alternatives*, from which exactly one is chosen for execution at runtime.

[3]The corresponding association is excluded from Figure 11.6 for greater clarity.

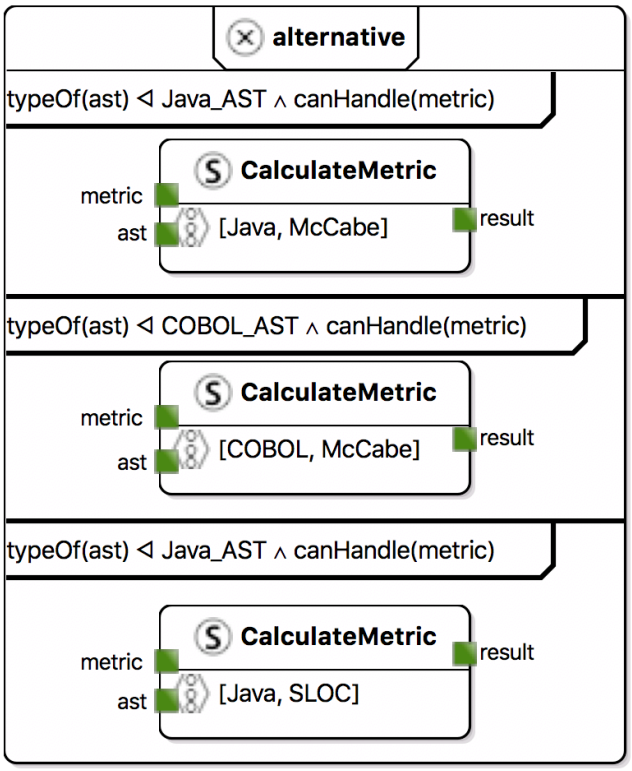



Figure 11.8: Example of a three-way conditional control flow branch modeled using SENSEI's structured *Alternative* construct. This is logically equivalent with the single service instance shown in Figure 11.3, as well as with the activity diagram-like syntax in Figure 11.4.

Each has a guard expression used to decide which branch to take. Guard expressions need to be mutually exclusive, otherwise the resulting behavior is unspecified.

Figure 11.8 shows the concrete syntax used to represent *Alternative Service Instances* and *Alternatives*. The example is the same as in Figure 11.4, which used an activity diagram-like syntax. As stated before, the exact same orchestration semantics can be expressed by just a single instance of the *CalculateMetric* service with all three capability tuples, as shown in Figure 11.3.

In its current version, SENSEI does not actually specify a concrete language for expressing guards. Control flow constructs for conditional branches have not been needed in any of the evaluation scenarios, even though they may seem to be the most straightforward ones. The reason is that common control flow branches can be expressed concisely, declaratively (and thus implicitly) through the use of required capabilities, as shown by the given example. *Alternative Service Instances* have therefore mainly been added for the sake of completeness.

A guard language would be easy to come up with, though: it needs to support basic logical operators, and a set of predicates that answer basic questions regarding the data at input and output ports, and the general state of the orchestration being executed. Inspection of data properties should only be allowed for basic types like booleans, integers, or strings, or for superficial metadata of complex types, like data type or size. Complex data queries should instead be encapsulated in dedicated services, returning a simple type (e.g. boolean), which only then is used in guard expressions.

The complete base metric calculation example, modeled as SENSEI orchestration and depicted in Figure 11.7, does not require explicit conditional branches either (it uses a loop, though). It shows sequential control flow represented by gray arrows. In the center, a *Concurrent Service Instance* allows the calculation of individual metrics to run in parallel with structure extraction. For reasons of reusability, the *CalculateMetric* service has been tailored to only calculate a single metric each time it is invoked. Therefore, it is wrapped in a *ForEachServiceInstance*, which iterates a list of metrics and invokes the nested service instance for each one. The resulting metric values get aggregated in a collection, again.

The example, stemming from the Q-MIG research project, was originally introduced in Section 2.2. An abstract representation as UML activity diagram is given in Figure 2.1. The comparison shows the direct resemblance of the two diagrams. In particular, the SENSEI orchestration is not significantly more complex: data ports and some auxiliary data flows have been added, the metric calculation step has been nested in a loop, because of the way the corresponding service is designed, and required capabilities substantiate what scenarios the corresponding toolchain will have to support.

## 11.5 Summary

The orchestration language of SENSEI allows domain experts to model their processes as orchestrations of services instantiated from the catalog. Service instances are interconnected by control and data flows in an intuitive, graphical notation that is completely free of any technical aspects. The semantics of orchestrations is expressed through the domain-specific terminology of the services, as established in the catalog. Furthermore, required capabilities provide domain experts with a simple yet concise way of expressing the needs of particular processes. Required capabilities contribute to the clarity of orchestrations due to their declarative nature, and their ability to substantially simplify control flow, that would otherwise have to be modeled explicitly. Domain experts only have to state their intents by declaring required capabilities, and leaving the rest to SENSEI to automatically generate the required conditions, control flow branches, and mappings of service instances to appropriate components.

The aim of the SENSEI orchestration language was to demonstrate the overall feasibility and advantages of SENSEI, and it has been fleshed out to the extent necessary to do just that. Its generic nature and independence of any particular technologies, leaves

room for extensions on both the conceptual level that may further amplify its general usefulness, as well as for implementations to make different design decisions to fit the needs of different use cases.

One aspect that lends itself to extensions is the capability model. As presented here, it is intended to model *functional* requirements, and assumes that particular functionalities are either absolutely required, or not required at all – there is no middle ground. For software evolution toolchains, this is a sensible assumption, as its activity usually requires exact results. In some scenarios though, it might not matter whether the *Calculate Metric* service can calculate the number of code lines (SLOC) or the number of statements – for example when trying to gain a rough impression of a software system's size. In these cases, the capability requirements could be relaxed accordingly, to be satisfied by components supporting SLOC *or* number of statements calculation. In some application domains, a graceful degradation of service quality might be preferable over not being able to provide a service at all.

Capabilities may also be extended from services to orchestrations: as mentioned before, service orchestrations are said to induce services, but do not support capabilities. One possible approach to change this would be to "lift" the capability classes from all the services instantiated in the orchestration, and make them capability classes of the induced service. Instantiating such a service again would then allow to specify capability tuples that would be passed through to the corresponding service instances. The induced service would be a true abstraction of the original service orchestration, essentially representing all orchestrations with instances of the same services, and connected by data and control flow as originally specified, but with all possible combinations of capability tuples.

A closer look reveals several open questions with such an approach: the lifting of capability classes from services is not completely trivial. Combining all classes in the induced service would lead to combined tuples to specify required capabilities, as well, which would have to be mapped unambiguously, and without loss of modeling power, to individual service instances of the orchestration. Issues would probably arise, for example, if the same service was instantiated more than once in an orchestration (which is otherwise perfectly fine within SENSEI). Thus, more research is needed to develop a proper solution for such an extension.

Another area for extensions is data flow: since SENSEI focuses on process integration, its data integration means have intentionally been left reduced to the essentials. Just as the simplistic data structure model of the service catalog can be extended to provide more sophisticated data integration means, e.g. common data models or ontologies, the data flow model on the service orchestration level can also be easily extended to include more sophisticated means. SENSEI processors (Section 9.1) may support to be configured for *smart data flows*, for example to store and retrieve data to and from a central repository instead of passing it around directly. Even more sophisticated measures could include smart caching, and automatic model differencing to only transmit

deltas (like the ones proposed by Kuryazov [2014] and Kuryazov and Winter [2014]) on repeatedly executed data flows. Mechanisms like this should be possible without necessitating a change to the SENSEI metamodel. In this regard, the minimalistic approach of SENSEI towards data flow specification, and the genericness of its service orchestration language in general, as well as clear separation of concerns, enable these potentially powerful code generation and automation facilities.

Other alternatives for data flow semantics have been considered, such as that of YAWL [Hofstede et al., 2010]. There, all data is passed around in a predefined (but extensible) XML format. Data is stored in *variables* defined on the *net* level, which can get passed to and from variables of *task instances* (variables roughly correspond to ports in SENSEI, nets to orchestrations, and task instances to service instances). What data gets passed on exactly is specified through XPath expressions, providing a very powerful means to manipulate (e.g. filter, transform) data. A similar approach could be envisioned for the software evolution domain, using GXL [Winter, Kullbach, and Riediger, 2002] as common data format, or TGraphs with GReQL [Ebert, Riediger, and Winter, 2008] for query expressions. However, this also introduces a comparatively high degree of complexity, that, following the overall SENSEI design, should not be visible on the service level, rather to be implemented as a component. While the YAWL approach may be better at facilitating *ad hoc* data manipulation, its XPath expressions used for data manipulation are bound to particular nets, and thus not available for reuse, in contrast to SENSEI.

In general, SENSEI does not offer any sophisticated, out-of-the-box means for data routing and manipulation. Another way to think about these aspects is in the sense of *enterprise integration patterns*, especially message routing patterns like *splitters* and *aggregators* [Hohpe and Woolf, 2004, pp. 259ff]. In SENSEI, such functionalities have to be expressed as services, explicitly, and must then be implemented as components. As has been said before, this is actually desirable as it promotes reuse. Additionally, it facilitates simplicity, as it does not necessitate the introduction of special concepts for these purposes, embracing the "*everything is a service*" spirit, instead.

In fact, SENSEI transformers work exactly this way (so it could be argued that there is some support, after all). Just like they are intended to be collected into libraries of standardized services and corresponding, reusable components, a SENSEI library to support routing patterns could easily be created. Those patterns that can be realized in a generic fashion (i.e. independent of concrete data formats used by different components, as opposed to transformers), would form a static library that does not grow over time, but rather contains a small, fixed set of generally useful data routing components. Middleware implementing enterprise integration patterns already exist, for example *Apache Camel* [*Apache Camel* 2020], so it should be easy to define the patterns in terms of SENSEI services, and create corresponding components that simply wrap around the preexisting middleware implementations.

CHAPTER 12

# Service-Component Matching

The key design aspect of SENSEI is the strict separation of services and components (recall the overview given in Chapter 9). The abstraction from implementation details and technical interoperability issues allows domain experts to model processes in clearer, simpler language that is close to the problem domain. Of the objectives described in Section 1.2, this mainly facilitates *flexibility*, by making it substantially easier to modify and evolve orchestrations, and thereby toolchains. On the other side of the separation line, the structure imposed by SENSEI on components implementing services promotes encapsulation, a prerequisite for, and major driving force of, *reusability*.

The separation of these aspects and their associated stakeholders and concerns can also be assumed to contribute to an increase in overall *productivity*. But just keeping service and components apart is an incomplete solution. Within the clear confines of the structure established by SENSEI on both ends, there also needs to be a bridge that allows to establish links between services and implementing components. This is the purpose of the *component registry*, which stores these implementation relationships. *Capabilities* are once more an indispensable means to make them sufficiently specific.

Notice that the component registry does not contain the actual components in terms of their source code or binary artifact, but rather references them, and houses metadata. How and where the actual artifacts are stored is implementation-specific – SCAffolder uses Maven coordinates [*Apache Maven* 2020], and relies on corresponding artifact repositories to retrieve artifacts on demand.

It therefore exists firmly on the conceptual side of SENSEI, and makes up the lower layer of its metamodel. On the modeling level, however, concrete instances of component registries may contain target platform-specific information. Also, a single component registry must not contain components designed for different target platforms, as this would preclude their integrated use in toolchains[1].

[1]The interpreter implementation SNOrcInS does not have this restriction, see Section 14.5.

| Component | Service | Capability Tuple |
|---|---|---|
| JavaFE | Parse | [ Java ] |
| COBOLFE | Parse | [ COBOL ] |
| JavaMetricCalculator | ExtractStructure | [ NoChoice ] |
| | CalculateMetric | [ SLOC, Java, File ] |
| | | [ SLOC, Java, JavaClass ] |
| | | [ SLOC, Java, JavaMethod ] |
| | | [ SLOC, Java, JavaPackage ] |
| | MapResultsToStructure | [ NoChoice ] |

Figure 12.1: An example of the service-component mapping table of a component registry for the base metric calculation example.

The information stored in the component registry is used to automate a large part of the toolchain implementation phase of the toolchain-building process (Section 3.1). It is the main artifact for tool developers aiming to either create new tools as SENSEI components, or adapt existing tools. Measures are taken to keep the required overhead minimal, so that the extra effort does not become an impediment for adopting SENSEI.

This chapter introduces the metamodel concepts that define the component registry (Section 12.1). As the title indicates, the main focus is on the process of service-component matching, which is not as trivial as just looking up entries in the component registry; several constraints regarding capabilities, restrictions, and data compatibility have to be taken into account. Therefore, Section 12.2 provides a high-level picture of the process of finding *compositions*, i.e. sets of components that can be integrated into toolchains in accordance with given orchestrations.

Solutions to the composition finding problem are prerequisites to generating toolchains for service orchestrations. The actual code generation constitutes a separate, downstream step in the overall process, though. Both finding compositions, and generating corresponding code to constitute toolchains, is realized by SENSEI's *processors* (as introduced in Section 9.1). Their prototypical implementations are the subject of Chapter 14.

## 12.1 Component Registry

In a first approximation, the component registry can be thought of as a table, with a column for components, and another for services – an example is depicted in Figure 12.1. Each row in this table establishes a relation: the referenced component *implements* the named service. This information is expected to be provided by tool developers. Whether they want to create new components, or adapt existing tools to be SENSEI-

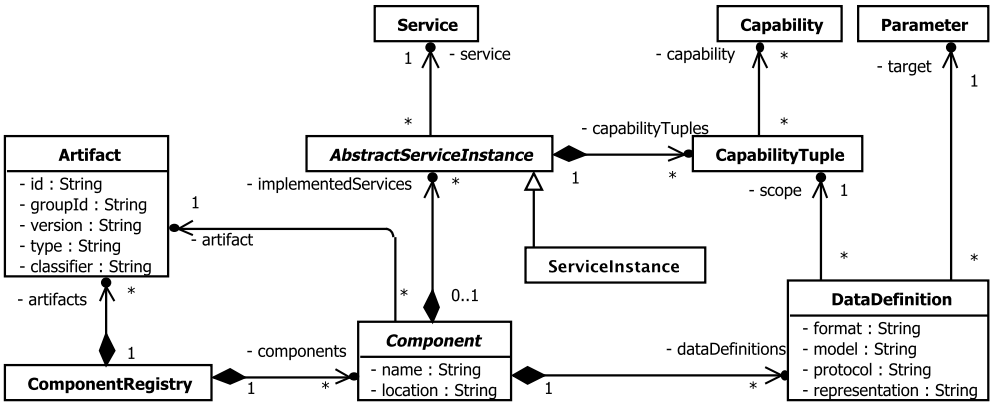



Figure 12.2: A view of the central component registry concepts defined in the SENSEI metamodel.

compatible, they have to look up services that represent the functionality that is (going to be) provided by their tools, using the catalog. This is a many-to-many relationship, i.e. a component can implement more than one service, and a service may be implemented by arbitrary many components.

Due to the abstract nature of service definitions in the catalog, components will usually not be able to implement every possible instance of a service. Just like domain experts select their required capabilities, and thereby instantiate services for use in orchestrations, tool developers also specify *service instances* with *provided capabilities* implemented by their components, to clearly define the extent of their functionality.

In the example, the JavaFE and COBOLFE components each implement a *Parse* service instance, but with different capabilities representing the programming languages they understand. Conversely, the Java Metric Calculator implements three service instances, with support for several capability tuples.

The metamodel excerpt in Figure 12.2 shows the central concepts to represent component registries, as well as concepts from the service catalog being referenced. Service instances and capability tuples are also already known from the service orchestration layer of the SENSEI metamodel (see Figure 11.1). However, entries in a component registry *do not reference* existing service instances from orchestrations. Rather, component registries use the same mechanism of service instantiation as service orchestrations do, but for a different purpose. While service instances and capability tuples contained in orchestrations on the on hand, and in component registries on the other hand, may be correlated, such relations are never persisted so that orchestrations and registries remain independent artifacts. This is important because orchestrations are evolved by

domain experts, while component registries are maintained by tool developers – establishing persistent links would violate separation of concerns, and require to update elements outside of the area of responsibility of either stakeholder group to preserve consistency. Correlating service instances of orchestrations to service instances declared by components in registries is a dynamic process: that of finding compositions, which will be detailed in Section 12.2.

Besides the class *ComponentRegistry* (which merely acts as a container), there are three concepts specific to this layer in the metamodel:

**Components** are, unsurprisingly, the main concept for registries, and serve as anchors for its entries.

**Artifacts** describe metadata related to binaries manifesting the referenced components.

**DataDefinitions** map the abstract data structures defined in the service catalog to concrete, technical means chosen to represent the data appropriately.

Each will be described in more detail in the following.

### 12.1.1 Components

Components represent entries in the component registry. As stated before, registries only reference the actual implementations in terms of executables or other artifacts, and merely contain metadata about them. Components are identified by a name. The content of the location attribute is target platform-specific – that means that the SENSEI processor implementations determine its exact syntax and semantics. It is meant to carry some kind of pointer to a component's physical representation, like a file system reference or URI. *Artifacts* can be used for the same purpose and offer more structure.

The main purpose of components is to establish links to one or more services, to express an implementation relationship. These links are not direct: similar to orchestrations, services are instantiated. Therefore, components are associated to one or more *AbstractServiceInstances*, or rather to basic *ServiceInstances*. These, in turn, are associated with capability tuples, and by extension with capabilities, as already established in Chapter 11.

The instantiation of services follows the same syntactical rules, but are given different semantics in the context of components: here, they define *provided capabilities*.

This can be illustrated using the Q-MIG base metric calculation example again. In the project, two parsers provided by industry partner *pro et con* were utilized: a COBOL parser and a Java parser, referred to as *COBOLFE* and *JavaFE*, respectively (*FE* abbreviates "Frontend"). Each parser was wrapped as an individual SENSEI component. Another pair of tools that were used are the *COBOL Metric Calculator* and *Java Metric Calculator*. The former was provided by *pro et con*, as well, while the latter was developed from scratch for the project.

```
id         = "JavaMetricCalculator"
groupId    = "de.unioldenburg.ses.qmig"
version     = "0.1"
type        = "jar"
classifier = ""
```

Figure 12.3: Artifact information of the *Java Metric Calculator* component.

### 12.1.2 Artifacts

Artifacts represent binaries[2] manifesting components, i.e. shared library files like Jars or DLLs. Artifacts allow SENSEI processors to automate dependency management and deployment. While the *location* attribute of *Components* is meant to straight-forwardly point to an artifact, e.g. a disk or network location, the information stored in *Artifacts* is declarative, tailored towards incorporating a package and dependency manager. SCAffolder downloads component artifacts and links them into a single toolchain application, whereas SNOrcInS (Section 14.5) interprets the *location* field as URLs pointing to REST endpoints of pre-deployed components.

Even though the SENSEI metamodel aims to be platform-independent, the attributes supported by *Artifacts* are taken straight from *Apache Maven* [2020]. This is because the format is very versatile, and has been adopted by several other build tools and repository managers. It allows to refer to an artifact mainly by its *id* (name), *groupId* (namespace), and *version*. In addition, a *type* (e.g. "jar", "dll", "zip", etc.) and a *classifier* (arbitrary additional information, e.g. to distinguish jar files containing binaries from those bundling JavaDoc documentation) may be specified. The artifact information stored in the component registry for the *Java Metric Calculator* is given in Figure 12.3 as an example.

As already indicated, SENSEI processors are rather free in their use of this information, and might not provide dependency management and deployment, at all. For example, SENSEI services may also be mapped to implementations offered "in the cloud", following the *Software as a Service* model [Mell and Grance, 2011]. In this case, only the generated composer would have to be self-hosted, while all dependencies would be off-site. Artifact information would then not be necessary, and the *location* attribute of *Components* could be used to point to cloud services[3], e.g. a URL to a WSDL file.

Since there are these two separate mechanisms to point to the concrete compo-

[2] If the target platform uses a non-compiled language, the files referred to might not actually be binary in nature, although even then, the code files are usually packaged in a single archive file, like *egg* packages in Python, for example.

[3] Due to the fact that "service" as a term is heavily overloaded with different meanings and interpretations (cmp. Section 7.2), discussing SENSEI in the context of cloud computing may become confusing. Here, *cloud service* refers to the internet-accessible interface of a component, which is different from a SENSEI service. In particular, the distinction between SENSEI services and components remains intact

nent, SENSEI processors may even combine the two approaches, mixing components hosted in the cloud with those available for download from an artifact repository, to be deployed to a suitable runtime environment. Of course, the deployment could target local infrastructure, or utilize a lower-level cloud model, again (either *Infrastructure as a Service* or *Platform as a Service*).

### 12.1.3 Data Definitions

Since the service catalog (intentionally) only provides a very abstract means to describe data structures, in addition to mapping services to components, tool developers have to map input and output parameter types defined in the catalog to concrete mechanisms to accept and provide data on a technical level. For this, *Data Definitions* are used.

Data definitions do not only have to be specified for each input and output parameter[4], but for each possible combination of provided capability tuple with each parameter. This allows for different implementations of data handling for different capability tuples without having to register different components.

A data definition has four attributes:

**Format** The concrete syntax and formalism used to describe the data. For example, file formats like XML or JSON, or relational database tables.

**Model** The concepts and their interrelations being represented, i.e. the abstract syntax. A metamodel or data model, consistent with definitions given in Section 8.3, and the static parts of the definition given by Brodie [1984], respectively.

**Representation** The in-memory data type. This is highly implementation-specific, e.g. SENSEI processors generating Java interfaces and code might require the class or primitive type. Data in XML format can be represented in many different ways, for example as plain string, with a file object, as JAXB object network, or DOM tree.

**Protocol** The protocol used to provide, transport, and access the data, for example through the local file system, over HTTP, FTP, or SOAP, or using a ODBC database connection.

The distinction provides a high degree of freedom when choosing how to implement data handling. The documentation of RedHat's SwitchYard framework served as an inspiration for the distinction between data format and representation [Red Hat, 2015, p. 130], and was the source of this terminology. Transformers in SwitchYard operate on these two aspects of handling data. However, they lack the concepts of *model* and *protocol* that data definitions in SENSEI possess. SENSEI transformers are used whenever there is a mismatch on one or more data definition levels between components

[4]Services instantiated in the component registry do not require and therefore do not possess ports, which is why the parameters of corresponding services are referenced directly.

that are expected to exchange data (which is determined by there being a data flow between the corresponding service instances in an orchestration).

SENSEI processors (Section 9.1) may only require a subset of these informations. Standard mechanisms for data formats, representations, and protocols may be imposed on tool developers. For example, a particular implementation may require that all data is expressed in the Graph Exchange Language GXL [Holt, Winter, and Schürr, 2000] (format), read from and written to corresponding text files on the local hard disk, or made available on an FTP server, e.g. in a distributed setting (protocol), passed into components as JAXB objects, or Service Data Objects [Resende and Feng, 2007] (representation). A plethora of technologies already exists, each covering one or more of these aspects of handling data. SENSEI makes no stipulations in this regard, so implementations may choose freely which to utilize, prescribe for use by components, or leave open for transformers to handle in case of a mismatch.

In addition, a fixed mapping of catalog-defined data structures to standard data models may be used, using SEON (*Software Evolution ONtologies*, Würsch et al. [2012]), for example. These data integration issues were left out of the scope of SENSEI, intentionally, and it was designed in a way that allows it to be complemented with existing solutions in this domain. As has been stated before, SENSEI nevertheless will work fine without a fully harmonized data model standard or ontology for the whole field of software evolution, employing reusable transformers instead. Adopting a standardized data model may lead to tight coupling and obstruct evolution of individual components [Josuttis, 2007, p. 38-39].

The data definition mapping allows for a high degree of freedom for SENSEI processor implementations: On one end of the spectrum, they may choose to provide the full range of flexibility to tool developers, by making no fixed assumptions on any of the dimensions of data definitions. This will incur a higher amount of effort required, though in the long term, reusability of components, and transformers in particular, is expected to counterbalance this, at least partly. On the other end of the spectrum, all aspects may be prescribed with standardized means and models, saving effort and dispensing with data definitions entirely, but also considerably constricting component development and evolution.

## 12.2 Finding Compositions

The component registry completes the SENSEI metamodel: while only a *service catalog* is prerequisite to creating *service orchestrations*, complementing it with a *component registry* allows to generate integrated software evolution toolchains using SENSEI processors. The process can be broken down into two main stages: first, *finding compositions*, and then *generating composer code*. This section is about the former, i.e. the process of matching all service instances of an orchestration to suitable components, consider-

ing all constraints arising from required capabilities, restrictions, data flow connections, and data structures.

Generating composer code is bound to the execution semantics of orchestrations, which have already been described in Chapter 11. An implementation of a composition generator is described in Chapter 14. An overview of the service-component matching semantics that are the basis for implementing composition finders will be given in the following sections. Finding compositions can be broken down further into three steps, as done by Meier [2014a, p. 7]:

1. Checking *orchestration consistency* – whether the provided orchestration satisfies certain prerequisite conditions, such as being free of contradictions that would make it impossible to execute under any circumstances (Section 12.3), e.g. when data flows connect ports typed with irreconcilable data structures.
2. Checking *component availability* – whether there are components in the registry for all the service instances in the orchestration, considering all required capabilities (Section 12.3.1).
3. Checking *component compatibility* – whether the components implementing the service instances in the orchestration implement ports connected by data flows with compatible formats and technologies (going beyond conceptual data structures and comparing actual implementation data types mapped to them by concrete components), and if not, whether data transformers can be inserted into the data flows to convert the data appropriately (Section 12.3.2).

Notice that the first steps analyzes orchestrations, only, the second step relates an orchestrated service instances and required capabilities to registered components and provided capabilities, and the third step operates on data definitions of components, i.e. on the registry level, only. The steps do not represent a defined order; the overall problem can be expressed declaratively. For example, if implemented in Prolog, as described by Meier [2014a], its backtracking algorithm will work through these steps, but potentially return to previous decision points in the solution space if it runs into an impasse.

If all checks are positive – and performed constructively – a composition results as a by-product (essentially a counterexample to the negation of the original constraint solving problem), and the composition generator takes over. The three steps are explained in the following.

## 12.3 Orchestration Consistency

The goal of the consistency check is to ascertain whether an orchestration is executable. This depends on data flows connecting ports, the compatibility of the associated data structures, taking into account all possible combinations of capability tuples on all service instances, and the restrictions, which in turn have an effect on data structures.

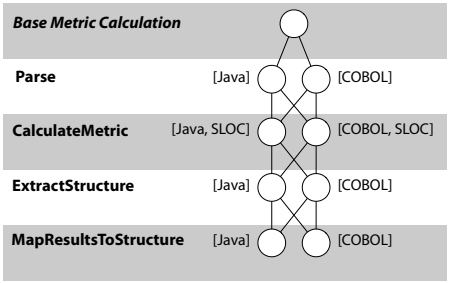



Figure 12.4: The base metric calculation example represented as a network of service instances with their capability tuples.

Orchestration consistency is also part of the semantics of service orchestrations, which were described in Chapter 11. However, consistency properties were only addressed briefly, implicitly, or not at all. This is because they become truly relevant in the context of service-component-matching, and are therefore discussed here.

The consistency check particularly relies on required capabilities specified on orchestrated service instances, data flows, and restrictions. Consider Figure 12.4: it depicts the service instances used in the orchestration for the base metric calculation example in a simplified manner. For every service instance and every capability tuple defined for it, a circle has been drawn. Because different capability tuples of the same service instance may be mapped to different components, a decision has to be made at runtime regarding the appropriate capability tuple, each time a service instance is to be invoked. The lines represent those decisions (not data or control flow).

Having an intuitive understanding of the capability tuples' semantics, it seems clear that only two ways through this network are sensible, the ones leading straight down: one represents the analysis of Java code, the other the analysis of COBOL code. Paths alternating between the two sides would make no sense, because, for example, a metric calculator for COBOL requires COBOL as input, and will not work if it is fed by a Java parser.

The objective of the consistency check is to come to this conclusion automatically, yet an automated mechanism does not have the intuitive insight described here. And while the matching capability names are no accident, formally, it must still be assumed to be purely coincidental: capabilities and capability classes are defined for each service individually, and may have completely different meanings. Data structures associated with the input and output ports of services are different, as their scope is the complete service catalog, i.e. if an output port type is *AST*, then any connected input ports must also be of that type, or a super-type.

Of course, the more capability tuples and service instances an orchestration contains, the more decision points there are. The declarative specification of required capabilities allows orchestrations to hide this complexity. In fact, even Figure 12.4 is

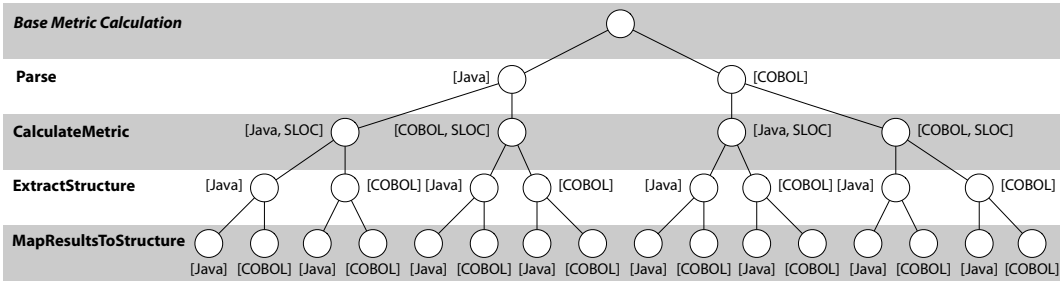



Figure 12.5: The complete decision tree, representing all possible orchestration trails through the base metric calculation example.

deceivingly compact. Expanding all possible decision points and their combinations leads to the concept of **orchestration trails**: paths through the network, determined by the decisions taken at every node. Depicting all possible orchestration trails results in decisions trees like the one depicted in Figure 12.5.

Using the concept of orchestration trails, consistency can be described more accurately: given input data at runtime, it should always be possible to determine the appropriate orchestration trail to take. If there is exactly one for all combinations of input kinds, the orchestration is *strongly consistent*. In practice, however, orchestrations may be designed in a way that not all combinations of input kinds actually make sense. Therefore, it is sufficient to have at least one input kind combination that has a valid orchestration trail; such orchestrations are *weakly consistent*. If no orchestration trails can be taken, ever, then the orchestration is said to be *invalid*. A special case arises if there are combinations of input kinds for which there are multiple orchestration trails, i.e. the orchestration is *underdetermined*. This might be due to insufficient information in the service catalog (missing restrictions), or there might be cases where, for certain data port allocations, two or more capability tuples become interchangeable. The former case represents modeling errors on the part of catalog maintainers, while the latter case may not be an error at all. However, it introduces non-determinism, i.e. the choice made at runtime is implementation-specific (and may be random).

Checking consistency therefore boils down to testing all possible combinations of input data kinds of an orchestration, each leading to a selection of capabilities, which may restrict the data types of some ports, which may select capabilities of service instances connected by data flows – and so on. The following concepts are used to further describe the composition finding process for implementation by SENSEI processors like the *CompositionFinder* [Meier, 2014a], that perform these consistency checks automatically:

**Restriction Consistency.** Service restrictions (Section 10.3) are defined in the service catalog, and define relationships between the data structure types accepted on

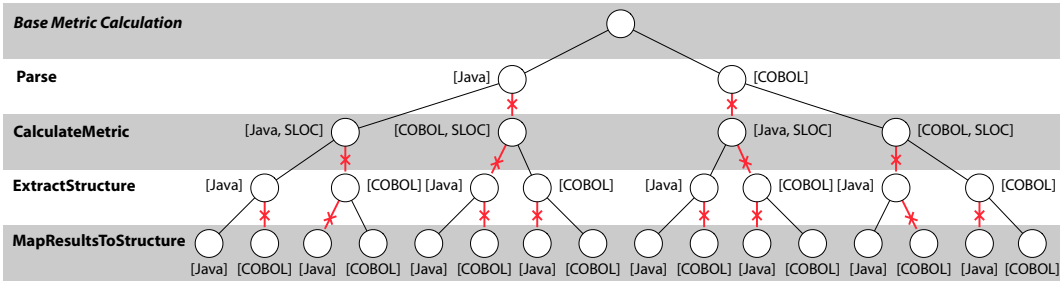



Figure 12.6: The decision tree of Figure 12.5, with paths whose ports are not assignment compatible crossed out, leaving only two valid orchestration trails.

service parameters and the capabilities the service defines. Inherently inconsistent restrictions on a service, i.e. containing contradictions are considered an error on the part of catalog maintainers.

**Assignment Compatibility.** This property actually has been introduced before, but with a "preliminary" definition. In the formal model, it is derived with mathematical rigor to reason about data structure compatibility, and hence the compatibility of data ports connected by flows, taking capabilities and restrictions into account.

**Activation.** This property captures the idea of capability tuples "triggering" associated restrictions, and provide a means to obtain *restricted data structures* of service parameters, given particular capability tuples.

**Simple Orchestration** Finding valid compositions is mainly constrained by relations between data structures, capabilities, restrictions, and data flows, while control flow can be ignored (assuming some limiting conditions). Simple orchestrations are basically orchestrations stripped of their explicit control flow constructs.

**Orchestration trail** At runtime, a single capability tuple has to be selected from each service instance in an orchestration. This induces a decision tree; each path from root to one of the leafs is called an orchestration trail, representing a possible combination of capabilities invoked at runtime.

As an illustration of what this process achieves, consider Figure 12.6: it depicts the decision tree shown earlier again, but with those paths, where the choice of capability tuples would lead to assignment incompatibilities on some data flow-connected ports. In this example, the kind of input data (either Java or COBOL source code), which is associated to capabilities via restrictions, allows to unambiguously choose exactly one capability tuple for each service instance, i.e. an orchestration trail, as elaborated earlier.

Having established orchestration consistency, the next steps are to check for component availability (Section 12.3.1), and then component compatibility (Section 12.3.2).

### 12.3.1 Component Availability

In terms of the SENSEI metamodel layers, the first step in finding compositions was constrained to service catalogs and service orchestrations. Orchestration consistency is a necessary precondition, but it does not establish any actual links between orchestrated service instances and service-implementing components. The second step of the process, component availability checking, does just that.

For this, the component registry layer of the metamodel has to be taken into account. In the end, this step yields a *matching function* that assigns concrete candidates for components to be used for each orchestrated service instance, based on the registry, and required and provided capabilities. In addition, superfluous capability tuples are filtered out: Theoretically, an orchestration could require capability tuples that are not part of any valid, executable orchestration trail. For example, the base metric calculation orchestration requires parsing capabilities for Java and COBOL. It would be possible to further require that the *CalculateMetric* service be able to also work on another programming language, e.g. Scala. Since Scala cannot be parsed, this capability to calculate metrics on Scala will *never* be reached, and therefore, no component needs to be mapped to it.

Through the matching function, concrete sets of components are given that provide all the capabilities required by an orchestration. However, this does not necessarily yield a valid composition: Orchestrations define data flows, along which the components matched to service instances and their capability tuples will need to exchange data. While orchestration consistency established that connected service ports are compatible on a conceptual level, introducing components and their *data definitions* requires to ensure compatibility on a technical level. To return to the base metric calculation example used before, a Java parser component is always expected to produce a Java AST, and a Java metric calculator needs to be able to consume it, by definition (in the service catalog), but the former might actually produce an XML file, whereas the latter expects a particular binary format. Selecting a candidate component for each service instance in such a way that this technical compatibility holds is done in the last step of the composition finding process.

### 12.3.2 Component Compatibility

The components selected for a composition need to be compatible in terms of the data formats they use to represent the data they need to exchange as specified by data flows connecting ports of the corresponding orchestration's service instances. Given a set of candidate components, even if data formats on connected ports are not directly compatible, it may still be possible to form a working composition out of the components,

because SENSEI has *transformers*. If there are transformer services defined – and corresponding components are available – that can translate one service's output format into the required input format of another, data compatibility is satisfied. It is even possible to chain multiple transformers to achieve the desired effect, and the formal SENSEI model accounts for this.

A transformer service is simply a service with exactly one input parameter and exactly one output parameter, with the additional requirement that the input parameter must be of a subtype of (or of the same type as) the output parameter. This relation must be maintained under restrictions. The rationale is that transformers should not make any changes on the level of the data's semantics. The data structure cannot really be changed, except by widening the type (i.e. omitting details from the output which were contained in the input).

Technically, the substructure relationship between input and output parameter is a prerequisite to insert transformer service instances into data flows of orchestrations. In the formal SENSEI model, *transformer injections* are introduced, which modify an orchestration (preserving full assignment compatibility) by

1. removing an existing data flow between two ports,
2. adding a transformer service instance, and
3. adding two new data flows to reconnect the severed link via the transformer.

Using transformer injection, orchestrations that cannot be implemented directly, because no directly compatible components are available, can be extended to yield a functionally equivalent orchestration. Therefore, a composition found for the new orchestration, i.e. one that also contains a match for the injected transformer, also satisfies the specification represented by the original orchestration.

If all three steps succeed, i.e. all constraints can be satisfied, then there is a composition of components available in the registry, that can be used to implement the given orchestration. Using constructive proof techniques, this process will also name the necessary components that together form such a composition, in the form of "witnesses" to the solution found. This is exactly what has been implemented in Prolog by [Meier, 2014a] in her CompositionFinder.

## 12.4 Summary

This chapter described the lowest level of the SENSEI metamodel, the component registry. Arguably, this is the simplest of the three layers, and can basically be described as a table that lists components together with the services they implement. However, together with SENSEI's capabilities, it forms the basis for a very powerful mechanism which allows to automatically map service instances in an orchestration to implementing components. As has also been explained in this chapter, this is not as trivial as

looking up components in the table that is the registry, because the way an orchestration is modeled, particularly in terms of required capabilities and data flows, constitutes a considerable amount of constraints that have to be satisfied in order for an overall composition of components to be properly executable.

Being able to automate this step is an important cornerstone of SENSEI, as it ensures that the process of specifying toolchains in terms of services remains free of implementation issues. Without this, domain experts would be forced to descent into the more low-level, technical world of components, would need to consider individual data formats, and pick transformers to manually arrive at a valid composition, impeding the ability to make changes quickly, and without knowledge of implementation technologies used by tools or for integration.

Instead, the SENSEI approach allows to keep service level implementation-agnostic, and facilitates the provision of working, integrated toolchains implementing the specified functionality by the press of a button. The feasibility of this approach to automatically match service orchestrations with component compositions has been shown by Meier [2014a] in her thesis, supervised by this author. Her Prolog-based CompositionFinder has been successfully applied to find compositions for SENSEI models, and it integrates itself as an additional *processor* (Section 9.1) in the overall architecture. While the toolchain-generating processor SCAffolder (Chapter 14) initially relied on a simplified, internal composition-finding algorithm, the alternative interpreter solution SNOrcInS (Section 14.5) fully relies on the CompositionFinder.

CHAPTER 13

# The SENSEI Editor

SENSEI is built around its metamodel (Section 9.2), which defines the structure and interrelationships between its three main artifacts: service catalogs (Chapter 10), service orchestrations (Chapter 11), and component registries (Chapter 12). Service catalogs are filled by catalog maintainers, who model services and potential capabilities relevant to a given domain. Domain experts then choose services along with required capabilities from such a catalog, and coordinate these service instances into orchestrations, which represent the processes to be automated by integrated toolchains. Tool developers fill component registries, also referring to services from a catalog, to establish which of their tools, wrapped as SENSEI components, implement which services, and provide which capabilities. These three kinds of artifacts together form SENSEI models.

The SENSEI metamodel also serves as the data model for storing these artifacts, but only its abstract syntax (see Section 8.3). To actually be able to create and modify SENSEI artifacts, a concrete syntax is needed to depict them, and, more importantly, an appropriate editor that enables its users to view and manipulate SENSEI models. In line with the objectives of this thesis (Section 1.2), particularly to increase productivity, such an editor needs to exhibit a reasonable level of usability. Very small SENSEI models can be created with little tool support, e.g. using a programming interface, or a simple text editor. This approach was used during early testing and prototyping of SCAffolder and the CompositionFinder (Chapter 14), but it quickly becomes infeasible for anything but the most trivial models. More to the point, this would be completely inappropriate for domain experts, who are generally neither proficient nor comfortable with such low-level, technical interfaces.

At the same time, domain experts are expected to model orchestrations, which, on the metamodel-level, are the most complex of the three artifacts. This emphasizes the need for a clear and self-explanatory concrete syntax. A graphical language is highly desirable, because SENSEI orchestrations have potentially complex control flow and

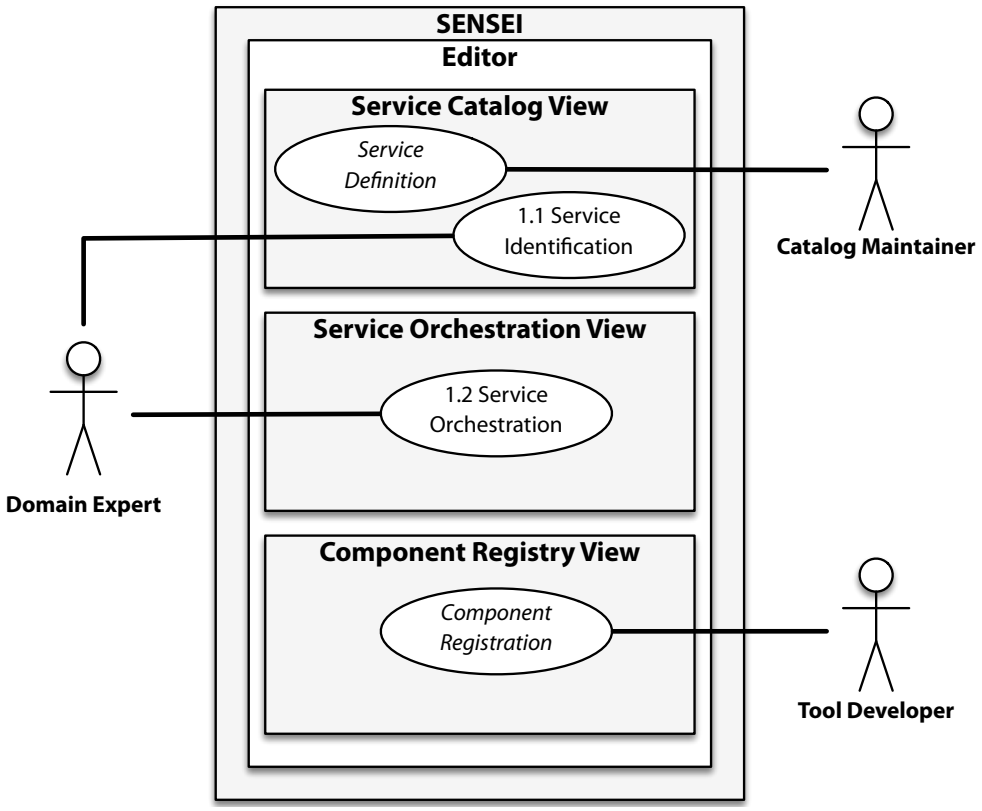



Figure 13.1: Use cases addressed by the SENSEI editor.

separate data flow. A textual notation requires to serialize all contained information in some form, making important relationships much less apparent, whereas a visual representation provides a natural immediacy. In fact, Chapter 11 already established a graphical concrete syntax for service orchestrations.

An editor for SENSEI models needs to address the concerns of three different user roles – catalog maintainers, domain experts, and tool developers, and the corresponding artifacts they are interested in modeling. So actually, SENSEI requires three editors, one for editing service catalogs, one for orchestrations, and one for component registries. Having to use different editors could be a productivity impediment, though, if there are users who take on more than one role in a project, e.g. domain experts who occasionally also act as catalog maintainers to add definitions for highly project-specific services, or technically versed domain experts who also take on responsibilities of tool developers by creating project-specific tools, or wrapping existing ones in SENSEI-compatible components. In small projects, in particular, this would be a very common

case. Also, domain experts need to browse service catalogs during the service identification step of the toolchain-building process (see Section 3.1). Therefore, another way to address the individual user roles is a single, integrated editor that offers separate views for each of the artifacts, between which it is easier to switch. Figure 13.1 depicts the roles of SENSEI and their corresponding use cases for the SENSEI editor and its views (recalling Figure 9.5 on page 157; the numbered use cases refer to the corresponding toolchain-building process steps).

In summary, proper editor support is viewed as a necessary prerequisite for applying and evaluating SENSEI, and an essential part of the overall approach, in general. The metamodel-centric nature of SENSEI facilitates the use of model-driven approaches to building editors, which are able to automatically generate the basic infrastructure for such applications. This has the potential to significantly reduce the manual implementation effort required, and therefore supports rapid prototyping. The editor developed as part of this thesis is meant to 1) contribute to the evidence of SENSEI's practicability, and 2) enable its application to non-trivial, practical use cases.

In the following, this chapter will motivate fundamental design decisions by comparing different approaches and available model-driven techniques for building an appropriate editor in Section 13.1. Next, Section 13.2 describes the SENSEI editor implementation. An example of how the integrated editors are used in practice is given in Section 13.3, before concluding the chapter with a summary in Section 13.4.

## 13.1 Technology Evaluation

Based on the requirements for a SENSEI editor described before, and a pilot study performed by Meier [2014b], it was decided to use *Eclipse Sirius* [*Sirius* 2020] as the basis for its development. This section briefly describes the core concepts of Sirius, and then discusses alternatives, and why they were considered less suitable.

### 13.1.1 Eclipse Sirius

Sirius allows the creation of visual DSLs and corresponding editor support in a declarative manner, based on an existing EMF [Steinberg et al., 2008] metamodel. While the SENSEI metamodel had been defined in the TGraph technical space ([Ebert and Franzke, 1995]; also see Definition 8.7), both EMF and TGraph metamodels can be specified using MOF-subsets (*Ecore* and *grUML*, respectively), and there is a bridge between EMF and TGraphs implemented in JGraLab [Heckelmann, 2010].

A comprehensive documentation of Sirius can be found online [*Sirius Specifier Manual* 2020]. The core concepts are as follows: Editors are specified declaratively, by modeling *viewpoints*, *representations*, and *tools for manipulation*. References to

concepts in the underlying metamodel are established by OCL/Acceleo[1] [*Acceleo* 2020] expressions. The definition of one or more editors is contained in *specification projects*, which contain *specification models*. Specification models are viewed and edited through a tree view; a screenshot of the specification model defining the SENSEI editor opened in Sirius is shown in Figure 13.3(a) – it will be further explained in Section 13.2. Note that the screenshots in Figure 13.3 are *not* depicting the SENSEI editor itself, but the "meta-editor" of Sirius: It may be described as an editor to edit editors, a notion that can lead to some confusion.

The top-level elements in a Sirius specification model are *viewpoints*. They allow to define different views onto the same underlying metamodel, and therefore satisfy a major requirement for the SENSEI editor. Viewpoints contain *representations*, which are used to specify what model elements should be visualized, and how. Representations contain *mappings* that select elements from the model being edited, and define how they should be visualized in the editor. *Styles* allow to specify fonts, labeling texts, icons, colors, etc. to be associated with the representation of a particular model element. *Tools* are used to specify user actions to manipulate the model, e.g. a context menu entry to create new elements, or edit or delete existing ones. *Validations* are used to define model consistency constraints, and messages that are reported to users if they are violated. *Java extensions* register functionality implemented in Java to be used in Acceleo expressions. This can be used to do computations on the model which are too complex to be adequately expressed in OCL/Acceleo alone.

### 13.1.2 Alternative Language Workbenches

Sirius can be considered a *language workbench* (Fowler [2005a]; also see Section 8.6), a class of tools that help author DSLs and corresponding editor support. The nature of the SENSEI metamodel, the fact that it is pre-existing, and the desire to use a *graphical* language for specifying orchestrations, precludes some language workbenches from being used from the outset. E.g., language workbenches aimed at textual DSLs, such as *Xtext* [*Xtext - Language Engineering for Everyone!* 2020], are not applicable. Also, frameworks which can use the MOF-based SENSEI metamodel more or less directly are to be preferred. This mostly excludes internal DSLs, i.e. languages that use a general-purpose programming language as host to utilize existing tool support. A similar approach that remains are UML profiles, as the host language is a graphical one, and UML lives in the technical space of MOF.

Meier [2014b, p. 26] follows Kühne and Wetzel [2006] in distinguishing these approaches as *heavyweight* (creating DSLs from scratch), and *lightweight* (extending an existing language, particularly using UML profiles). For evaluating the lightweight route, *IBM Rational Software Architect* and its UML profile-building facilities were

[1] *Acceleo* is an implementation of the OMG MOFM2T standard [*MOFM2T 1.0* 2008], which contains an OCL dialect.

chosen. Tools considered for the heavyweight approach included *MetaEdit+* [Kelly, Lyytinen, and Rossi, 1996; MetaCase, 2020], *Microsoft DSL Tools* [Microsoft, 2016], *JetBrains Meta Programming System* (MPS) [Pech, Shatalin, and Völter, 2013], and *Eclipse EMF* and *GMF* [*Graphical Modeling Framework* 2020; Steinberg et al., 2008], choosing the latter.

Using both approaches and their corresponding tools, the same, very small and simplistic graphical editor was realized [Meier, 2014b, pp. 28-36]. The results were then compared according to the following criteria:

- quality and extent of the tools' *documentation*,
- *customizability* of the editors to SENSEI-specific needs,
- *visual adequacy* of the concrete graphical syntax,
- *complexity and effort* required for the example's realization, and
- *licensing* issues that may limit the way the respective tools can be used.

Except for complexity and effort, using EMF/GMF was deemed the more appropriate choice regarding these criteria. The limited ability of creating editors that are truly tailored to the specific needs of SENSEI with UML profiles was found to be unsatisfactory, and to outweigh the higher implementation effort necessary for a GMF-based solution.

The study concluded with a successful prototype implementation of the editor, but also found that the EMF/GMF approach made it hard to extend [Meier, 2014b, p. 122]. GMF also exhibits rather poor, convoluted usability [Kolovos et al., 2010, 2009; Wienands and Golm, 2009]. In a comparison of five different, model-driven frameworks for tool/editor-building [Kouhen et al., 2012], GMF was attested the highest level of customizability, but was found to require by far the highest effort, and its user interface scored low in visual coherence.

Another tool that was seriously considered as the basis for the SENSEI editor was *MetaEdit+*. In the study by Kouhen et al., it was found to have a remarkable lead in terms of productivity over its competition. MetaEdit+ was found to be a very powerful alternative, with a solid, well thought out foundation at the meta-metamodel level. However, because of this, it also represents a technical space that is markedly distinct from MOF-based solutions. Its adoption would have required the development of a bridge between MetaEdit+ and the TGraph technical space, so that the JGraLab-based SCAffolder would be able to use SENSEI models produced with editors created with MetaEdit+. Because of this, MetaEdit+ was ultimately not chosen.

The usability and productivity issues associated with GMF have spawned new frameworks in the MOF technical space, aimed at simplifying the editor-building task. These include *Graphiti* and *Sirius* [*Graphiti* 2020; *Sirius* 2020]. The former builds on the more low-level Eclipse *Graphical Editing Framework* (GEF), the latter directly on GMF. At the time of the study by Meier [2014b], *Eclipse Sirius* had only recently been open-sourced, and had not yet gained enough visibility, which is why it was missed and was not made part of the evaluation. Of the two, Sirius was chosen over Graphiti,

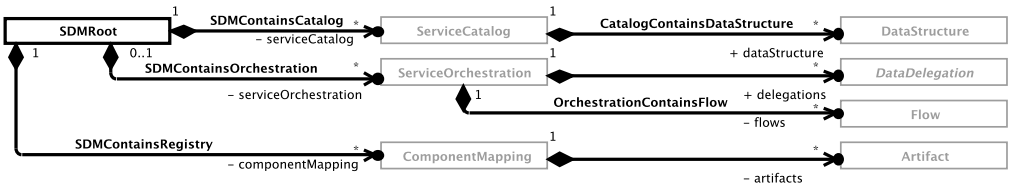



Figure 13.2: Elements added to the SENSEI metamodel to make it Ecore-compatible.

based on experience reports and comparisons [Strittmatter et al., 2016; Vujović, Maksimović, and Perišić, 2014], and because it seemed more approachable.

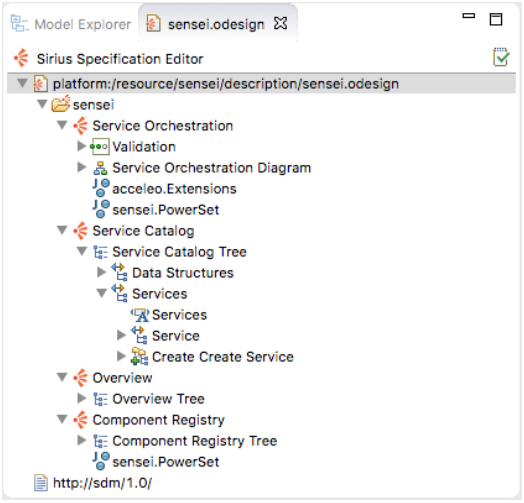
## 13.2 SENSEI Editor Implementation

Since the Sirius framework is rooted in the EMF technical space (Section 8.7), some additions had to be made on the metamodel-level, to account for some limitations of Ecore. This is briefly sketched in Section 13.2.1, before Section 13.2.2 describes how the SENSEI editor was implemented using Sirius.

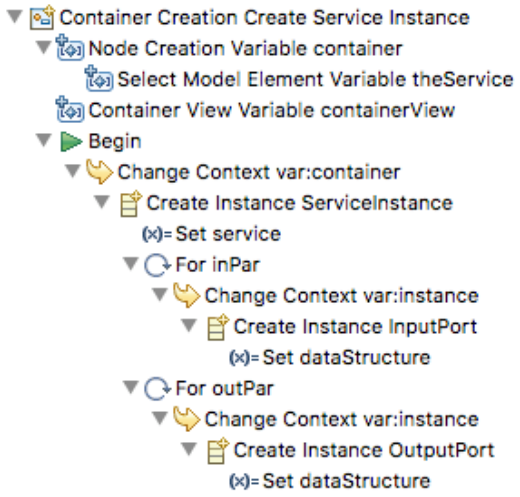
### 13.2.1 Metamodel Extensions

To keep using the existing SENSEI metamodel as a single point of truth and basis for both SCAffolder and associated meta-tools, as well as the SENSEI editor, it was slightly extended to make it Ecore-compatible. The additions to the model are shown in Figure 13.2. Because Ecore only supports tree-like models, its metamodels must define a root. This role is played by the newly added *SDMRoot* metaclass. Furthermore, all model elements besides instances of *SDMRoot* must be contained in other model elements because of the tree structure. This is modeled using composition associations. For several model elements, such containment relations are semantically natural and were therefore already present in the SENSEI metamodel. For other elements, this was not necessary, at least not explicitly. For Ecore compatibility, explicit containment relationships had to be added in these cases: three composition associations connect SENSEI's "top-level" concepts to the root metaclass, and four additional relationships were required to associate further metaclasses with their containers. This is essentially adding redundant information to SENSEI models.

The SENSEI metamodel (in XMI format), extended in this fashion, remains to be used as the source for the JGraLab generated API. The Ecore metamodel is not derived from the XMI representation directly, but rather from the TGraph schema produced by




(a) Sirius specification editor.




(b) Service instance creation tool.

Figure 13.3: Screenshots of Eclipse Sirius, and the viewpoint specification project that defines the SENSEI editor.

JGraLab. This ensures that SENSEI models in Ecore format (instances of the Ecore metamodel produced this way) can be converted into TGraphs conforming to this schema. The converter itself also had to be adapted. As pointed out by Meier [2014b, pp. 88-90], the Ecore-to-TGraph bridge that is part of JGraLab only allows to transform a model together with its metamodel in either direction. However, this is mostly an issue of an unfortunate interface design. A modified version of the Ecore-to-TGraph converter, that allows to provide a pre-existing metamodel (TGraph schema), and only run the transformation on the model level, has been developed. SDMTGraph, a Java class library containing an API generated from the SENSEI metamodel using JGraLab, also provides the *ECoreIO* class as facade to this converter, which in turn is used by SCAffolder to load SENSEI models in Ecore format. The generation of the Ecore metamodel can be triggered using the *xmi2tg* Maven plugin: with the property *ecore* set to true, it will output an Ecore metamodel file in addition to the TGraph Java API. The build process of the SDMTGraph Maven project is set up to do just that.

The resulting Ecore metamodel is the basis of an EMF project. Similar to JGraLab, EMF generates Java code that provides an API to work with instances of the metamodel. The EMF project and the code it generates are the foundation for the SENSEI editor.

### 13.2.2 Implementation with Sirius

This section describes how the SENSEI editor was modeled with Sirius, using the concepts given in Section 13.1.1. This is meant to convey the general idea of its inner

workings, and not as a full reference of the SENSEI editor implementation. The features of the SENSEI editor are described in Section 13.3 from a user's point of view, instead.

The Sirius *specification model* defining the SENSEI editor is contained in a specification project. A screenshot of the model, opened in Sirius, is shown in Figure 13.3(a). For the SENSEI editor, four viewpoints are defined. Three of these correspond to the main layers of the SENSEI metamodel (*service catalog*, *service orchestration*, and *component registry*), and therefore also to the views presented initially in Section 9.4, addressing the different tasks of catalog maintainers, domain experts, and tool developers. For practical reasons, there is an additional *overview* viewpoint, used, for example, to create or delete orchestrations. This is not possible in the orchestration view, because it is bound to a single service orchestration. The need for such an additional editor was already recognized by Meier [2014b, pp.84-85, 108–109]; the first prototype included the similar *orchestration list editor*.

Each viewpoint has a single representation definition: service orchestrations are represented graphically, specified by the *service orchestration diagram* representation. Each of the other viewpoints contain a tree-based representation. As an example of how the metamodel elements are mapped to graphical elements in the editor, in Figure 13.3(a), the service catalog tree representation is expanded to reveal its two top-level mappings that are used to group all data structures and services, respectively. Nested within the latter are three more nodes: the first, also labelled "*Services*", contains *style* definitions, e.g. font, a labeling text, an icon, colors, etc. The second element is another mapping, which contains an OCL *candidate expression* that selects all services in a given catalog: `self.services->sortedBy(name)` (`self` refers to an instance of *ServiceCatalog*). At runtime, this creates a node in the tree view for each service returned by that expression. The last element defines a *tool* to create new services in the model.

There are also several *Java extension* nodes, such as *sensei.PowerSet*. They register functionality implemented in Java to be used in Acceleo expressions. For example, the class *PowerSet* contains methods that return all possible capability tuples for a given service instance (excluding the ones already defined on it, if that is desired). They are used to present the user with a selection dialog when creating new capability tuples in an orchestration, or for a registered component.

Figure 13.3(b) shows a subtree of the specification model that defines the tool for creating service instances in the orchestration view. The child nodes at the top represent *variables* that provide context, e.g. the variable `container` refers to the parent model element – it could be the orchestration being edited itself, but also a control flow construct nested within it. The *Select Model Element Variable* has the effect of presenting a dialog to the user when using the tool, which allows him to select a service from the catalog that should be instantiated in the orchestration.

The *Begin* node is used to define what changes to the model should be made in response to the user having selected and used this tool, using *model operations*. *Change Context* essentially decides to what element the `self` keyword will refer to in all ex-

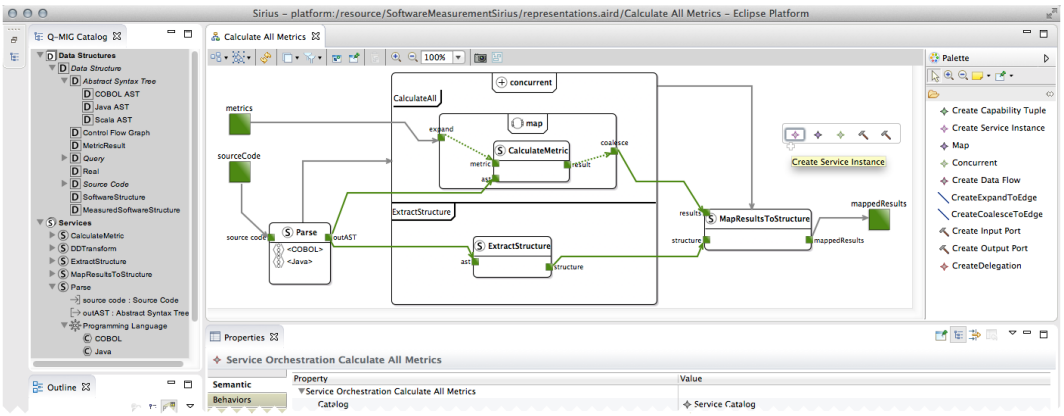



Figure 13.4: Screenshot of the integrated SENSEI editor views with a model of the base metric calculation example being opened. The service catalog view is on the left, and the graphical orchestration view is seen in the center.

pressions of elements nested below. The *Create Instance* node adds a new *ServiceInstance* object to the model. Its nested *Set* operation assigns a reference to its *service* attribute (to the service specified by the user). Then, there are two *For* loops. They iterate over the service's input and output parameters, respectively, and create corresponding ports for the service instance. This way, the user does not have to create them manually, and by not allowing to manually modify ports, at all, the model is ensured to always remain consistent in this regard.

While these are just a few examples, the overall SENSEI editor is implemented along these principles, and, save for a few snippets of Java code, fully defined using the declarative model of Sirius. The following section describes the resulting editor, and how to use it.

## 13.3 Using the SENSEI Editor

Sirius produces editors that can be deployed as Eclipse plugins, or as standalone, Eclipse-based applications. A screenshot of the SENSEI editor is shown in Figure 13.4. On the left, the service catalog view is open and shows the contents of the *Q-MIG Catalog* (Chapter 15) contained in the currently opened SENSEI model. The window in the center is the orchestration view, showing the *Calculate All Metrics* orchestration. Part of this view is the tool palette on the right. At the bottom is the properties window, which shows attributes and their values of elements selected in the editor window currently possessing the focus. In this screenshot, the component registry view is not open, but it will be shown later in this section. All the basic features of the underlying

Eclipse rich client platform are available, so for example, all these windows can be rearranged, docked to a side, hidden in a tray, or maximized.

Using the *base metric calculation* as a running example (see Section 2.2), the use of the SENSEI editor is explained in the following. Section 13.3.1 begins with describing how to create and configure a modeling project, needed as a basis for all tasks of the SENSEI-supported toolchain-building process. Then, the functions of each of the main views are explained, starting with the service catalog view (Section 13.3.2), followed by the orchestration view (Section 13.3.3), and concluding with the component registry view (Section 13.3.4).

### 13.3.1 Creating SENSEI Modeling Projects

As an Eclipse-based application, the SENSEI editor has the same concepts of *workspace* and *projects* as the well-known IDE has. So, to get started, a *modeling project* must be created. If this project type is not directly available in the menu (*File → New*), selecting *Other* will open a dialog with all available project types to choose from. The modeling project is in the *Sirius* folder. The newly created project contains only a single file called *representations.aird*.

Next, a SENSEI model should be created in the project. The corresponding wizard is found in the *SENSEI* folder and called *SDM Model*[2]. It will create a new file in the project. Its corresponding node in the model explorer can be expanded to reveal that it already contains an *SDM Root* element, which in turn has a single *component registry* and a *service catalog*.

By default, the viewpoints defined in Sirius for SENSEI models are all disabled. They can be toggled in the project's context menu by clicking on *Viewpoints Selection*. Users can focus on the task associated with their roles – catalog maintainers, domain experts, or tool developers – by selecting only the appropriate viewpoint. It is also possible to activate multiple or all viewpoints at once, to view and edit all aspects of a SENSEI model. The selected viewpoints are shown as child nodes of the representations.aird file. New representations, e.g. orchestration diagrams, will appear here.

To start filling the model, *representations* are needed. Once the viewpoints are activated, the associated representations become available in the project's context menu under *Create Representation*. The wizard will ask to first select a representation, such as a service catalog tree, and then associate this with a model element, such as a *ServiceCatalog* instance.

Note that representations can only be created for already existing model elements. Since the initial model does not contain an orchestration, no corresponding diagram can be created, either. To create orchestrations, an overview tree representation should

[2]This is derived from the commonly used file extension, *sdm*, which used to stand for *Service Description Model*. The models are now simply referred to as SENSEI models, but the abbreviation of the earlier moniker stuck.

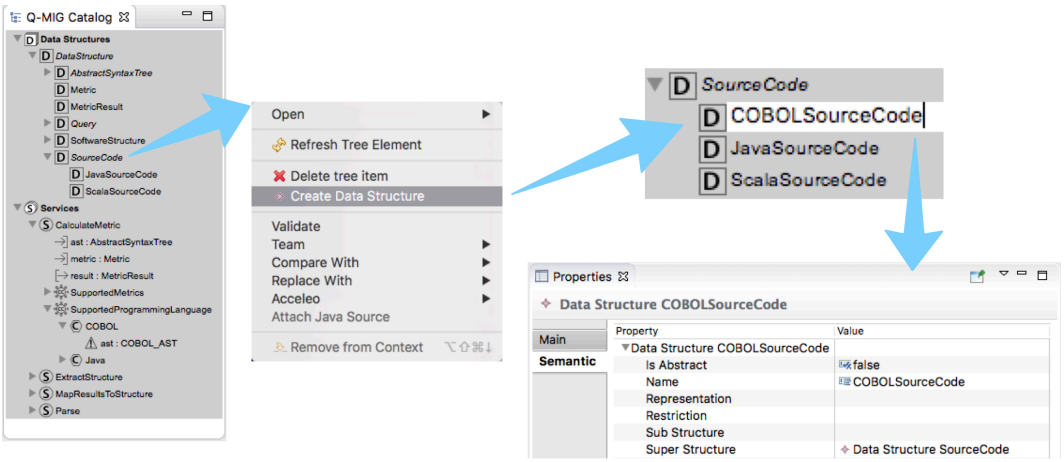



Figure 13.5: Creating and editing data structures.

be created for the *SDM Root* element. The overview editor makes the *Create Service Orchestration* action available to the user from the context menu of the model's service catalog. Once the service orchestration has been created in the model, a diagram representation can be created in a second step.

### 13.3.2 Defining Services

When starting with an empty SENSEI model, the service catalog has to be filled first, as orchestrations and the component registry depend on it. This is the job of catalog maintainers, and supported by the service catalog view.

The tree editor for an empty service catalog shows only two nodes: *Data Structures* and *Services*. They do not correspond to any model elements themselves, but serve to organize the contents of the catalog. The tree editor can be used to *create* and *delete* elements through context menu actions. To *update* attribute values, a model element is selected in the tree, which can then be edited in the properties window. The name of most element types can alternatively be edited directly in the tree editor by double-clicking it.

The left-hand side of Figure 13.5 shows a screenshot of a service catalog opened in the editor. The contents of the catalog were created for the base metric calculation example. At the top, the data structures are shown. They appear nested in the editor according to their sub-structure relation. In this catalog, there is a single, top-level data structure – called *Data Structure*. This is not necessary, in general, but it can be useful to have a single, top-level type that all others derive from (like *Object* in Java).

Figure 13.5 illustrates a sequence of actions: New data structures are created by

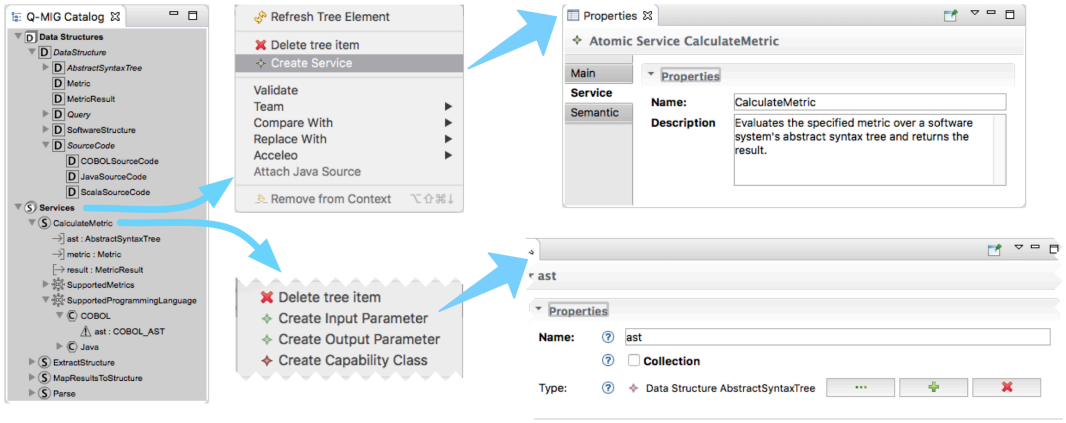


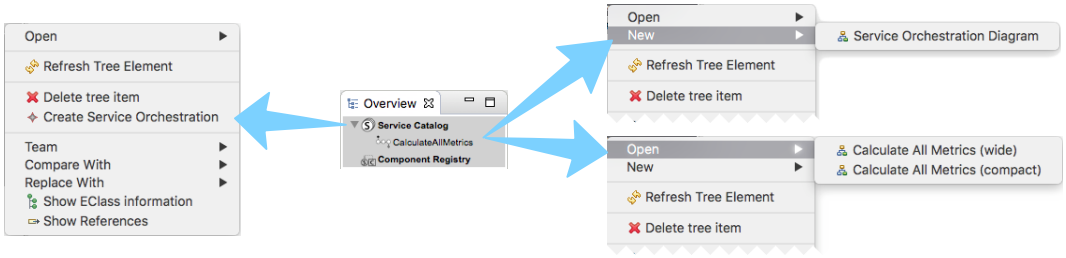

Figure 13.6: Creating and editing services.

right-clicking on either the *Data Structures* node, or on an existing data structure element, to create a sub-structure of it (center). In the example, a new sub-structure of *Source Code* is created, to represent COBOL code. The new data structure will appear immediately in the tree view, and its name can be edited (top-right). Further attributes can be viewed and modified in the properties window (bottom-right)[3].

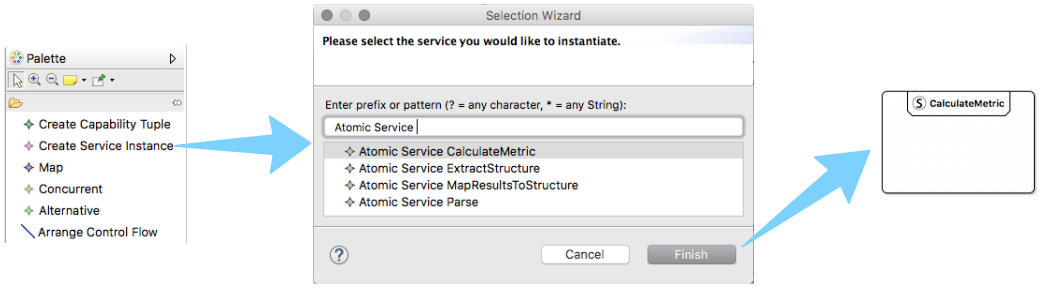
The sub-structure relationship is created automatically, as a result of invoking the context menu action on an existing data structure. Note that representation in the tree editor does not restrict the ability of data structures to have more than one super-structure, as is explicitly allowed by the SENSEI metamodel. A data structure with multiple super-structures will appear multiple times in the tree view. Each visual representation refers to the same, single model element though, as can be tested by renaming it through one node, and witnessing all other nodes representing the element instantly change, as well. Changes in one view are also immediately reflected in all other views: for example, if the *COBOLSourceCode* data structure where set to be *abstract* in the properties view, the catalog tree view would refresh to show the data structure node's name in italics to indicate this.

Creating new services follows basically the same mechanics. Examples are shown in Figure 13.6, with the properties view for services on the top-right, and the one for parameters on the bottom-right. The service only has two text properties, its name and a description. Parameters also have a boolean attribute *collection*, which is edited using a check box, and a reference to its *type*, which can be set by clicking on the "..."-

[3]Starting with Sirius 4.0, an alternative to the standard properties view is provided [*Sirius Specifier Manual* 2020]. The SENSEI editor was originally developed prior to the release of Version 4.0, but can still take advantage of the new features, at least with default settings. Examples of the Sirius-generated property views are shown, e.g. in Figure 13.6




(a) Creating orchestrations and associated diagrams.




(b) Creating service instances in orchestrations.

Figure 13.7: Creating orchestrations and service instances.

button. This opens a dialog that allows the user to choose an existing data structure from the catalog.

In the same manner, capability classes and their capabilities are created. Restrictions are created and shown as child elements of capabilities, and must be bound to a parameter (belonging to the same service) that is being restricted, and a data structure (that is a sub-structure of the parameter's type), using the properties view. With this, all concepts specified by the service catalog layer of the SENSEI metamodel are covered.

### 13.3.3 Modeling Orchestrations

With a service catalog in place, domain experts can start creating orchestrations, relying on the service catalog view to look up services, and, primarily, on the service orchestration view. Creating and editing orchestrations is significantly different from working with both the service catalog and the component registry, because of its graphical notation and associated diagram editor. The service orchestration view uses the syntax introduced in Chapter 11.

A new orchestration is created from the (very simple) overview editor, an example of which is depicted in the center of Figure 13.7(a). The context menu of the service catalog node provides the user action to create orchestrations, which will appear as child nodes of the catalog in the overview editor. For orchestrations, a *representation* has to be created first. This is not necessary for either the service catalog or the component registry, since newly created SENSEI models always contain exactly one instance of each, and this cannot be changed. In contrast, a SENSEI model can contain an arbitrary number of orchestrations.

Representations of orchestrations are called *service orchestration diagrams*. They are created in the context of orchestrations (top-right of Figure 13.7(a)), and can then be opened from there (bottom-right). The diagram stores layout information, which the editor keeps separate from the SENSEI model, i.e. they do not get persisted to the SDM file, but to the *representations.aird* file of the modeling project. As shown in the figure, there can be more than one representation for the same orchestration. In the example, there are two different layouts, a wide and a compact one, as indicated by the names of the diagrams.

Opening the diagram of a newly created orchestration will show the orchestration view window with an empty canvas, and a palette of tools that can be used to create service instances and control flow constructs. For better clarity, data flow concepts are kept on a separate *layer* of the editor, which is disabled, initially. This also hides several tools in the palette associated with data flow. In the following, the creation of service instances and capability tuples is described first. Then, modeling control flow is explained, and finally data flow will be covered.

#### Service Instances

The palette drawer of the orchestration view is shown on the left of Figure 13.7(b). Selecting the *Create Service Instance* tool, and then clicking into the canvas, will open a dialog to choose a service from the catalog. When the user confirms his choice, a new service instance is created in the model, appearing as a rounded box in the editor, with a separate name compartment at the top. It shows the service symbol (an encircled "S"), followed by the name of the service this instance refers to.

A newly created service instance is not associated with any capability tuples. To add one, the corresponding tool from the palette can be used, first clicking the tool, and then the service instance to which a required capability tuple should be added. Alternatively, the orchestration view will show a balloon icon above service instances, when hovering the cursor over them. This makes the same tool accessible directly in context of a service instance; an example is shown in Figure 13.8 on the left.

When creating a capability tuple using either of these two ways, a dialog will appear next, showing all possible capability tuples for the service instance, except for those already present on it. In this example, a capability tuple is created on an instance of *CalculateMetric*. This service defines two capability classes: *SupportedMetrics* and

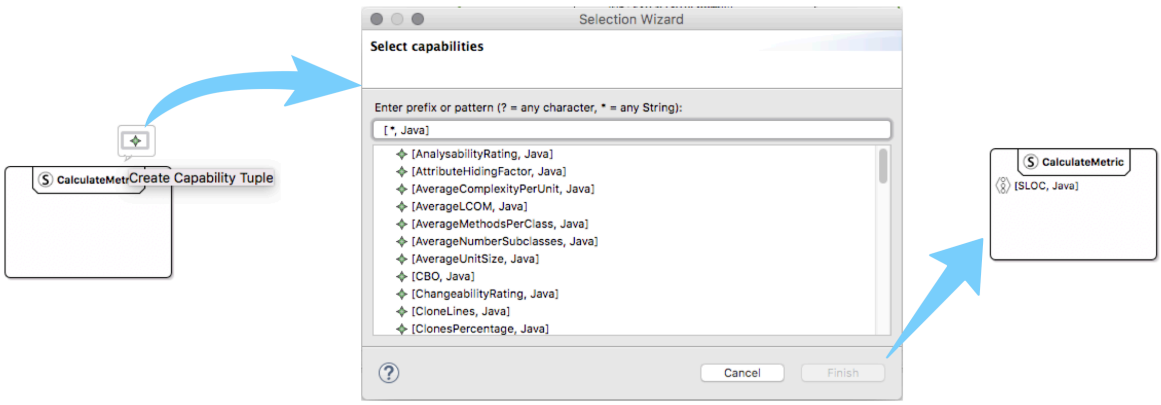



Figure 13.8: Creating required capability tuples.

*SupportedProgrammingLanguages* (see Figure 13.6). The classes have fifty-four and two capabilities, respectively, which results in a total of one hundred eight possible capability tuples. The list presented to the user is sorted alphabetically, and can very easily be filtered by entering parts of a capability name, for example, or using simple wildcards (as shown in the center of Figure 13.8). The required capability tuples should therefore be easy to find and select. Furthermore, the editor allows the user to select multiple tuples in the dialog, so that all required capability tuples can potentially be added at once, without the need to repeatedly select the tool and choose from the dialog's list.

Required capability tuples are shown in the box that represents the associated service instance, one below the other. The size of service instance boxes can be adjusted as necessary, to fit all capability tuples. If the box is too small, a scroll bar will become visible, which can be used to browse through the list of all required capability tuples present on the service instance. Capability tuples can be removed by selecting them and either hitting the Delete key on the keyboard, clicking on *Delete from Model* in the toolbar at the top of the editor, or by opening the context menu and selecting *Edit →Delete from Model*. This is actually the same for nearly all model elements represented in the orchestration view, including service instances.

### Control Flow

As soon as a second service instance is created in an orchestration, gray control flow edges are created automatically to put them in sequence. Multiple service instances in the same container are always in a defined order – the orchestration view does not allow service instances not connected into a control flow sequence. Control flow edges are therefore not added manually, but can only be *rearranged*. Existing edges can be

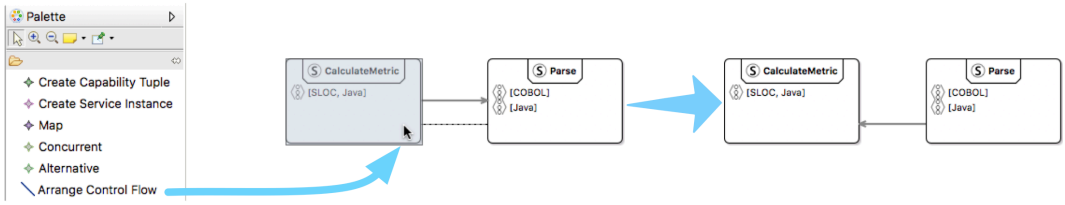



Figure 13.9: Rearranging the sequence of control flow.

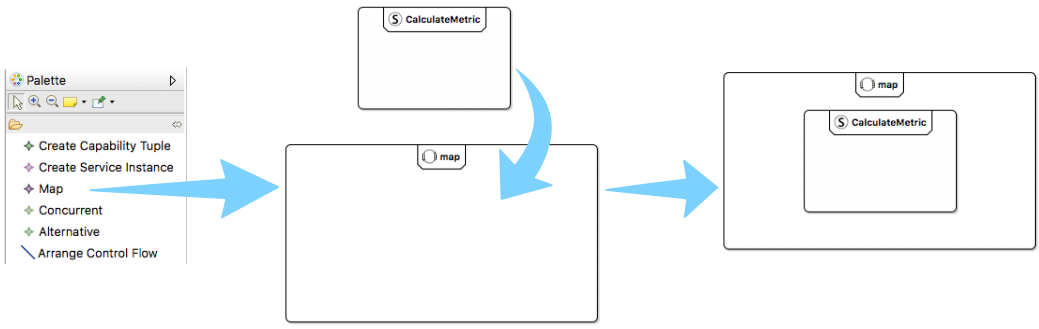



Figure 13.10: Creating control flow constructs and nesting service instances within them.

reconnected, by grabbing either the start or end of a control flow edge, and dropping it on another service instance. This moves the service instance that stayed connected to the edge to become the new direct predecessor (or successor) of the service instance the edge' head (or tail) was dropped on.

Another way is to draw control flow edges to rearrange service instance in the control flow sequence. This is depicted in Figure 13.9: First, the corresponding tool is selected from the palette. Then, a line is drawn from the service instance that should become the new direct predecessor, to the service instance that should be moved to right after the former. In the example, the *CalculateMetric* service instance is moved, so that *Parse* will become its new predecessor in the control flow sequence.

Since control flow in SENSEI orchestrations is structured, branching and splitting the control flow is achieved by nesting service instances within control flow constructs. The three constructs, *map*, *concurrent*, and *alternative*, each have a tool in the palette associated with it. Figure 13.10 shows an example of creating a *map* construct. Once created, they appear in the orchestration view canvas as rounded rectangles, similar to service instances, but each featuring a different symbol, and the name of the construct instead of the service name at the top.

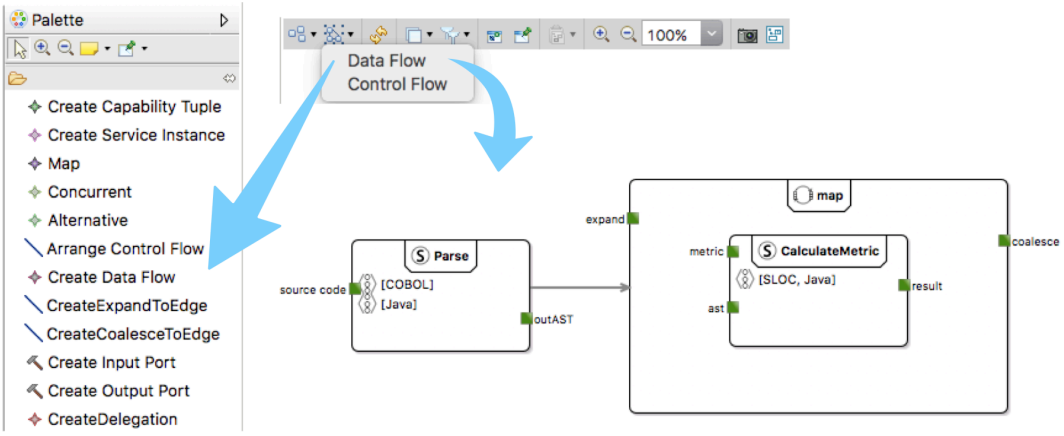



Figure 13.11: Enabling the data flow layer.

The body of control flow constructs represents a sub-orchestration. Service instances can be created within them, just like creating them within the orchestration directly. Existing service instances (or other control flow constructs) can be dragged and dropped into the bodies of constructs. The bodies of concurrent and alternative control flow construct is split into two lanes[4]. Each lane of an alternative control flow construct shows its guard in the top-left corner, which can currently only be edited through the properties view.

### Data Flow

So far, data flow aspects, in particular the ports of service instances, have been ignored. This is not an oversight: the SENSEI orchestration view is built so that data flow exists on a separate layer, so that ports and data flow edges can be hidden from view. This is useful because even small orchestrations can quickly become confusing with lots of data flows criss-crossing all over it. A large part of the functionality and coordination of service instances modeled by an orchestration can be understood without the data flow aspects, so for better clarity, it makes sense to only enable the data flow layer when it is actually needed.

Figure 13.11 shows the toolbar of the orchestration view on the top. It has several controls, e.g. to apply an auto-layout algorithm, or to zoom in or out. The availability of individual controls on the toolbar is context dependent – the figure shows the default state, when nothing is selected. Clicking the layers control opens up a menu to enable or disable specific layers. Activating the data flow layer lets all the data flow concepts appear in the orchestration, and makes additional tools available in the palette.

[4]The number of lanes is currently fixed in the editors, even though the SENSEI metamodel allows arbitrary many lanes. However, by nesting the control flow constructs, the same semantics can be expressed.

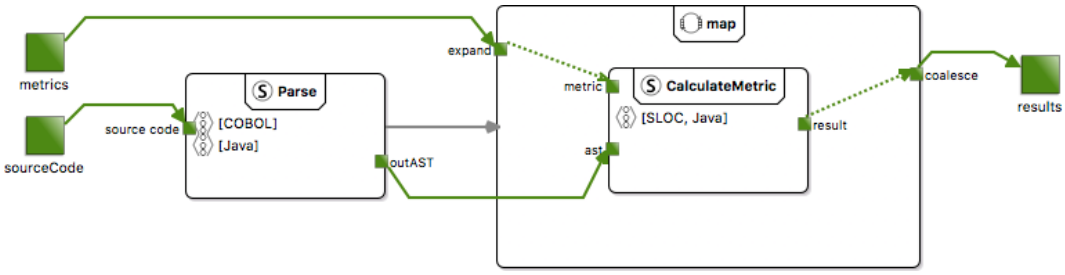



Figure 13.12: Orchestration view with the data flow layer activated, showing an orchestration with ports and data flows added.

As shown in Figure 13.11, all the service instances already possess ports corresponding to the parameters of their respective services. They are depicted as small green rectangles on the border of service instances, with a label next to it, containing the name of the corresponding service parameter. Ports do not have to, and in fact cannot be created manually. When creating service instances, its ports are created along with it automatically.

To connect ports to each other, the *Create Data Flow* tool from the palette is used. The data flow is created by simply selecting the tool, and then first clicking an output port as source, followed by an input port that should receive the data produced at the source. The new data flow edge will be shown as a green arrow connecting the selected ports. Bendpoints can be added by clicking and dragging the arrow, to route them however desired, e.g. to improve overall readability and clarity of the orchestration diagram[5].

There are several special kinds of data flow edges. The expansion and coalescence port of map control flow constructs require to be connected to ports of service instances nested within them by such special edges. To create them, the *CreateExpandToEdge* and *CreateCoalesceToEdge* tools are used. The resulting kinds of edges are drawn as green arrows with a dotted line, to distinguish them from normal control flows. This is to highlight that they, in fact, behave differently, as they either break up the input collection of data elements and carry each to their destination one by one, or carry individual data elements into a collection, filling the target rather then overwriting it in each step.

Another kind of data flows are delegations. They are used in conjunction with input and output ports of the orchestration itself. The latter *can* be created manually, using the *Create Input Port* and *Create Output Port* tools, respectively. Orchestration ports

[5]The data flow edges in the examples are drawn to approach their source and sink ports at an angle. This is to avoid overlapping with the port labels, which currently cannot be moved freely, but are always aligned centered on one of the sides of the square that represents the port.

are represented by green rectangles, larger in size than the ports of service instances. They can be positioned freely on the orchestration diagram canvas. To connect them to ports of service instances contained in the orchestration, the *CreateDelegation* tool is used. The difference here is that delegations connect either an orchestration input port to an input port of a service instance, or an output port of a service instance to an output port of the orchestration, whereas normal data flow edges always connect an output port to an input port. Delegations are drawn exactly the same as normal data flows, since apart from this difference, their semantics (copying the whole data element from the source to the sink) is exactly the same.

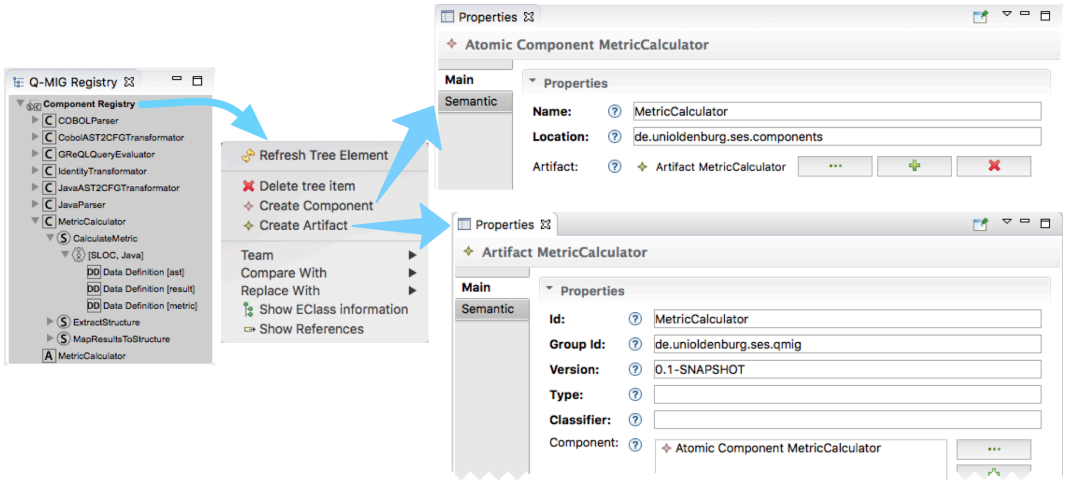
A screenshot of the orchestration view as an example of a simple orchestration with data flows added is shown in Figure 13.12. This is not yet the complete orchestration to model the base metric calculation, which was shown previously in Figure 13.4. However, all major functions of the orchestration view have been introduced, so that this intermediate result can now be extended to the full orchestration model by using the editing tools and controls described in this section.

### 13.3.4 Registering Components

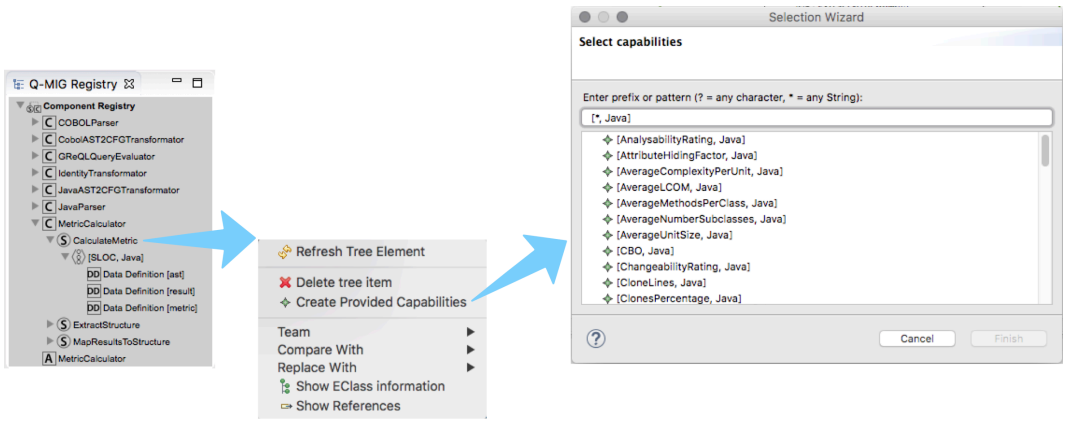
The component registry view is conceptually similar to the service catalog view, both being based on a tree view editor. This is what tool developers use to describe their tools in terms of SENSEI components, and establish links to implemented services (which can be browsed using the service catalog view). Figure 13.13(a) depicts the component registry view on the left. Under the root node, *components* and *artifacts* contained in the registry are displayed. Both types of elements can be created using the context menu of the root node (shown in the center of the figure), and selecting the *Create Component* and *Create Artifact* action, respectively. Their attributes can be viewed and edited using the properties view (right side of the figure). Components can be associated with the artifact in which they are contained. This relationship can also be viewed and edited from the opposite end, as the properties view of artifacts contains an editable list of contained components.

Once a component is created, it can be registered to implement services. This is done via the *Implement Service* action in its context menu, which leads to a selection dialog to choose from the services available in the service catalog – this is the same dialog used in the orchestration view when creating service instances there (see Figure 13.7(b)). A node representing an instance of the selected service is created as a result, as a child of the component.

The next step is to specify provided capabilities. This is done via the service instance's context menu action *Create Provided Capabilities*. An example is shown in Figure 13.13(b). The dialog that opens is, once again, a familiar one: it is the same as the one used in the orchestration view when specifying *required* capabilities on service instances in an orchestration (see Figure 13.8). Selected capability tuples will be




(a) Creating components and artifacts.




(b) Creating provided capabilities.

Figure 13.13: Creating artifacts, components, and provided capabilities of implemented services.

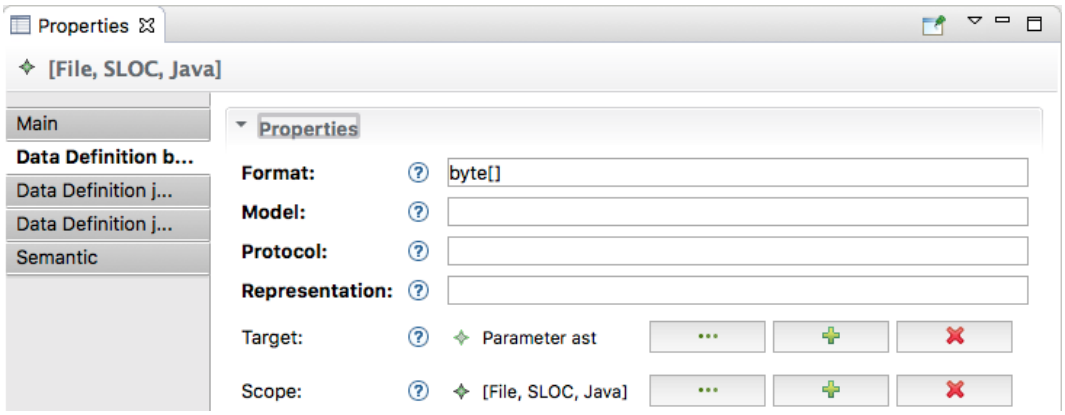



Figure 13.14: Properties of data definitions.

created as children of the current service instance node. In addition, all necessary data definitions are created automatically, as well: one for each parameter of the instantiated service will appear below each capability tuple node. Their attributes are edited as usual, using the properties view, which are also available as tabs when the corresponding capability tuple is selected in the component registry view. An example of this property view is shown in Figure 13.14.

## 13.4 Summary

This chapter gave an overview of the implementation and usage of the SENSEI editor. By providing three separate but integrated editors, each corresponding to one of the layers of the SENSEI metamodel, the tasks and concerns of catalog maintainers, domain experts, and tool developers are addressed individually, while still resulting in a single model that can be further processed, e.g. by the CompositionFinder, SCAffolder, and SNOrcInS.

Sirius has been found to be a more than adequate framework solution for building the editors. Creating the editor required far less effort than building the original editor prototype, even though it had a more limited feature set. It has also proved to be easily extendable, even in case of changes in the underlying SENSEI metamodel. Much more problematic than adjusting the editors in such cases were existing SENSEI models becoming incompatible – a problem which cannot fully be avoided.

There are some small limitations, like concurrent and alternative control flow constructs being fixed to two sub-orchestrations in the orchestration view, but this does not impact the expressiveness of the orchestration language available through the editor. There are also certainly several usability issues; one example is the rearranging

of control flow in the orchestration view, which can at times feel counter-intuitive and thus require a bit of trial and error to achieve the desired result. Just like SCAffolder, this is a research prototype, and thus product-level maturity cannot be expected, and was not the objective.

The SENSEI editor has been used extensively to create models for the evaluation scenarios which will be presented in Part V, and have proven their value. Also, they have successfully been used by Küpker [2015] during the course of his thesis.

CHAPTER 14

# SCAffolder: A SENSEI Toolchain Generator

The toolchain-building process that SENSEI aims to support is separated into two phases: *specification* and *implementation* (recall Figure 9.4 on page 156). The SENSEI editor supports all three SENSEI roles – catalog maintainers, domain experts, and tool developers – in the creation of SENSEI models, which corresponds to the specification phase. SENSEI structures the overall process with the aim of cleanly separating specification and implementation. To this end, it introduces the service catalog and component registry, and corresponding tasks of service definition and component registration, which would otherwise be performed only implicitly, and jumbled in with implementation steps. This separation facilitates the automation of the implementation phase.

At the interface of specification and implementation are SENSEI models. They are the output of the SENSEI editor, and form the input of **SCAffolder**, an implementation of a SENSEI *toolchain generator* (see Section 9.1), named both for the target platform *Service Component Architecture (SCA)* [2015], as well as for the fact that it *scaffolds* the structures necessary to support the overall toolchain.

Figure 14.1 depicts the use cases covered by SCAffolder (recall Figure 9.5 on page 157, and Figure 13.1 on page 210). The first step of the implementation phase is *service-component matching*, i.e. finding appropriate components for all the service instances in a given orchestration. Since domain experts specify *required capabilities* for the service instances in their orchestrations, and tool developers define *provided capabilities* in the component registry that is part of SENSEI models, all information is already present in machine-readable form for this step to be fully automated by the *composition finder* of SCAffolder. The result is an orchestration augmented with adapter and transformer service instances, and a mapping of each contained service instance to an implementing component called a component *composition*.

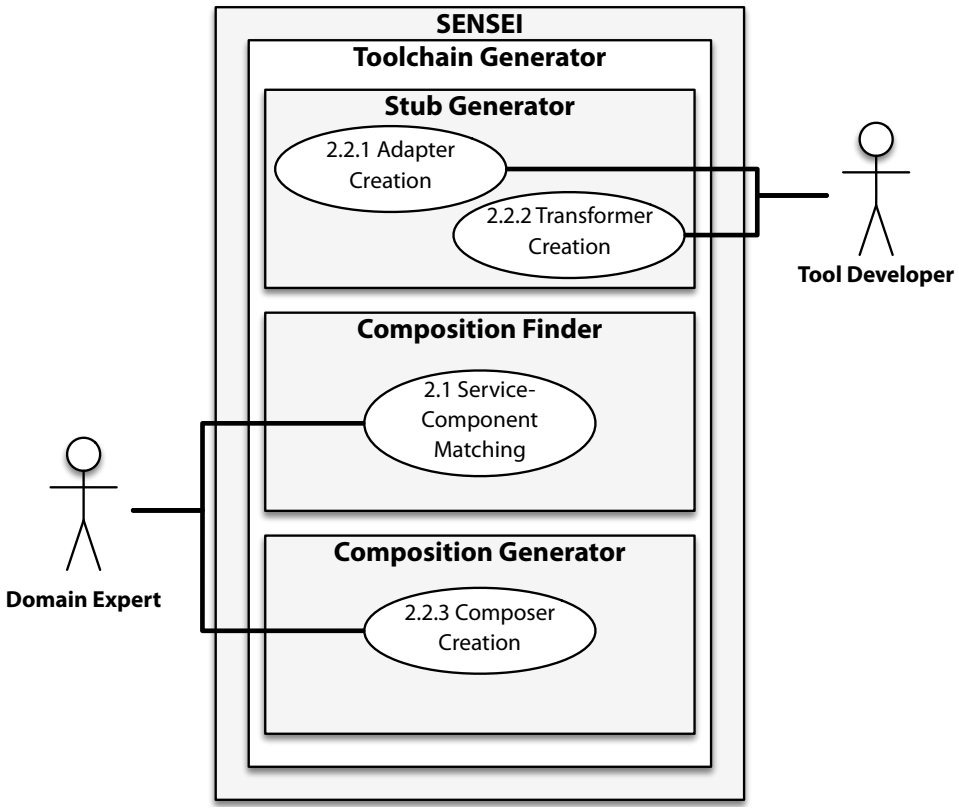



Figure 14.1: Use cases addressed by SCAffolder.

The next two steps of the implementation phase are *adapter creation* and *transformer creation*. While these are still manual tasks to be performed by tool developers, SENSEI requires both adapters and transformers to be encapsulated as components, facilitating reuse and encouraging their development up-front, decoupled from the actual toolchain-building. Thereby, they become part of the collection of components entered into the registry, available to fill in gaps between otherwise incompatible tools. The *stub generator* of SCAffolder supports tool developers in creating adapters and transformers by generating boilerplate code.

With these preparations, the *composer creation* step of integrating the individual components according to the given service orchestration can also be fully automated. The *composition generator* of SCAffolder takes SENSEI models, containing a service catalog, orchestrations, and a component registry, including adapter and transformer components, as well as concrete component compositions as input and outputs fully integrated, executable toolchains, by creating composers through code generation.

The remainder of this chapter is organized as follows: first, Section 14.1 provides a high-level overview by describing the processes to be supported by SENSEI toolchain generators on an implementation-agnostic level. Section 14.2 motivates design decisions and briefly introduces technologies used for SCAffolder. An overview of its implementation is given in Section 14.3. Section 14.4 shows how SCAffolder is used along an example. In Section 14.5, runtime interpretation of SENSEI models as an alternative to code generation is discussed, and a corresponding implementation created by Küpker [2015], SNOrcInS, is briefly described. A summary is given in Section 14.6.

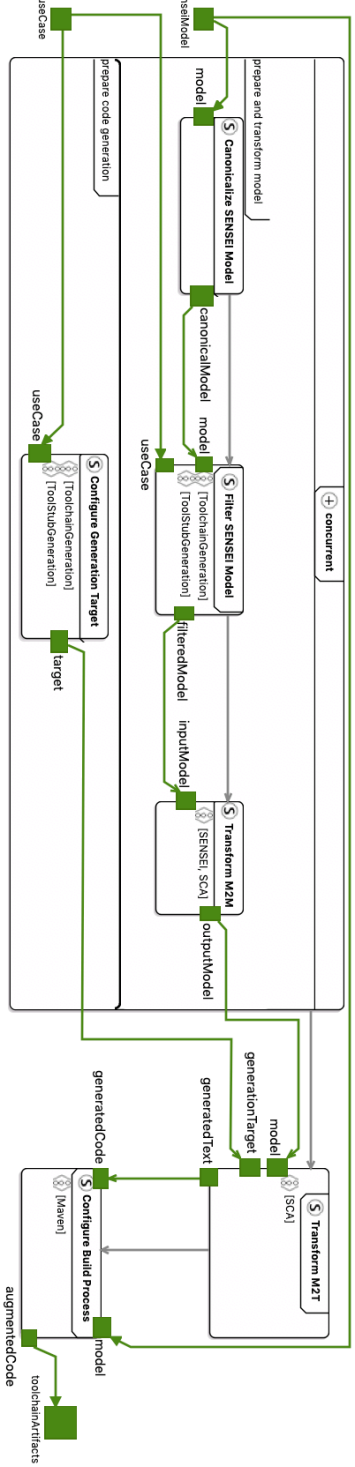
## 14.1 Specification

There are four use cases in Figure 14.1, owed to their origin in the toolchain-building process (Section 3.1), but there really only need to be two processes to cover them all: *adapter creation* and *transformer creation* both yield SENSEI components, which are meant to serve different purposes, but follow the exact same structural rules. Both use cases are therefore supported by the same process of *stub generation*. *Service-component matching* is a necessary prerequisite to *composer creation*, but the former is not usually performed without subsequently performing the latter, which is why the two are combined into a single process, which in the following will be referred to as *toolchain generation*.

Figure 14.2 depicts the processes that SCAffolder needs to implement to support the use cases of both domain experts and tool developers – modeled in terms of SENSEI orchestrations. Figure 14.2(a) is a high-level view of the *toolchain generation* process. It shows an orchestration of two service instances: *Find Composition* and *Generate Toolchain Artifacts*, which correspond to *service-component matching* and *composer creation*, respectively. *Find Composition* consumes SENSEI models, and outputs them again, supplemented with mappings between orchestrated service instances and components contained in the model's registry. *Generate Toolchain Artifacts* also takes a SENSEI model as input – in this orchestration, the one that has been augmented with a service-component mapping. The output is a set of artifacts, which together form an executable application that *is* the toolchain.This service has another input for the desired use case, which here will always be fed the same value of "toolchain generation"; the need for this input parameter will become apparent in a moment.

Figure 14.2(b) is an orchestration to *Generate Toolchain Artifacts*, which both corresponds to *stub generation*, as well as to *composer creation*, as it is a drill-down into the second service instantiated in that orchestration. This means that, on a fundamental level, the only difference between the two applications of SCAffolder – generating stubs and generating composers (i.e. toolchains) – is the preprocessing of SENSEI models as performed by the *Find Composition* service. Otherwise, the basic process is actually the same, and is configured at runtime for either of the two use cases, which is why the corresponding input parameter appears in the Figure 14.2(a).


(a) Service orchestration for toolchain generation.




(b) Service orchestration for SCA artifact generation (used for both stubs and toolchains).

Figure 14.2: SENSEI service orchestrations specifying SCAffolder.

The *Generate Toolchain Artifacts* orchestration works as follows: first, the SENSEI model input is processed by an instance of the *Canonicalize SENSEI Model* service. It ensures certain wellformedness properties and augments the model, if necessary – this does not change its semantics, but simplifies subsequent model transformations by making otherwise necessary case distinctions redundant. Next, *Filter SENSEI Model* strips the SENSEI model of anything not necessary for the given use case, which is passed in as a parameter and is either "toolchain generation" or "stub generation". For example, orchestrations are completely ignored, and thus filtered out, for stub generation.

After these prerequisite steps, the actual model transformation begins with *Transform M2M*. As opposed to the previous services, this one is highly generic. It is refined by requiring the capability of transforming SENSEI models to SCA models, represented by a corresponding capability tuple.

Concurrent to this series of service invocations is *Configure Generation Target*. Once again, the use case parameter is used to activate one of two capabilities for either *ToolchainGeneration* or *ToolStubGeneration*. The service produces a list of artifacts (files) to be generated into the final output.

Together with the SCA model produced by the *Transform M2M* instance, this artifact list is consumed by *Transform M2T*, another generic service, to generate code and configuration files needed to form either SCA applications (toolchains) or components. Finally, *Configure Build Process* does some post-processing to the generated code, particularly the build script, to add the components required by the generated toolchain (read from the composition contained within the SENSEI model) as dependencies for automatic resolution by the build tool. For now, SCAffolder supports integration with *Apache Maven* [2020] for this purpose. Because SCAffolder can be integrated into existing projects, the build script may have to be amended rather then generated from scratch, which is why this has been modeled as a service separate from the model-to-text transformation.

## 14.2 Technology Evaluation

Summing up the previous section, SCAffolder's processes for stub and toolchain generation rely on generic services representing model-to-model transformations followed by model-to-text transformations for the heavy lifting, with some simple, custom pre- and post-processing steps to complement them. The generic services can therefore be implemented with existing tools of model-driven development.

This section briefly introduces the concrete technologies that have been chosen to realize SCAffolder in, namely the TGraph technical space and its GReTL transformation language for model-to-model transformation, as well as Velocity for the model-to-text step. Another design decision that has been made concerns the use of the *Service Component Architecture* (SCA) standard as target platform. A high-level overview of SCA concepts relevant to this thesis is given in Section 14.2.3.

### 14.2.1 Model-to-Model Transformations with TGraphs

The TGraph technical space has been briefly introduced in Section 8.7: TGraphs are based on solid graph theory foundations, and represent a specific class of graphs that are typed, directed, attributed, and ordered [Ebert and Franzke, 1995]. TGraph schemas can be modeled using UML class diagrams in the *grUML* dialect [Ebert, Riediger, and Winter, 2008], enabling their use in model-driven development. In this manner, a Java API was generated from the SENSEI metamodel, and is provided as class library called *SDMTGraph*[1].

In contrast to other technical spaces, such as the widely-used EMF, edges in TGraphs are first-class elements, as opposed to reference links between vertices. This makes navigating and querying graphs easier, and enables, for example, following edges against their direction, and querying for edges, directly. Furthermore, metamodeling in EMF's EMOF can be restrictive, as it enforces a tree-like structure on models. Relations crossing this tree structure are represented as reference links, rather than first-class edges, which can make navigating and querying ECore models cumbersome at times. Therefore, the TGraph technical space was chosen over EMF, which does not have such restrictions.

TGraphs can be modeled, represented, and manipulated using the Java library *JGraLab*. Similar to EMF, JGraLab can take a metamodel and generate corresponding Java interfaces and classes from it to form an API. There is also a bridge implemented in JGraLab, from TGraphs into the EMF technical space[2], which enables the use of models created with the EMF-based SENSEI editor.

JGralab also implements GReQL (*Graph Repository Query Language*, Kamp [1998] and Marchewka [2006]) and GReTL (*Graph Repository Transformation Language*, Ebert and Horn [2012]), the latter relying heavily on the former to express its mappings. A GReTL program consists of transformation rules to create vertices, edges, and attributes in the target model. It does so by relying on GReQL queries into the source model: for each source model element returned by a transformation rule's query, a corresponding target model element is created.

Rather than going any deeper on how these model transformations work at this point, concrete examples will be given and thoroughly explained in Section 14.3. For a comprehensive overview of GReTL, including the basics of TGraphs and GReQL, the reader is referred to Ebert and Horn [2012].

### 14.2.2 Model-to-Text Transformations with Velocity

The TGraph technical space does not prescribe any particular tool or language for model-to-text transformations. Therefore, the template engine *Apache Velocity* [2020]

[1] *SDM* stands for *service description model*, meaning the SENSEI metamodel.

[2] The aforementioned tree structure has to be forced onto TGraph metamodels to have this work, though.

was chosen due to its maturity, its documentation, requiring very few dependencies, its low complexity, and its ease of use.

Template engines are a convenient approach at code generation: Instead of writing all the necessary code to an initially empty file, templates are used, which contain the static, "boilerplate" code, as well as placeholders and embedded directives that get processed during generation, and replaced with code fragments derived dynamically from an input model.

The *Velocity Template Language* (VTL) uses directives beginning with a # sign, e.g. to loop over elements from the source model, or to invoke macros. During code generation, the directives are evaluated and replaced with their result. To reference the source model from which code is to be generated, Velocity allows to access variables which have been stored in its context, using their names prefixed with a *$* sign. The values of these variables can be arbitrary Java objects, and Velocity allows invoking their methods, too, using regular dot notation.

Again, instead of giving more in-depth details here, concrete examples are deferred to Section 14.3. The template syntax is actually pretty straight-forward, but a comprehensive reference is available as part of the official documentation.

### 14.2.3 Target Platform Service Component Architecture

In support of the requirements for *Tool Interoperability*, *Uniform Interfaces*, and *Reusability*, component-based technology (Chapter 6) is employed on the integration and implementation-side of the SENSEI approach. While SENSEI as a conceptual framework does not dictate any particular target platform – a framework and runtime environment with an underlying component model – when it comes to implementing a concrete toolchain generator, a decision has to be made as to which one(s) to support.

For SCAffolder, a comprehensive comparison study was performed by Ringe [2013], analyzing the requirements to derive assessment criteria, and then evaluating different component models, including SCA [Chapman et al., 2011], *OSGi* [2020], EJB [*Enterprise JavaBeans* 2019], *CORBA Component Model (CCM) 4.0* [2006], and *COM: Component Object Model Technologies* [2018], though the latter was dismissed early on, as platform independence was a knock-out criterion, and COM is Microsoft Windows-specific. For all others, at least two conforming frameworks were considered. The SCA framework *Apache Tuscany* [2016] scored the best ratings overall, set apart by its ability to incorporate existing software built with diverse technologies and for different platforms. Furthermore, the SCA standard incorporates service-oriented principles most directly. The option of mapping to an existing orchestration language such as BPEL, although formally supported by SCA, has not been pursued, though, as Ringe [2013, pp. 73ff], as well as further case studies [Crone, 2013; Meier, 2012; Tihonov, 2013], revealed that the available frameworks and associated tooling in this regard was too immature.

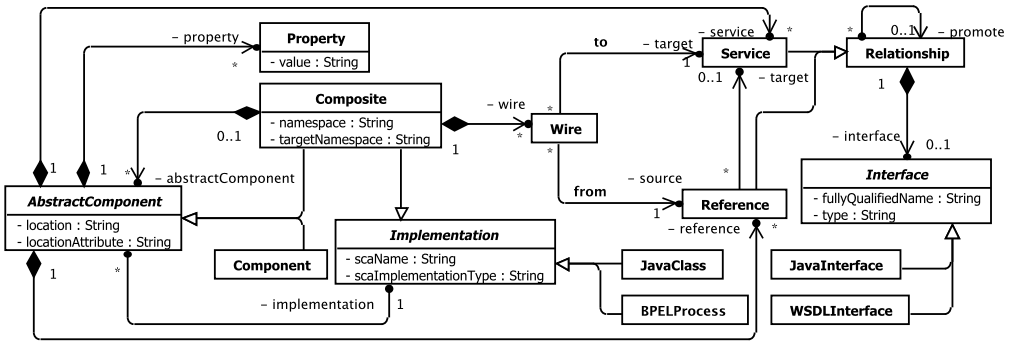



Figure 14.3: The central concepts of SCA (excerpt from SCAffolder's target metamodel).

The Service Component Architecture (SCA) is a component-based and service-oriented technology and standardization effort, born out of several vendors agreeing to combine their efforts towards a common conceptual solution. These vendors included IBM, BEA, Sun Microsystems (now Oracle), and SAP [Marino and Rowley, 2009, p. 7]. The joint venture was created in the fall of 2005, and named *Open SOA Collaboration* (OSOA). After developing an initial version of SCA, work was transferred to OASIS in 2007, which created a dedicated member section and six technical committees focused on different aspects of SCA, for continued development and formal standardization.

**The Fate of SCA**

In April 2016, activities of the SCA technical committees had ceased and they were officially closed, without any final standard documents having been produced (all the latest specification drafts remaining publicly available, though). Preceding this was a vote of the SCA Bindings Technical Committee in June 2013 – about half a year after Ringe completed his comparative study – to approve the latest version of its specification as an official standard [*Mailing List Archives of the SCA-Bindings Technical Committee* 2013]. The approval failed, to the surprise and upset of several committee members, due to Oracle's representatives blocking it. Members of Metaform Systems, Siemens, and Tibco criticized this quite strongly on the committee's public mailing list. Nevertheless, this move seems to have been the beginning of the end of SCA as an official OASIS standard.

As a result, the Apache Tuscany project has been retired [Yandell, 2016] due to lack of development and maintenance activity. Likewise, IBM announced the deprecation of SCA support in products such as *Rational Software Architect Designer*

**The Fate of SCA (cont.)**

[IBM, 2016]. And sometime in 2018, the webpage of *Fabric3* [2016] disappeared, as well. *SwitchYard* [2020] had been under development since at least 2011, but Version 1.0 was only released in August 2013 (therefore, it was not included in the comparison by Ringe [2013]). It is clearly based on SCA, though this fact is not used to advertise the product. It remains available, but no new releases have appeared since August, 2016.

The conceptual framework that is SENSEI is not affected by the failure of the standardization efforts; SCAffolder, however, does generate SCA-conforming tool stubs and toolchains. While they were intended to be run on the now discontinued Apache Tuscany implementation, SCAffolder-generated applications have also successfully been deployed to other runtimes with no, or only minor, manual adjustments. Tuscany remains available, as well, but without support or maintenance from the original contributors at the Apache Software Foundation. As an alternative to SCAffolder, there is SNOrcInS, which does not rely on SCA.

Figure 14.3 depicts the central concepts of the *assembly model*, which is essentially the component model (Definition 6.2, page 104) of SCA. In fact, this is a partial rendering of the target metamodel created for SCAffolder (the complete metamodel is explained later, and is depicted in Figure 14.6). SCA defines the XML-based *Service Component Definition Language* (SCDL) to describe assembly models, which can also be represented graphically.

SCA *components* are either atomic, or they have other components nested within them; the latter kind are called *composites*. The commonalities of both are represented in Figure 14.3 by the metaclass *AbstractComponent* (which is not a concept of the SCA assembly model standard). Components and composites refer to their *implementation*, which can be of different types, e.g. *Java classes* or *BPEL processes*.

Composites and components declare *properties*, *services* and *references*. Composites may also declare *wires*, which bind references to services. Both services and references are specified by their associated *interfaces*. Again, SCA allows for different ways to define them, but the most common ones are *Java interfaces* or *WSDL interfaces*.

An example is shown in Figure 14.4, using SCA's graphical rendering of SCDL. This model describes a simple toolchain for metric calculation in the Q-MIG project (see Chapter 2) as SCAffolder is expected to generate (see Chapter 15 for the full application of SENSEI to Q-MIG). There are four composites, drawn as rounded rectangles, each containing one component. The *Composer* component of *CalculateAllMetrics* exposes a single SCA service, indicated by the chevron-like shape on the left, which is used to invoke the toolchain, and declares five references (chevrons on the right). The composer uses these references to invoke the required functionality on the SCA composites and components that make up the toolchain. To this end, the references are connected by

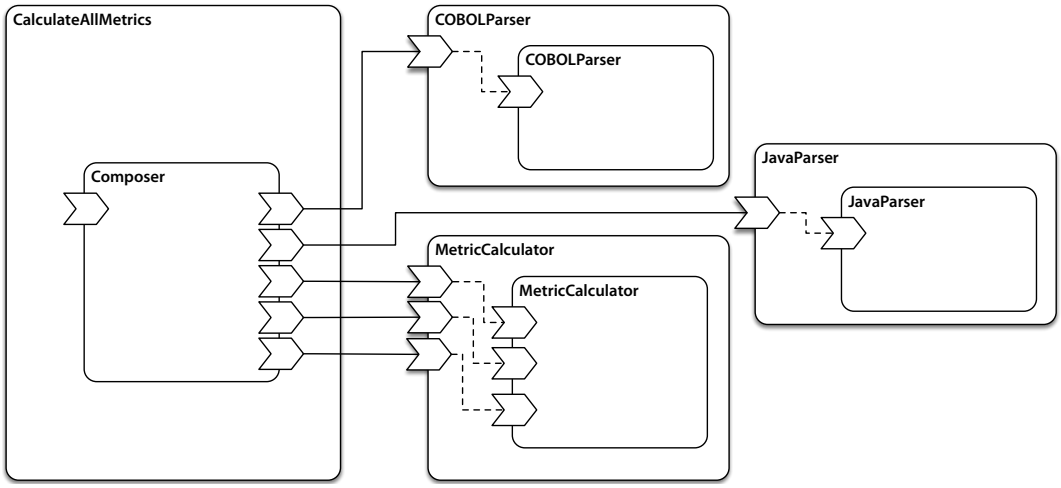



Figure 14.4: An SCA assembly model example using the graphical SCDL notation.

*wires* (solid lines) to compatible SCA services exposed by the other composites. The dashed lines represent *promotions* of SCA services exposed by an SCA component to the surrounding composite.

Individual SCDL files always represent a composite. Therefore, all components are wrapped in one, to each be described in separate files and be bundled up with component implementations into SCA *contributions* (usually just zip files). The wiring is defined by *CalculateAllMetrics*, referencing wire targets by name. They get resolved at runtime, if all necessary contributions are present. The graphical notation has no syntax to depict links between SCA components and their implementations, and SCA services and their interfaces. With Java, SCDL files simply contain corresponding entries referring to Java classes and interfaces, respectively, by their fully-qualified names.

There is, of course, a lot more to SCA, but for now this brief introduction should suffice as a basis for the remainder of this chapter. When additional aspects come up at later points, they will be explained there. A deeper dive into SCA is offered by Marino and Rowley [2009] and Laws et al. [2011], the former presenting a slightly more implementation-agnostic picture, while the latter focuses on Apache Tuscany.

## 14.3 SCAffolder Implementation

The basic breakdown of SCAffolder into components is depicted in Figure 14.5, using the SCA notation as just introduced, with additional labels for SCA services, and a box at the bottom of components to indicate the implementation technology (Java). The overall composition corresponds to the specification given in Section 14.1 as fol-

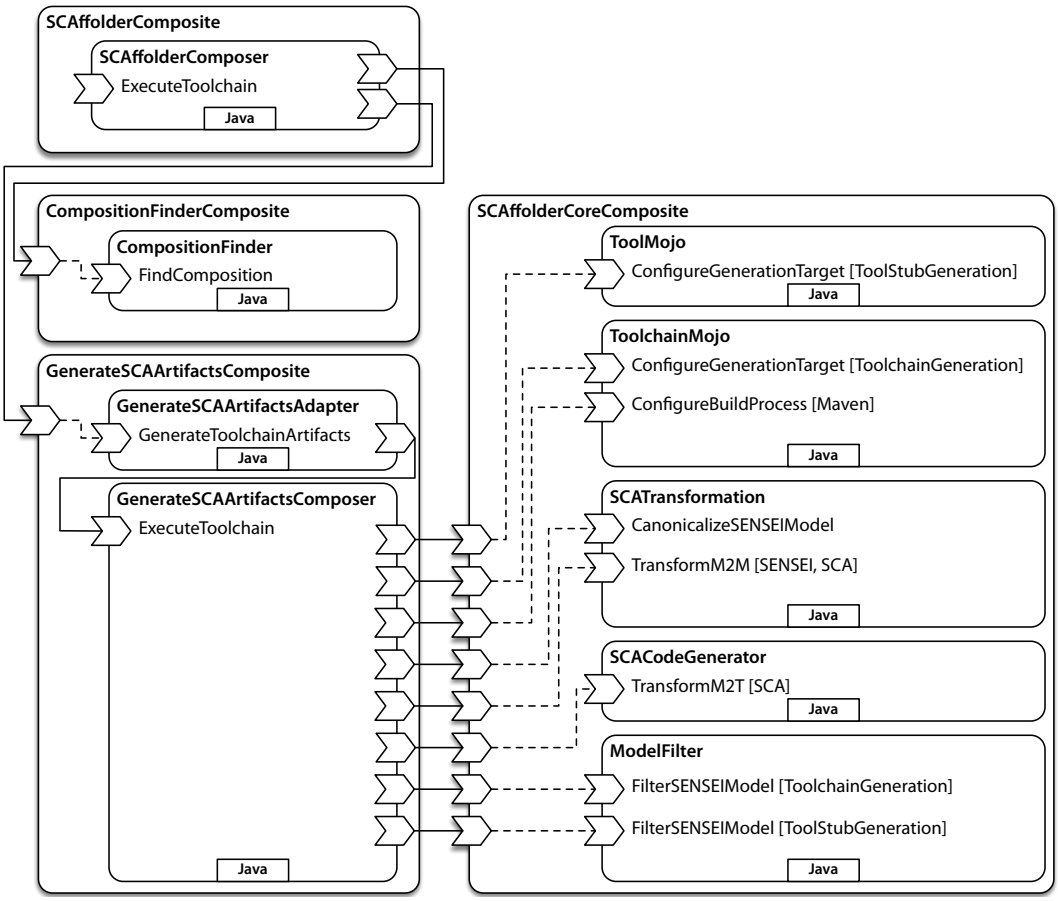



Figure 14.5: Basic components of SCAffolder.

lows: *SCAffolderComposer* implements the toolchain generation orchestration (Figure 14.2(a)), by simply invoking first the *FindComposition* SCA service on the *CompositionFinderComposite*, followed by the *GenerateToolchainArtifacts* SCA service of *GenerateSCAArtifactsComposite*. The latter is itself an implementation of the orchestration to generate toolchain artifacts (Figure 14.2(b)), with an adapter to fit the generic *ExecuteToolchain* SCA service interface to the expected one. *GenerateSCAArtifactsComposer* contains the actual coordination logic conforming to the orchestration.

The actual service logic is implemented by components contained in *SCAffolderCoreComposite*: depicted at the top are *ToolMojo* and *ToolchainMojo*, which implement the configuration services that take care of some pre- and post-processing. Notice that both components implement *ConfigureGenerationTarget*, but each providing a different capability. The "mojo" qualifier is due to SCAffolder's integration with the Maven

build system and refers to "*Maven plain Old Java Objects*". By tapping into the Maven API, SCAffolder can be plugged into the build process of new or existing projects to generate toolchain code (more on that in Section 14.4).

The most complex work is done by *SCATransformation*, which implements *TransformM2M*, the model-to-model transformation from SENSEI to SCA. The comparatively trivial *CanonicalizeSENSEIModel* is also implemented by this component. The model-to-text stage is performed by *SCACodeGenerator*, which implements *TransformM2T*. *ModelFilter* implements the remaining *FilterSENSEIModel* service, for both of the capabilities required by the orchestration.

The implementations of the post- and pre-processing steps are quite straight-forward: For *ConfigureBuildProcess*, *ToolchainMojo* taps into the Maven Plugin API to add dependencies that contain the components used by the generated composer, so that the build process can fully assemble the toolchain.

For *CanonicalizeSENSEIModel*, *SCATransformation* augments the SENSEI model using the JGraLab API. It gives control flow elements, e.g. for-each-blocks, conformance relations to artificially introduced catalog services. The reason is found in a fundamental limitation of metamodeling infrastructures like MOF, as described in Section 8.3.

The SENSEI editor relies on control flow elements being instances of *AbstractServiceInstance* defined in the SENSEI metamodel – this is the *linguistic* modeling dimension. *Ontologically*, service instances must also conform to catalog services. However, control flow elements are an integral part of SENSEI itself, and thus are not based on user-defined services. The SENSEI editor does not create services for them, either, as they would appear in the catalog and confuse users. Instead, they are added here so subsequent model-to-model transformations can rely on their presence, which simplifies many transformation rules.

Both *ToolMojo* and *ToolchainMojo* simply return a list of files to be generated for *ConfigureGenerationTarget* – which files those are will be explained together with the description of the code generation implementation.

This leaves three large parts of the implementation to be explored in more detail in the following: Section 14.3.1 gives an overview of the *CompositionFinder*, Section 14.3.2 provides insights into the implementation of *SCATransformation*, including a view of the full platform-specific target metamodel that has been created for this purpose, and Section 14.3.3 describes the final code generation step.

### 14.3.1 Composition Finder Component

Finding valid mappings between orchestrated service instances and implementing components is a non-trivial task. Solving this problem "on the side" as part of the model-to-model transformation is hard, as SENSEI's *capability model*, data compatibility requirements, and the ability to have transformers and adapters automatically dropped into

orchestrations as needed, can lead to very complex sets of constraints. Transformation languages are simply not an appropriate choice for this kind of job.

Composition finding is, in essence, a *constraint satisfaction problem*, as defined, for example, by Tsang [1993, p. 1, pp. 5ff]. Constraint satisfaction is a broad field that offers diverse solution techniques and algorithms, and has developed a range of different technologies to apply to this class of problems. For her bachelor's thesis, Meier [2014a] surveyed different approaches and frameworks, and chose *Prolog* as an appropriate language for encoding SENSEI's composition finding problem. She subsequently designed and implemented a *CompositionFinder*, which solves this problem.

Meier [2014a] evaluated five Prolog implementations, and two *Object Constraint Language* solvers, and chose SWI Prolog [Wielemaker, 2020]. The CompositionFinder wraps the Prolog-based finding algorithm in a Java application. It generates Prolog fact bases from SENSEI models before invoking the Prolog program.

The composition finding algorithm is quite complex; it is described in great detail by Meier [2014a, pp. 63-85]. Conceptually, it follows the phases described in Section 12.2. Among the features this implementation offers is the ability to insert transformers into data flows that connect incompatible ports. This is done both on the orchestration level, by inserting appropriate transformer service instances, and on the composer level, by mapping these service instances to implementing transformer components.

The CompositionFinder even considers the case in which incompatible ports of orchestrated services are connected by data flows. In terms of the formal SENSEI model developed in Chapter 12, they are not *assignment compatible* (Section 12.3). *Strong* assignment compatibility (which takes restrictions into account, as well) has since been considered a prerequisite for valid orchestrations. This is because transformers can only be inserted automatically to remedy syntactical differences in the data being exchanged. Neither the semantics of the services, nor the intention of an orchestration's modeler are at the disposal of automated composition finding. Allowing orchestrations with incompatible data flow connections would lead the CompositionFinder to insert some transformer, or even a chain of transformers, that ensures syntactic compatibility by some arbitrary means. A trivial, non-sensical transformer would simply discard its input, and output some constant data that conforms to the expected data format.

The CompositionFinder itself is not faulty, it simply cannot be expected to produce correct results if its *input* is faulty, or semantically incomplete. The work by Meier also drove the formalization of the SENSEI metamodel, and the introduction of concepts like assignment compatibility, activation, and orchestration trails (Section 12.3), to define service-component matching semantics more rigorously. When passing in SENSEI models with orchestrations that are valid with respect to these definitions, the CompositionFinder will not have to try to remedy incompatible data flow connections, and will return correct results.

### 14.3.2 SCA Transformation Component

The core of SCAffolder is arguably its model-to-model transformation, implemented entirely in GReTL, with a total of well over 250 transformation rules. Those rules fill a model conforming to the target metamodel, which has been created for SCAffolder. A part of it – the SCA assembly model part – has already been introduced in Section 14.2.3.

SCA supports different (programming) languages and technologies for implementing its components, and specifying interfaces. For SCAffolder, Java was selected to fill both roles. To support the generation of *arbitrary* Java code, however, the target metamodel would have had to be extended to represent the whole language specification. Constructing such a complex metamodel was deemed infeasible in the course of this thesis, in particular because it would have done little to further its research objectives. Representing Java down to its most basic language features would also have significantly increased the number of required transformation rules. Therefore, a more pragmatic approach was chosen:

Figure 14.6 depicts the complete target metamodel[3]. The upper part corresponds to the SCA assembly model. Below this are concepts that model Java classes and interfaces, their methods, parameters, and types, as well as annotations. Finer granularity levels, i.e. statements and expressions, are not modeled in detail, though. Instead, this level is abstracted by several compound statement meta-classes (e.g. *InvokeAndReturn*), that represent small sets of code snippets needed to generate composer logic. The rest is left to the model-to-text transformation stage.

The following gives a basic, high-level overview over the mappings that SCAffolder defines between source and target model. The metamodel in Figure 14.6 serves as reference for this, but will not be explained in detail, here. For an in-depth description of how conforming target models are created during transformation, see Appendix B.

- SENSEI orchestrations become *composers*: they consist of an SCA component, wrapped in a composite, defining an SCA service with an associated Java interface *ExecuteToolchain*. The SCA component is realized by a Java class, that implements the SCA service interface, and is additionally labelled with SCA's *@Service* annotation. The interface defines a single method, *execute*, with a signature based on the input and output ports of the orchestration[4]. The method's body contains statements that implement the orchestration's data and control flow.
- As an alternative to the generic *ExecuteToolchain* interface, interfaces are also created for each *orchestration trail* (see Section 12.3). These are also implemented

[3] Some elements of SCA that had been shown previously in Figure 14.3, like the *Property* concept, are omitted here, since SCAffolder does not use these.

[4] Since there can only be one return value in Java, SCAffolder will generate a compound type as a wrapper, if there are multiple output ports defined.

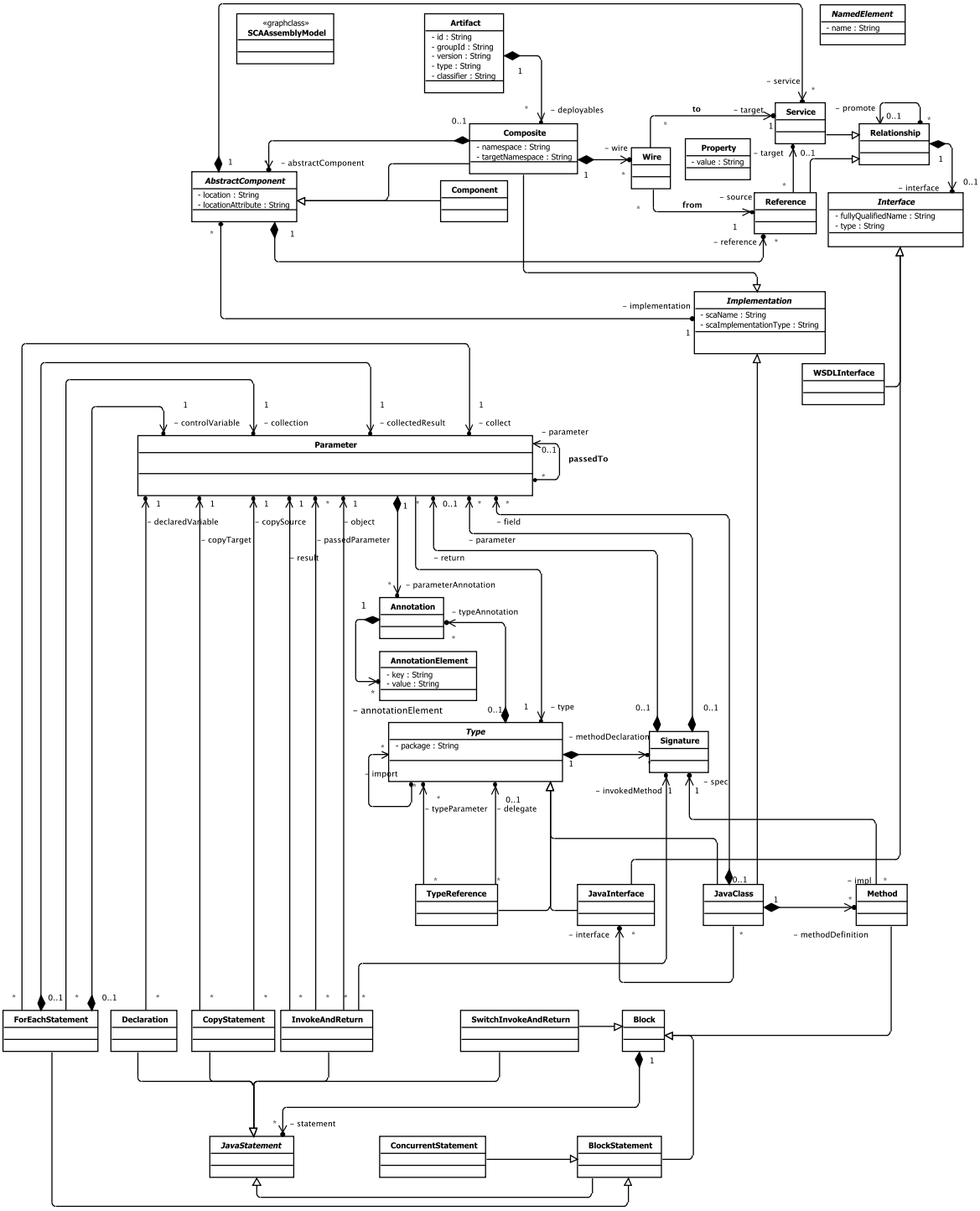



Figure 14.6: The complete target metamodel (PSM) of SCAffolder.

by the *Composer* class, and allow to invoke a particular trail, specifically, whereas the *execute* method will try to dynamically determine which capability tuples to activate based on the runtime data.

- SENSEI service instances in orchestrations become SCA references. More specifically, for each capability tuple of an orchestrated service instance a SCA reference is created. For the *Composer* Java class, fields corresponding to these references are generated and annotated with *@Reference*.
- SENSEI components become SCA components, wrapped in composites, and associated with a Java class for implementation.
- Service instances implemented by components become SCA services, exposed by corresponding SCA components, and promoted to surrounding composites.
- Capability tuples provided by an implemented service instance become Java interfaces with a single method in it.
- Input ports defined on implemented services instances become parameters of the method defined in the corresponding Java interface.
- Output ports defined on implemented services instances become the return parameter of the method defined in the corresponding Java interface. Since there can only be one return value in Java, SCAffolder will generate a compound type as a wrapper, if there are multiple output ports defined.
- Restrictions on an implemented service become additional methods in the corresponding Java interface. These methods take a single argument, and return a boolean – tool developers implement these methods to indicate whether their tool can handle the provided data, and composers call them to decide which component to use for a particular service invocation.
- SENSEI compositions, i.e. the mapping between orchestrated service instances with required capability tuples and service instances implemented by components with provided capability tuples, become SCA wires. They connect each SCA reference to an appropriate SCA service provided by the SCA component which corresponds to the SENSEI component named by the composition.

This is not a complete list, as a lot of miscellaneous elements in the target model need to be created. For example, the association of Java class fields, method parameters, and local variables with type information, and the generation of these types from SENSEI's data structures and data types, has been omitted. Similarly, the mapping of orchestrations to Java statements is not covered in detail, but will become clearer with examples given in Section 14.3.3. A more comprehensive documentation of the model-to-model mappings implemented by SCAffolder is provided in Appendix B.

Figure 14.7 provides an impression of how the actual implementation code looks like. It shows the GReTL transformation rules that create SCA composite instances for each SENSEI component in the source model – this is why all semantic expressions

```
1  new CreateVertices(context, Composite.VC,
2          "from c : V{Component} "
3          + "reportSet c,'Composite' "
4          + "end").execute();
5  new SetAttributes(context, attr("Composite.name"),
6          "from c : V{Component} "
7          + "reportMap tup(c,'Composite') -> c.name "
8          + "end").execute();
9  new SetAttributes(context, attr("Composite.targetNamespace"),
10         "from c : V{Component} "
11         + "reportMap tup(c,'Composite') ->
   ↪  'http://se.uni-oldenburg.de/ses/' "
12         + "end").execute();
13 new SetAttributes(context, attr("Composite.scaImplementationType"),
14         "from c : V{Component} "
15         + "reportMap tup(c,'Composite') -> 'composite' "
16         + "end").execute();
17 new SetAttributes(context, attr("Composite.scaName"),
18         "from c : V{Component} "
19         + "reportMap tup(c,'Composite') -> c.name "
20         + "end").execute();
21 new SetAttributes(context, attr("Composite.location"),
22         "from c : V{Component} "
23         + "reportMap tup(c,'Composite') -> c.name "
24         + "end").execute();
25 new SetAttributes(context, attr("Composite.locationAttribute"),
26         "from c : V{Component} "
27         + "reportMap tup(c,'Composite') -> 'name' "
28         + "end").execute();
```

Figure 14.7: Transformation operations creating composite instances in the target model, and setting attribute values, for each source model *Component* instance.

(GReQL queries) range over the set of all SENSEI components (`V{Component}`). The first rule creates vertices, while all the others then set their attribute values. Their semantic expressions return a mapping from archetypes corresponding to (previously created) target model elements, to values that should be assigned to one of its attributes. These values can be derived from the source model, or be constants. E.g. in Line 8, each *Composite* instance's name is assigned the name of the corresponding *Component* instance in the source model. Line 13 is an example of a constant value; it gets assigned to the *targetNamespace* attributes of all *Composite* instances.

### 14.3.3 SCA Code Generator Component

When the model-to-model transformation is done, the Velocity-based SCA code generator component takes the produced platform-specific model and uses it to create the

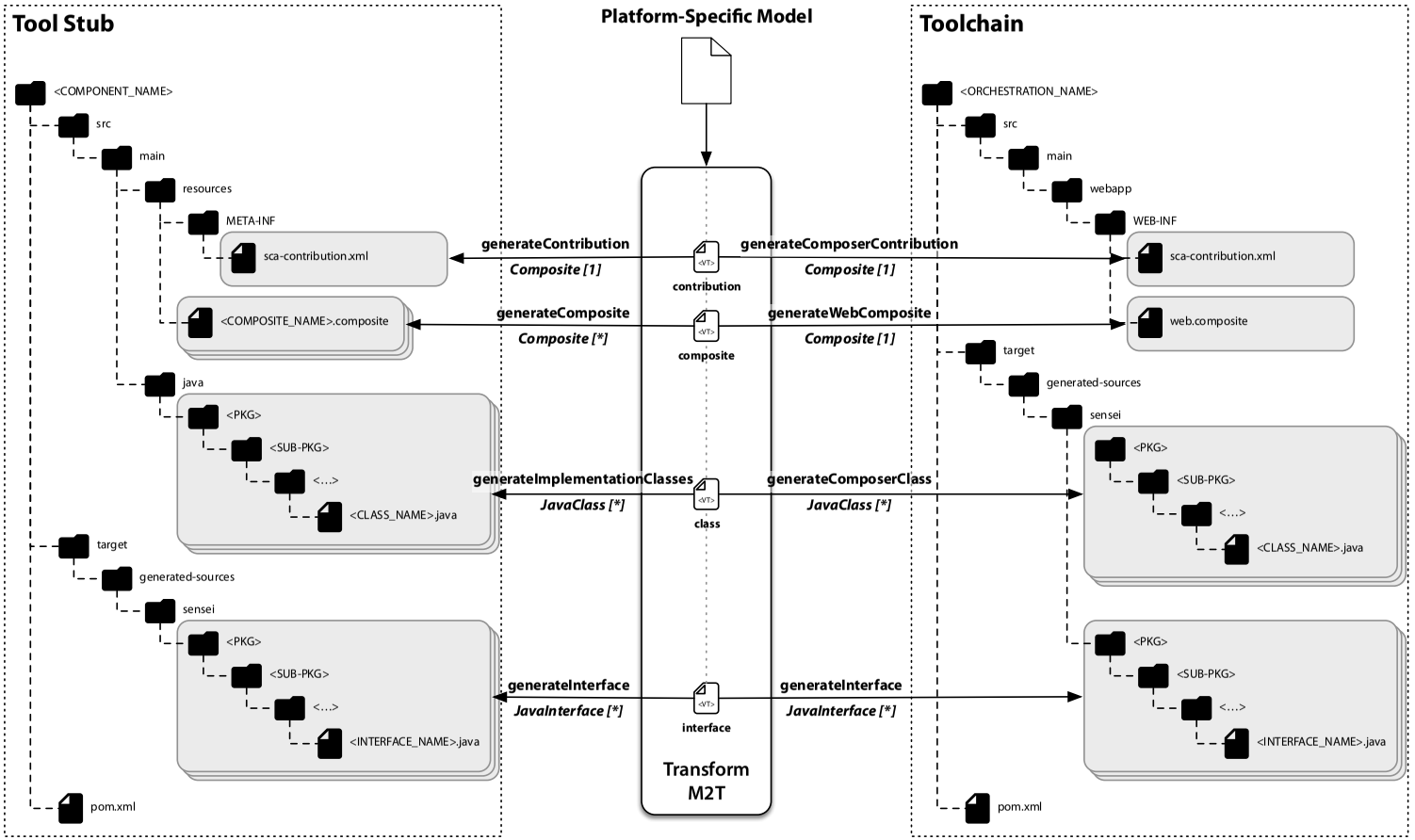



Figure 14.8: Overview of the files and directory structure generated by SCAffolder for tool stubs (left) and toolchains (right).

actual configuration and code files. It relies on JGraLab to work with the model that was output by the SCA Transformation component. It exposes the corresponding API to the Velocity templates, so the model can be accessed by template directives.

The code generator takes a second input provided by the *Configure Generation Target* service instance, or more precisely, one of its two implementing SCA components, *ToolMojo* or *ToolchainMojo*. Both produce a list of files to be generated, so the code generator knows which Velocity templates to process, and where to save the results.

An overview of the generated files and directory structure is provided by Figure 14.8, showing tool stubs on the left, and toolchains on the right. The box in the center shows the four main Velocity template files used for code generation. The outgoing arrows are labeled with the *generate…* methods used, and the type of elements that provide the information to be merged with the template. An asterisk in square brackets indicates the generation of multiple files (one per model element), while a one indicates a single file being generated.

Directory structures and generated artifacts are similar for both tool stubs and toolchains. One difference is that generated Java classes go into the regular source file tree for tool stubs, while they go into the *target* folder for toolchains (which is deleted during Maven's *clean* phase). This is because stub classes are meant as a starting point for adapter implementation, to be finished manually, whereas toolchain classes contain the fully auto-generated composer logic. Not meant to be altered manually, it can be considered disposable, in the sense that the code can be regenerated at any time.

The *generate* methods being used configure the code generation process as follows:

**generateInterface** selects all instances of *JavaInterface* from the platform-specific model to generate a Java file for each of them, based on the *interface* template file. The target directory is "target/generated-sources/sensei", followed by a path derived from the element's *package* attribute, in accordance with Java conventions.

**generateComposite** selects all instances of *Composite* from the platform specific model to generate a SCDL file (".composite") for each of them in the "src/main/resources" directory, using the *composite* template file.

**generateImplementationClasses** selects all instances of *JavaClass*, except for those representing composers, since this is a method used during tool stub generation. The *class* template is used, the target is the Maven project's Java source directory (usually "src/main/java"), followed by a path derived from the Java package.

**generateContribution** selects a single *Composite* instance (the first one contained in the model) to generate a "sca-contribution.xml" file into the "src/main/resources/-META-INF" directory, based on the *contribution* template. This file declares the SCA contribution's deployable composites.

**generateComposerClass** selects only those instances of *JavaClass* that represent composers, to generate their implementation into Java files. The *class* template is

```
52      {
53  ## Check if there are any statements: if not, its a stub method.
54  ## Otherwise, its a composer method, for which a map for global variables
55  ## is created, and the method's parameters are added to it.
56  #if( $method.get_statement().iterator().hasNext() )##
57          final Map<String,Object> data = new ConcurrentHashMap<>();
58                  //Collections.synchronizedMap(new HashMap<>());
59  #foreach( $parameter in $method.get_spec().get_parameter())##
60          data.put("#varName($parameter)", $parameter.get_name());
61  #end##
62  #blockBody($method $method)##
63          return (#typeName($method.get_spec().get_return().get_type()))
                data.get("#varName($method.get_spec().get_return())");
64      }
65  #else##
66  #blockBody($method $method)##
67      }
68
69  #end##
```

Figure 14.9: Excerpt from the *class* template, showing the code responsible for generating Java code inside of composer method bodies.

used and the target directory is “target/generated-sources/sensei”, followed by a path derived from the Java package.

**generateWebComposite** selects a single *Composite* instance named “Composer” to generate a “web.composite” SCDL file into “src/main/webapp/WEB-INF”, using the *composite* template. The path and file name are SCA conventions for web application server deployment. Composers can, but do not have to be deployed to such runtime environments.

**generateComposerContribution** selects a single *Composite* instance with the name “Composer”, stored in the Velocity context as “composite”, to generate a “sca-contribution.xml” file into “src/main/webapp/WEB-INF”, based on the *contribution* template.

The four main templates are *contribution*, *composite*, *class*, and *interface* (Figure 14.8). Each is used to generate files of the eponymous type. Velocity offers means to modularize templates, including the ability to define callable *macros*, to incorporate Velocity code from other files, as well as to copy code snippets stored in separate files verbatim into the output document. Both means are utilized by the SCA code generator component to reuse common functionality, and include repeated, static code patterns.

The *contribution* template is very simple, with only four lines of code. The *composite* template has 51 lines of code, and is still rather straight-forward, and the same is true for the *interface* template (27 lines). By far the most complex template is thus *class*,

```
221 final Collection<java.lang.Object> coalesce_336 = new
    ↪  java.util.ArrayList<>();
222 for (Object current_metric__342
223         : (Collection<java.lang.Object>) data.get("metrics__327"))
224 {
225     data.put("metric__342", current_metric__342);
226     Map<Integer, Number> collect_336 = calculatemetricfilejavasloc.
227             calculatemetricSLOCJavaFile((String) data.get(
228                     "metric__342"), (byte[]) data.get("outAST__330"));
229     setResult(data, "collect_336", collect_336);
230     add(coalesce_336, (Object) data.get("collect_336"));
231 }
```

Figure 14.10: An excerpt from composer code generated by SCAffolder for the base metric calculation orchestration (shown in Figure 11.7 on page 189; the example was first introduced in Section 2.2). The creation of tracing (extra logging) statements has been turned off for better clarity.

both in terms of its size (81 lines of code, plus the external macro definitions and code snippets, each of which amount to a combined 236 and 30 lines of code, respectively), as well as regarding the intricacy of its template logic.

The template is too large to list it in full. Its most interesting part is arguably the one dealing with generating the statements for composer method bodies. This part is shown in Figure 14.9. It first checks whether there are any statements – the *$method* variable is defined outside of this code snippet, in a surrounding for block. Its values are taken from the *JavaClass* instance's *ContainsMethodDefinition* adjacencies contained in the platform-specific model. If the method body is not empty, a Java statement is written into the output file for each of its arguments, storing their values in the global hash table. Then, the *#blockBody* macro is invoked, which controls the generation of further, potentially nested blocks, and statements. Lastly, a return statement is generated. If there are no statements in the method, only the *#blockBody* macro is called.

An example of the code that can result from this template is provided in Figure 14.10. It is a short excerpt from the complete composer class generated by SCAffolder from a SENSEI model containing the orchestration used for the base metric calculation example. One thing that may stand out are the identifiers, which all have a number appended to them. These parameter names are created this way in the model-to-model transformation stage to avoid any naming collisions. The numbers appended to the parameter names are the IDs of the corresponding vertices, which are guaranteed to be unique throughout the graph, i.e. the SENSEI model used as input to SCAffolder.

Lines 226 through 229 are generated for the single service instance nested in the *map* operator in the orchestration, and the *InvokeAndReturnStatement* instance generated for it by the SCA transformation component in the target model. The macro outputs a method call statement on the field holding the SCA reference corresponding

```
1  <build>
2      <finalName>JavaParser</finalName>
3      <plugins>
4          <plugin>
5              <version>1.0-SNAPSHOT</version>
6              <groupId>de.unioldenburg.ses</groupId>
7              <artifactId>scaffolder-maven-plugin</artifactId>
8              <executions>
9                  <execution>
10                     <id>sensei</id>
11                     <configuration>
12                         <model>metrics.orchestration.sdm</model>
13                     </configuration>
14                     <goals>
15                         <goal>generate-toolstub</goal>
16                     </goals>
17                 </execution>
18             </executions>
19         </plugin>
20         <!-- ... -->
21     </plugins>
22     <!-- ... -->
23 </build>
24 <!-- ... -->
```

Figure 14.11: Excerpt from a Maven POM file, showing the configuration settings to generate tool stubs as part of the build process.

to the service that needs to be invoked. The result returned by the method is then stored in the global hash map.

## 14.4 Using SCAffolder

SCAffolder integrates with Maven by exposing its functionality as Maven plugin, which can easily be inserted into the build process of new or existing projects. This way, building the overall project will automatically invoke SCAffolder, which generates code that is subsequently compiled, packaged, and deployed.

Figure 14.11 shows a snippet from a Maven build configuration, called a *Project Object Model* (pom.xml) file. It embeds SCAffolder in the project's overall build process and is configured to generate a tool stub for a component implementation.

First, Maven's existing *finalName* tag (Line 2) is used to specify the component for which to generate stub files. Alternatively, a dedicated *component* tag in the configuration section can be provided. A component with the corresponding name has to exist in the SENSEI model's component registry. When configuring SCAffolder for toolchain generation, the value of the *finalName* tag has to refer to an orchestration in

```
1 mvn archetype:generate -DinteractiveMode=false \
2                        -DarchetypeGroupId=de.unioldenburg.ses.maven \
3
  ↪  -DarchetypeArtifactId=sensei-component-archetype \
4                        -DgroupId=de.unioldenburg.ses \
5                        -DartifactId=JavaParser \
6                        -DcomponentName=JavaParser \
7                        -Dsensei-model-file=metrics.orchestration.sdm
```

Figure 14.12: Creating a SENSEI component project with preconfigured SCAffolder support using Maven and the supplied archetype from the command line.

the SENSEI model for which to create a composer implementation. Again, a dedicated configuration tag *orchestration* can be used to specify this as well.

In the *plugins* section, SCAffolder is included, and configured within its *executions* section. The most important setting is the path to the SENSEI model file (Line 12). Another mandatory setting is the selection of a goal, either *generate-toolstub* or *generate-toolchain*, implemented by the Maven "Mojos" of SCAffolder, *ToolMojo* and *ToolchainMojo*. An optional configuration tag is *remotable*, which is set to *true* by default. If set to *false*, generated Java interface will not be annotated as remotable, meaning only invocations of the corresponding SCA service from within the same runtime environment will be allowed. For toolchain generation, the boolean *standalone* tag can be given. It is disabled by default; if set to *true*, the packaging type is changed from *war* (Java web archive) to standard *jar*, and SCA configuration files will be generated into different target directories. While the default behavior allows to deploy generated composer into application servers with SCA support, the standalone setting is more appropriate if the toolchain is run locally only, with an embedded SCA runtime (Section 15.7 gives an example of how to launch composers in such an environment).

Like all Maven plugins, SCAffolder's execution can be explicitly bound to a specific build lifecycle phase. By default, both its goals will execute during the *generate-sources* phase, a pre-compile phase, ensuring that the automatically created code and configuration files will be considered during compilation and packaging.

When starting out from scratch, a complete pom.xml file, and the project directory structure has to be created. To avoid having to do this manually, Maven offers a large collection of *archetypes*. They are used to intialize a project with a preconfigured pom.xml file, create the necessary directories, and potentially code and further configuration files, depending on the project (arche-)type.

Along with SCAffolder, two Maven archetypes have been defined for this thesis, to initialize either a component project with tool stub generation support, or a toolchain project with support for auto-generating a composer implementation. They can be used through Maven's standard archetype:generate goal from the command line – an example is shown in Figure 14.12. This will create a new directory *JavaParser*, and within

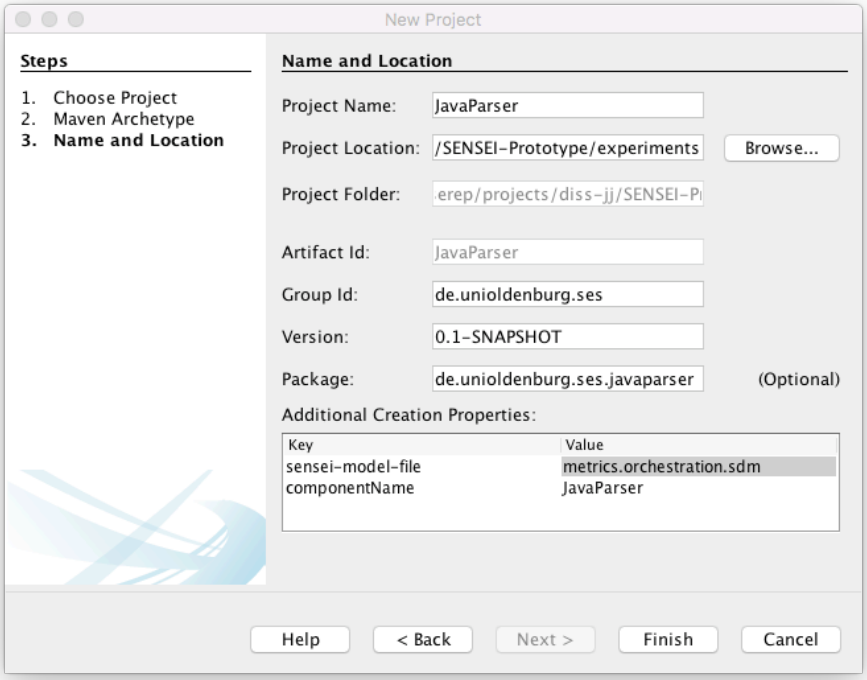
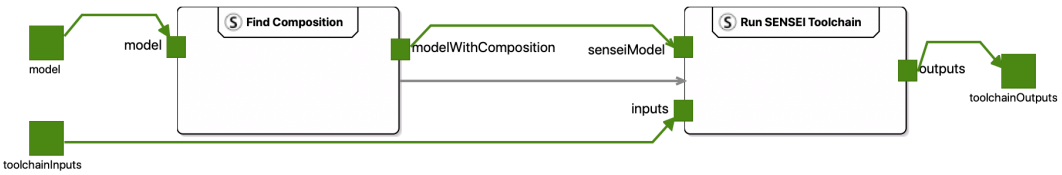


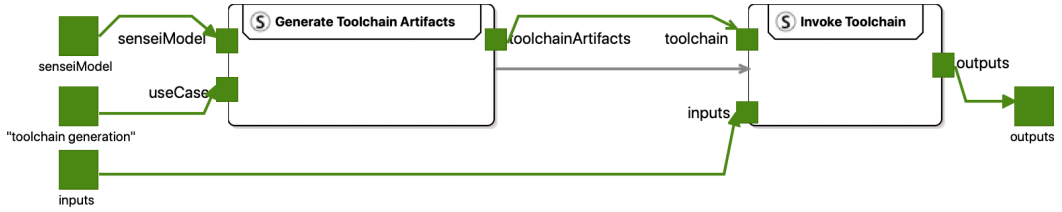

Figure 14.13: The *New Project* wizard of the NetBeans IDE being used to create a tool adapter project, based on a custom Maven archetype. Its Maven build file will be preconfigured to generate stubs of required files, such as Java interfaces and an SCA composite file, as part of the regular build process.

that, a fully preconfigured pom.xml file, as well as the directories *src/main/java/de/unioldenburg/ses* and *src/main/resources*. Invoking Maven in that directory again, e.g. with the *compile* or *install* goal, will cause SCAffolder to generate all the necessary boilerplate code to implement the JavaParser component, exposing all the services it declares in the provided SENSEI model file.

The archetype is only needed once to initialize a project. Still, the command line interface can be a bit bulky. Modern Java IDEs that have good support for Maven, like NetBeans, offer an alternative, by including the option to create archetype-based projects through its standard *New Project* wizard. This allows to simply search or browse for the desired archetype, and then enter the basic project information (corresponding to the last four parameters in Figure 14.12). Figure 14.13 shows a screenshot of the NetBeans wizard, being used to create the exact same JavaParser component project as before, but in an arguably more convenient way.




(a) High-level service orchestration for toolchain execution.




(b) Service orchestration to generate and run toolchains.

Figure 14.14: SENSEI service orchestrations for executing toolchains.

For a component, this is the starting point for tool developers to fill the generated method stubs with the appropriate logic, either adapting an existing tool, or creating a completely new one. The Maven project created by the archetype can be extended in virtually any way desired, e.g. by adding additional files and folders, or extending and modifying the pom.xml file. Even the SCA artifacts can be extended, as long as the auto-generated parts remain intact. Depending on the chosen execution environment, it may suffice to *install* the project into the (local) Maven repository to make it available for generated SENSEI toolchains to use. Alternatively, an SCA-enabled application server can be used, e.g. Apache Tomcat with Tuscany SCA running inside.

Generated toolchains are basically ready to go after initializing the project through the archetype, and running the build up to the packaging phase. This produces a *jar* file (alternatively, *war* or *zip* files are also possible) that can be referenced in other projects, to use the composer's exposed API to invoke the toolchain. In addition, the composer exposes an SCA service itself, so it integrates well into larger, SCA-based service-oriented applications. The generated toolchain can also be complemented with a manually created user interface, e.g. if the composer is to be deployed to a web application server, a web UI can be created within the same project, using any arbitrary technology available in the Java world for this. Examples of using these facilities in practice are provided in Chapter 15.

## 14.5 The SENSEI Model Interpreter SNOrcInS

An alternative to creating toolchains through code generation is to directly *interpret* SENSEI models. Such an interpreter was implemented by Küpker [2015] for his master's thesis and called *SNOrcInS*[5]. Before briefly describing its design and implementation, the specification of SCAffolder is revisited to illustrate how the two SENSEI processors relate to each other.

In Figure 14.2, the ability of SCAffolder to generate toolchains was specified. Figure 14.14 builds on this, but additionally considers that toolchains are executed after they are generated. In Figure 14.14, a new service *Run SENSEI Toolchain* is instantiated. Figure 14.14 drills down into this service, modeling it as the sequence of *Generate Toolchain Artifacts* (which in turn is further specified by the orchestration shown in Figure 14.2(b)), followed by *Invoke Toolchain*.

In this picture, both SCAffolder and SNOrcInS can be considered alternative implementations of the *Run SENSEI Toolchain* service: In SCAffolder, this is broken down further by the aforementioned orchestrations, whereas SNOrcInS can be considered a component that implements it directly. Therefore, on a functional level, the two are the same. There are use cases in which either code generation or runtime interpretation may be the better approach, depending on non-functional requirements, like runtime performance, ease of integration with existing technology stacks and deployment environments, or the ability for dynamic toolchain modification.

SNOrcInS does not target SCA, but the WSO2 middleware platform [*WSO2* 2020], which was used in the context of the NEMo research project (see Chapter 16). Like SCAffolder, it is implemented in Java, and relies on JGraLab and GReQL, as well as the API generated from the SENSEI metamodel. Its central classes, along with their most important methods, are shown in Figure 14.15. SNOrcInS also reuses the existing CompositionFinder implementation by Meier [2014a].

The main entry point to SNOrcInS is the *orchestrate* method of class *SNOrcInS*. From here, the sub-graph representing a specified orchestration contained in the input SENSEI model is traversed recursively, following the *visitor pattern* [Gamma et al., 1995, pp. 331ff].

The semantics of control flow constructs are implemented in Java; for example, Figure 14.16 shows the code used to handle conditional branches (*AlternativeServiceInstance*) in SENSEI orchestrations. To retrieve elements from the SENSEI model, SNOrcInS uses GReQL queries extensively, each of which are embedded into Java classes. Here, an instance of *AlternativeQuery* is used to retrieve the list of *Alternative* elements with incidences to the currently handled *AlternativeServiceInstance* instance (Line 303). These alternatives are iterated over, retrieving and evaluating the guard expression for each of them (Lines 307 through 310). The first alternative with a guard that evaluates

[5] *SENSEI Orchestration Interpreter Service*.

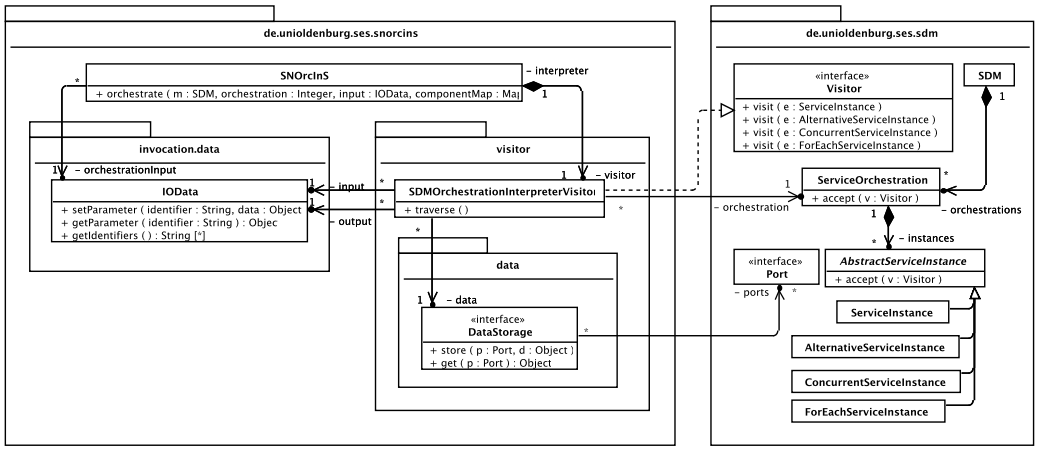



Figure 14.15: The central classes of SNOrcInS, and some of its dependencies into the SDMTGraph API. Based on [Küpker, 2015, p. 35]

.

```
300 @Override
301 public void visit(AlternativeServiceInstance e) {
302     // Get all alternatives
303     List<Alternative> alternatives = (new
    ↪  AlternativeQuery()).query(e.getId(), datagraph);
304     log(e, "Searching for contained alternatives...");
305     // Evaluate the first guard that passes to true and orchestrate
    ↪  instances contained within
306     GuardEvaluator evaluator = new GuardEvaluator();
307     for (Alternative a : alternatives) {
308         String guard = a.get_guard();
309         log(e, "Evaluating guard '" + guard + "' from Alternative#" +
    ↪  a.getId() + "...");
310         if (evaluator.eval(guard)) {
311             log(e, "Guard evaluated to true. Executing control flow
    ↪  within alternative...");
312             indentationLevel++;
313             a.accept(this);
314             indentationLevel--;
315             return;
316         }
317     }
318 }
```

Figure 14.16: The *visit* method implemented in class *SDMOrchestrationInterpreterVisitor* of SNOrcInS for handling *AlternativeServiceInstance* elements. Implemented by Küpker [2015].

```
260 // Get and invoke component
261 Set<Component> componentCandidates = componentMap.get(e);
262 IOData componentResult = null;
263 for (Component component : componentCandidates) {
264     log(e, "Assessing " + component.get_name() + " for possible
    ↪ invocation");
265     ComponentWrapper wrapper =
    ↪ wrapperFactory.createWrapper(component);
266     if (wrapper.canHandle(componentInput)) {
267         log(e, "Invoking component '" + component.get_name() + "' at "
    ↪ + component.get_location() + "...");
268         componentResult = wrapper.invoke(e, componentInput);
269         log(e, "Invocation of '" + component.get_name() + "'
    ↪ complete!");
270     } else {
271         log(e, "Component " + component.get_name() + " cannot handle
    ↪ given input");
272     }
273 }
274 if (componentResult == null) {
275     throw new NoInvokableComponentRuntimeException("Could not find a
    ↪ component for ServiceInstance#"
276             + e.getId()
277             + " which can handle the current input on the data flow:"
278             + componentInput);
279 }
```

Figure 14.17: The *visit* method implemented in class *SDMOrchestrationInterpreterVisitor* of SNOrcInS for handling atomic *ServiceInstance* elements. Implemented by Küpker [2015].

to true is *visited* again (Line 313), i.e. it and all its potential children of this subtree are handled recursively by dedicated visit methods.

The leafs in the traversed orchestration tree are *ServiceInstance* elements, which need to be invoked on implementing components. The code that selects appropriate components and invokes services on them is shown in Figure 14.17. SNOrcInS relies on a mapping of service instances to components, provided by the CompositionFinder and stored in the *componentMap* field (Line 261). All candidates are then iterated over, and the first one that is able to handle the provided input data is used to invoke the service (Lines 263 through 273).

SNOrcInS has adopted the tool adapter concept from SENSEI and SCAffolder, but due to its different target platform, and the dynamic nature of the interpretation approach, the interface that components have to implement is different. All SNOrcInS-compatible components implement the same, generic interface *ComponentWrapper*. Because of this generic nature, the signature of its *invoke* method requires a *ServiceInstance* as argument to identify which service instance exactly is to be executed. It further takes

and consumes *IOData* instances to represent all actual input and output parameters.

There is another method, *canHandle*, enabling SNOrcInS to ask available components to inspect the input data passed in, and report whether it thinks it can handle it before the actual service is invoked. This concept is borrowed from SCAffolder, which realizes a similar pattern. However, SCAffolder takes capabilities and restrictions into account, whereas SNOrcInS currently does not.

The generic component wrappers of SNOrcInS offer an additional degree of freedom over the tool adapters required by SCAffolder, as they do not actually require a fixed commitment to a single target platform. Whereas SCAffolder relies on SCA to bind and connect to components, SNOrcInS moves this logic into the component wrappers. This enables the use of component registries containing components hosted on different target platforms, and mixing them while executing a single orchestration. In particular, it should be easy to implement a SNOrcInS component wrapper that binds to existing components using SCA and SCAffolder conventions.

This higher flexibility comes, to some degree, at the expense of component uniformity (cmp. the *Uniform Interfaces* requirement), as there is no single component model to which all components adhere. SNOrcInS can therefore make less assumptions about, and has less direct control over, components. Also, component wrappers are potentially more complex, as there is no unified middleware providing utility functions like SCA does for SCAffolder. Of course, a middleware like SCA or WSO2 can still be adopted by convention, either for all, or only some of the components in a SENSEI component registry.

## 14.6 Summary

SCAffolder provides all the functionality of the SENSEI processors to find compositions, and generate stubs and compositions (toolchains). It therefore serves as a proof of concept, and as the basis for applying the overall approach in practice.

As a research prototype, the objective was not to achieve product maturity, but rather to proof general feasibility, and to gain further insights into the SENSEI approach in general, and the design of processors like SCAffolder in particular. The most prominent example of an improvement made after an early implementation iteration is the separation of the CompositionFinder as a separate component. With SNOrcInS, a completely different design was successfully demonstrated to be able to take the same role as SCAffolder within the conceptual framework that is SENSEI, showing the versatility and universality of the approach. Both alternatives are based on the same principles that are the foundation of SENSEI; particularly, both are model-driven: while SNOrcInS does not follow the "classical" MDA structure (see Section 8.4) as much as SCAffolder does, the interpreter does rely on rigid models and metamodels, and is very literally *driven* by SENSEI models.

SCAffolder has been used to apply and evaluate SENSEI in practice for building the Q-MIG toolchain. SNOrcInS served the same purpose in the context of the NEMo project. Those applications will be described in Chapter 15 and Chapter 16, respectively. However, a first application was already shown in this chapter: the specification of SCAffolder in Figure 14.2 in terms of SENSEI service orchestrations is actually an application of SENSEI to itself. Of course, there is a bootstrapping issue, as initially there was no SENSEI processor to generate the toolchain corresponding to the modeled orchestrations, so SCAffolder was first implemented and integrated "by hand". As shown in Figure 14.5, SCAffolder can be broken down into SCA services and components. In fact, SCAffolder is now able to basically generate itself, by feeding it the orchestrations that specify it, together with a component registry containing the individual, corresponding components.

# PART V

# Evaluation

Part IV explained the SENSEI approach in depth, and its use has been illustrated using small examples. Furthermore, Chapter 14 presented a concrete realization of SENSEI's concepts in the form of the implementation SCAffolder, proving general feasibility. As mentioned in Section 14.6, SCAffolder, in an instance of "eating one's own dog food", has been designed as a SENSEI toolchain itself, and thus represents a first full-blown demonstration of the applicability and usefulness of the approach.

The following two chapters each present a different application scenario, for which SENSEI was utilized to create appropriate support through integrated toolchains, showing its scope of applicability in different domains:

**The Q-MIG project** was previously introduced (see Chapter 2), and an example from it is running through this whole thesis (Section 2.2). This case study describes the application of SENSEI to the project, by going through all steps of the toolchain-building process. It demonstrates the advantages of using SENSEI compared to the original, manually built tool support, as well as its ability to scale to industrial applications.

**The NEMo project** is aimed at improving mobility of citizens living in rural areas, by developing a mobility platform that offers its users innovative, potentially inter- and multi-modal mobility services. This case study presents the results of applying SENSEI to design business processes to be supported by the mobility platform, and to automatically integrate its components, accordingly. It shows that the principles of SENSEI can be applied to software development in general, and highlights the flexibility conferred by employing the approach.

Using these applications of SENSEI, the approach is validated against the objectives defined in Chapter 1. Sound arguments are given regarding the practical improvements SENSEI provides compared to "manual" tool integration efforts. More precisely, the following validation goals provide the guiding research questions for the case studies:

- **Feasibility:** Can the approach be technically realized?
- **Applicability:** Can the approach be practically applied in the intended domain, and on an industrial scale, while conferring its desired benefits?
- **Generalizability:** Can the approach be practically applied beyond its intended domain, to confer its desired benefits in a more general scope?
- **Utility:** Does the approach confer the desired benefits expressed by its objectives of *flexibility*, *reusability*, and *productivity*?

These goals will be revisited in Chapter 17, and discussed in the context of the two case studies. Finally, Chapter 18 will reflect on the contributions of this thesis, summarize the benefits and possible future of SENSEI in research and practice.

CHAPTER 15

# The Q-MIG Toolchain

The Q-MIG project was introduced in Chapter 2 as an example of a software evolution project in need of tool and toolchain-building support. During the course of the project, the tool support was originally built "manually". This chapter describes the process of building toolchains for Q-MIG using SENSEI, and compares the process and its outcome to the original process and toolchain to demonstrate the feasibility of the approach, and provide evidence for its benefits in accordance with the objectives of this thesis as stated in Section 1.2.

The aim of this chapter is to give a clear impression of how SENSEI has been applied to the Q-MIG project, rather than to provide a fully exhaustive description of all the models and code artifacts created during this case study. Therefore, only examples of modeled services and orchestrations, implementations, automatically generated code, as well as resulting toolchain deployments and execution results will be presented. Nonetheless, a comprehensive service catalog and a component registry, as well as all the orchestrations omitted here, are available for reference in Appendix A.1.

The outline follows the toolchain-building process originally established in Section 3.1, and mapped to the terminology of SENSEI in Section 9.4 (see Figure 9.4, page 156). A prerequisite step in that process is *goal determination*. Section 15.1 briefly recaps the Q-MIG project goals as originally introduced in Chapter 2, to then provide a more concrete overview of all the activities that required (integrated) tool support.

Then, the main steps of the toolchain-building process (recall Figure 9.4, page 156) applied to Q-MIG are described, starting with *service identification* (Section 15.2), and followed by *service orchestration* (Section 15.3), *service-component matching* (Section 15.4), *adapter creation* (Section 15.5), *transformer creation* (Section 15.6), and *composer creation* (Section 15.7). Results are summarized in Section 15.8.

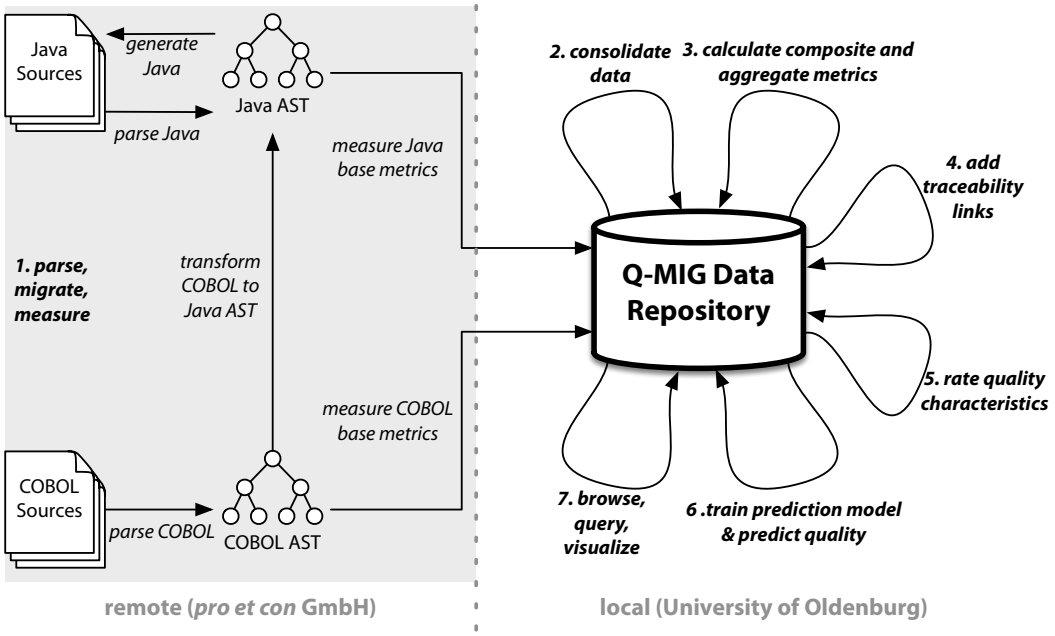



Figure 15.1: Activities in the Q-MIG project in need of integrated tool support.

## 15.1 Goal Determination

To briefly summarize Chapter 2, Q-MIG was a research project jointly conducted by the Software Engineering Group of the Carl von Ossietzky University, and industry partner *pro et con* GmbH. The main research interest of the Software Engineering Group was to complement the migration toolchain of *pro et con* with a quality control center to measure, compare, and predict the inner quality of software systems before and after migration.

Several tool integration challenges were observed during the project, including the lack of interoperability of existing software evolution tools, the need for interoperability and reusability even in ad-hoc, custom tooling, the essential significance of fully automated toolchains without gaps, and the technical complexities of tool integration upstaging the project's actual objectives. Q-MIG has thus been a central motivation for the creation of SENSEI.

Figure 15.1 depicts the central activities (in bold italics) of the Q-MIG project that required automation, arranged around the Q-MIG repository, containing structural descriptions of software systems undergoing migration and quality analysis, as well as associated quality measurements. The left-hand side (grey area) represents activities performed off-site by *pro et con*, providing the basic data for all subsequent quality analyses. The activities performed by the Software Engineering Group (right-hand side)

during the project have been broken down into six further major areas, seen on the right-hand side of the figure. The activities are as follows:

1. *Parse, migrate, measure*. This activity is the main source for the data in the repository, taking COBOL and Java software systems, parses them, and then calculates base metrics for them.
2. *Consolidate data*. This activity subsumes all steps dealing with unexpected data inconsistencies.
3. *Calculate composite and aggregate metrics*. This activity uses base metrics provided by Activity 1. to derive *composite metrics*, which combine different types of measurements to form new ones, and *aggregate metrics*, which combine the same type of measurements on lower-level software system elements (e.g. Java methods) to form measurements for higher-level elements (e.g. Java classes).
4. *Add traceability links*. This activity establishes traceability links between corresponding elements of legacy COBOL and migrated Java software systems, a prerequisite for comparative analyses of quality and different migration strategies.
5. *Rate quality characteristics*. This activity collects ratings for quality characteristics like modularity, reusability, etc., from human experts to have a baseline for, and establish correlations with, measured quality metrics.
6. *Train prediction model & predict quality*. This activity applies machine learning techniques to train prediction models, which are then used to extrapolate from the correlations found in the data, and applying them to new systems and migration strategies.
7. *Browse, query, visualize*. This activity contains all steps taken to make the raw data available to interested stakeholders in the form of HTML reports and visualizations like trend charts, scatter plots, bar and line charts.

It is important to note that the tooling to support these activities was developed incrementally during the project. In particular, data consolidation, machine learning techniques for quality prediction, and report generation and visualizations have been adapted and extended continuously. The in-house developed tools for base metric calculations – first and foremost, the Java metric calculator – also saw several releases. However, its development had to be stabilized and frozen at some point due to the lack of *integrated* tool support: every fundamental change to the tooling would have necessitated repeating *all* the activities, incurring unbearable manual effort. This underscores, once more, the objective of SENSEI to provide *flexible* toolchains amenable to change (Section 1.2).

The degree of process integration of the original tooling varies across these activities: data consolidation, for example, consists of several steps that are integrated tightly. This makes this activity highly automated and easy to perform, but it negatively affects evolvability. The tooling for data consolidation probably saw the most unforeseen

change requests, as issues with the delivered data were uncovered little by little, and thus became quite complex, and harder to extend every time. Other activities, such as quality prediction, provide little to no process integration, mainly due to lack of time. Performing machine learning experiments on the available data therefore remained rather cumbersome, and the extent of quality prediction activities was far more limited than originally planned, again, due to the large amount of time it took to create the tool support, overall.

Thus, the goals of applying SENSEI are:

1. To automate the individual activities of Q-MIG, as described above, to the fullest extent possible, and compare the original solution with the SENSEI-approach.
2. To automate commonly performed processes across these activities, and across the boundary between local and remote systems as distributed toolchain.

In the following, all major steps of the SENSEI toolchain-building process (Section 9.4) are described, starting with *service identification* in Section 15.2. Examples are given for each of the Q-MIG-specific goals of applying SENSEI named above. They will focus on data consolidation and metric calculation, i.e. activities 1., 2., and 3.. Select examples of the produced SENSEI artifacts, such as service descriptions, orchestrations, component registry entries, and generated toolchains, are also presented. As a reference, an overview of all of these artifacts is provided in Appendix A.1.

## 15.2 Service Identification

Since the application to Q-MIG was the first major use of SENSEI, there was no ready-to-use catalog to pick services from, so a new catalog had to be created first, i.e. services had to be identified and modeled from scratch. For this case study, the pragmatic bottom-up approach for service identification, introduced in Section 9.4, was used. As described there, each of the major Q-MIG activities was broken down into sub-activities, hierarchically, to identify service candidates. Then, services are described in conformance with the SENSEI metamodel to constitute a service catalog for Q-MIG.

This section will give examples for service identification from three of the activities introduced in Section 15.1:

**parse, migrate, measure** shows a "classical" software evolution activity, much more complex than the *base metric calculation* example used throughout this thesis;

**consolidate data** shows SENSEI applied to model tooling around a central repository;

**calculate composite and aggregate metrics** shows the power of *capabilities* by modeling the activity as a single, versatile service.

Together, these examples should be sufficient to get a good impression of how services were identified and modeled for the Q-MIG project. The same activities will be used to continue the examples during the subsequent step of service orchestration

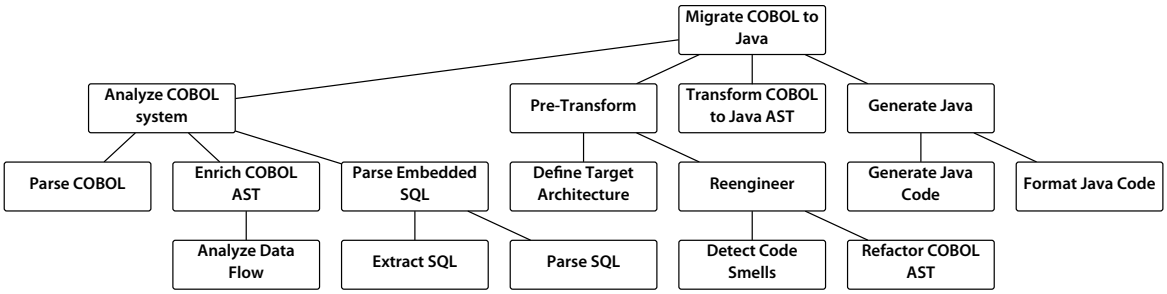



Figure 15.2: Activity decomposition of the COBOL-to-Java migration.

(Section 15.3). The interested reader may find services identified from, and described for, the remaining activities in Appendix A.

### 15.2.1 Services to Parse, Migrate, Measure

The left-hand side of Figure 15.1 shows multiple possible paths, starting from either a COBOL or a Java software system as main input, and producing either base metric measurements, or a Java software system. Two of these paths correspond to the *base metric calculation* example used throughout this thesis: starting from either a Java or a COBOL software system, which is parsed and metrics are calculated for it. For Q-MIG, the *CalculateMetric* service gained one more capability class, *SupportedGranularityLevel*, to evaluate metrics at different levels in a hierarchically decomposed software system (e.g. on class or package level in case of Java; see Section A.1.1 for all defined capabilities). Otherwise, the services remain as introduced back in Section 10.1.1.

This section will therefore focus on the common software evolution activity that is language migration, in this case from COBOL to Java. Even though this part was exclusively performed by project partner *pro et con*, and the integrated tooling was already in place, it is instructional to see how this process can be modeled and automated using SENSEI, instead.

Figure 15.2 shows a decomposition tree for this activity. Besides understanding the project-specific goals of the activity, deriving such a breakdown requires domain knowledge. In Q-MIG, employees of the industry partner *pro et con* provided the necessary software migration expertise, complementing the scientific insight of the Software Engineering Group. A first-level breakdown identifies four phases:

**Analyze COBOL system** prepares a model of the legacy system for further processing. It consists of *parsing COBOL*, *enriching the COBOL AST* by *analyzing data flow*, and *parsing embedded SQL*, by first *extracting SQL* and then *parsing SQL*. Even though the data flow analysis was the only kind of semantic enrichment performed in Q-MIG, knowing that this is a common type of post-parsing activity in software evolu-

tion projects (which turn an AST into an *abstract semantic graph*, ASG, compare e.g. Ferenc et al. [2001] and Lin, Holt, and Malton [2003]) led to the inclusion of another intermediate level in the decomposition tree.

**Pre-Transform** applies semantic-preserving changes to make the system more amenable to being transformed. It consists of *defining the target architecture*, and *reengineering*, which is further broken down into *detect code smells* and *refactor COBOL AST*. In Q-MIG, the target architecture was influenced in terms of the desired Java package structure that should result from the migration [Becker and Kaiser, 2014]. *Reengineering* tasks necessary before migration include, for example, removing GOTO statements as much as possible to ease migration to Java, and make the resulting program more structured [Becker and Kaiser, 2016].

**Transform COBOL to Java AST** performs the actual programming language translation. It is not broken down any further. While it is arguably the most central and computationally complex step in the overall migration process, it is considered atomic from a service modeling viewpoint.

**Generate Java** creates source code from the generated Java model (AST). It consists of *generating Java code* that is compilable, and then (optionally) *format the Java code* for better readability.

The activity breakdown provides the SENSEI service candidates, primarily by looking at the tree's leafs. The structure of the tree can be used to guide the hierarchical organization of (sub-)orchestrations. Inner nodes also become services, because service orchestrations are (composite) service instances, and therefore conform to a service.

To get from service candidates to services proper, they are modeled according to the SENSEI metamodel, i.e. input and output parameters and their types have to be identified. In addition, variation points can be identified and modeled as capability classes to abstract from project-specific needs and derive more generic services with wider applicability.

Figure 15.3 gives three examples of SENSEI services identified from the activity decomposition. Their parameters should be mostly self-explanatory, as are the data structures created for them, which in SENSEI are nothing more than names to be mapped to concrete, technical data types by components (see Section 10.1, page 164, and Section 12.1.3, page 200).

More interesting are the capability classes (compare Section 10.2.1): *ExtractEmbeddedCode* has two of them, one for the host language, and one for the embedded language, so that they may vary rather independently. In particular, implementations may be conceived that are able to identify embedded code (assumed to remain unparsed in host ASTs, stored in string-typed fields of its nodes) for arbitrary host languages. The opposite can also be considered, i.e. an extractor that identifies different kinds of embedded code in a particular host language based on certain host language statements.

*FormatCode* has a single capability class for the coding style, but none for the programming language, as the the two are co-dependent: the *Code Conventions for the*

| | | |
|---|---|---|
| **Service** | Name | ExtractEmbeddedCode |
| | Description | Extracts unparsed, embedded source code fragments from a host language AST. |
| | Input | ast : AbstractSyntaxTree |
| | Output | code : SourceCode |
| | Capability | HostLanguage = {COBOL} EmbeddedLanguage = {SQL} |
| | Classes | |
| **Service** | Name | FormatCode |
| | Description | Takes source code in a particular programming language and formats it according to the specified coding style. |
| | Input | style : CodingStyle<br>source : SourceCode |
| | Output | formattedSource : SourceCode |
| | Capability | CodingStyle = {JavaOracle} |
| | Classes | |
| **Service** | Name | TransformProgrammingLanguage |
| | Description | Transforms a software system in one programming language into a semantically equivalent software system in another programming language. |
| | Input | source : AbstractSyntaxTree |
| | Output | result : AbstractSyntaxTree |
| | Capability | LanguageTransformations = {COBOLtoJava} |
| | Classes | |

Figure 15.3: Three of the services modeled for Q-MIG's migration activity.

*Java Programming Language* [1999] are – obviously – only applicable to the Java programming language. This avoids nonsensical combinations (capability tuples) and is sufficient for this case study. This is a pragmatic judgement call, because there are coding styles that apply to a whole family of languages (e.g. indentation styles like *Allman*) – if this was to be supported as well, a different service modeling approach could have been considered.

Linked to the design of capability classes is the level of abstraction of services, in general. For example, the task of transforming ASTs from one programming language into another could be fulfilled by a highly generic model-to-model transformation service. Again, these decisions come with a certain margin of discretion – for Q-MIG, the more specific *TransformProgrammingLanguage* service seemed clearer and more appropriate.

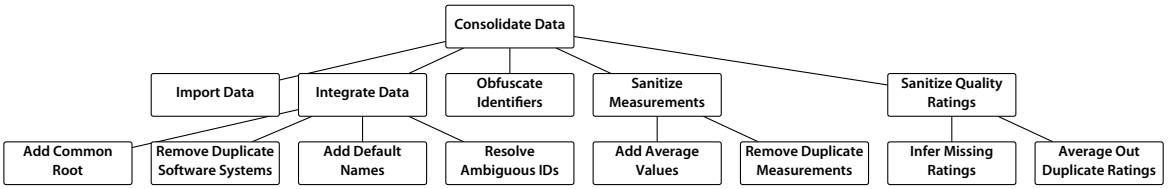



Figure 15.4: Activity decomposition of Q-MIG's data consolidation.

### 15.2.2 Data Consolidation Services

Migrating COBOL systems to Java, and measuring metrics on both legacy and resulting systems yields the data on inner software quality stored in the Q-MIG repository. Due to the sensitive, confidential nature of the source code being analyzed, these steps were performed by a team of industry partner *pro et con* at their offices. To ensure that the received data was complete and consistent, a data model was rigidly specified early on, and distributed and discussed among all project members. Still, some misconceptions about the process of migrating and measuring employed by *pro et con*, and the requirements of University of Oldenburg's Software Engineering Group regarding the data only became apparent late in the project, when a lot of data had already been generated and exchanged.

For example, the industry-scale COBOL systems used as examples each consisted of multiple programs. These were analyzed individually, which led to a disconnect in the data, the repeated analysis of copy books shared by the programs, missing names in entries representing COBOL *divisions*, and some non-unique identifiers.

As opposed to parsing, migration, and measurement, this activity and its individual tasks are highly specific to the Q-MIG project. Therefore, there are little to no opportunities for mapping them to more general, common software evolution tasks. Originally, the tool support was implemented in a couple of Java classes as the issues occurred. By the end of the project, this had led to several highly complex methods which were invoked during data import into the repository, to perform all the necessary steps on the data to make it consistent. A breakdown of the activity is shown in Figure 15.4, derived from the original code and its documentation.

**Import data** merges a new set of data with the data already present.

**Integrate Data** performs a series of steps on a given data set to ensure consistency. It consists of the following: Aggregating the structural descriptions of multiple COBOL programs belonging to a single, overall software system by *adding a common root* node to the hierarchical structure. *Removing duplicate software system* entries each representing the same copy book. *Adding default names* for divisions, if missing, based on the division type (identification, data, environment, or procedure). *Resolving ambiguous IDs* by adding contextual information to the ID strings during data integration.

| | | |
|---|---|---|
| Service | Name | QMIGImportData |
| | Description | Merges all the provided data model instances into the specified, existing repository. |
| | Input | repository : QMIGDataModel<br>importData [*] : QMIGDataModel |
| | Output | updatedRepo : QMIGDataModel |
| | Capability | – |
| | Classes | |
| Service | Name | QMIGInferMissingRatings |
| | Description | Tries to infer missing rating values by exploiting certain redundancies in the hierarchical subsystem structure, and by using the average of subsystem ratings, if the parent system's rating value is missing. |
| | Input | repository : QMIGDataModel |
| | Output | updatedRepo : QMIGDataModel |
| | Capability | – |
| | Classes | |

Figure 15.5: Two of the services modeled for Q-MIG's data consolidation activity.

**Obfuscate Identifiers** replaces all names of software systems and subsystems (files, programs, COBOL divisions and sections, etc.) with generic names that eliminate any chance of positively identifying the analyzed systems, or their owners. This was a confidentiality requirement to protect the interests of the clients of *pro et con*.

**Sanitize Measurements** removes inconsistencies in stored metric values. It consists of *adding average values* for missing metric values of software systems that can be reasonably recovered from corresponding metric values stored for its subsystems, and *removing duplicate measurements* occurring due to overlap of software systems that were analyzed separately.

**Sanitize Quality Ratings** removes inconsistencies in values of quality characteristics rated by experts. It consists of the following: *Inferring missing ratings* by assuming an average of all rated subsystems for parent systems, and by leveraging data model redundancies – there can be elements in the subsystem hierarchy essentially referring to the same artifact, e.g. a COBOL *program* and the *source file* that contains the program. And *averaging out duplicate ratings*, if the same expert had given multiple different ratings for the same software system and quality characteristic.

All leafs of the decomposition tree have been modeled as SENSEI services. The full listing can be found in Appendix A (each service has been given the name prefix "QMIG" to group them together in the catalog). An example is shown in Figure 15.5.

Many of the services share the basic structure of the *InferMissingRatings* service, having one input parameter to accept the repository in its current form, and one output parameter to return it in its modified state. Because of the project-specific nature of these services, they do not require capability classes to model variation points.

As depicted in Section 15.1, most Q-MIG activities are arranged around a central data repository. At first sight, the repository-centric architecture might seem like a bad fit for SENSEI, as it has explicit data flows, and, more importantly, its services are supposed to be stateless. The question arises whether this breaks fundamental principles of SENSEI, and with what consequences. Data consolidation is particularly well-suited as an example in this matter, as not only the activity as a whole, but all the individual services are actually modeled to take the repository as input, and they output an updated repository.

This service design is already part of the answer: by making the repository an explicit part of each service's signature, they remain formally stateless, i.e. their output only depends on their input, and not some internal state. SENSEI is very flexible in terms of the underlying data architectures it can be applied to. However, it should also be noted that more complex issues not needed in the context in Q-MIG, like ensuring transactionality across the execution of a service or orchestration, cannot be guaranteed using SENSEI alone. This discussion will be taken up again in Section 15.8.

### 15.2.3 Composite and Aggregate Metric Services

Base metric calculation works on the software systems under study, directly, to measure its properties. In addition, further metrics can be derived from these basic ones: either, a software system's value for a metric is inferred from the same metrics' existing values of its subsystems (aggregated metric), or a metric value is calculated based on the values of other metrics of the same software system (composite metric).

Even though this is also a technically a calculation of metrics, it is both syntactically and semantically different from the *Calculate Metric* service that is used for base metric calculation (see Section 10.2, particularly Figure 10.5, page 171). The latter analyzes ASTs of software systems to derive metric values. Composite and aggregate metric calculations work on the Q-MIG repository, i.e. the input data is markedly different. There is also no analysis of the input. Instead, the base metrics stored in the repository are appropriately assigned to variables present in the calculation rules for composite and aggregate metrics, to then execute these calculation rules. Because of this differences, it seemed more sensible to model a separate service, as opposed to reusing the existing one, and extending it with appropriate capabilities and input / output data structures.

The calculation of composite metrics and aggregate metrics is, however, very similar, which is why they are modeled by a single service to *Calculate Derived Metric* (this is also why there is no decomposition tree for this activity). The service is shown in Figure 15.6. Its signature is similar to that of *Calculate Metric*. The capability class

| Service | | |
|---|---|---|
| | Name | CalculateDerivedMetric |
| | Description | Derives values for the specified composite or aggregate metric for all software systems and their subsystems found in the repository that miss the value, if the necessary base metric data is present. |
| | Input | repository : QMIGDataModel<br>metric : Metric |
| | Output | updatedRepo : QMIGDataModel |
| | Capability Classes | SupportedMetrics = {ModularityRating, ReusabilityRating, AnalysabilityRating, ChangeabilityRating, ModificationStabilityRating, TestabilityRating, ComplexityRating, PortabilityRating, ELOC, McCabe, HalsteadVocab, HalsteadSize, HalsteadVolume, HalsteadDifficulty, HalsteadEfforts, HalsteadErrors, HalsteadTestingTime, CommentsPercentage, ClonesPercentage, AverageLCOM, AverageComplexityPerUnit, AverageUnitSize, AverageNumberSubclasses, AttributeHidingFactor, MethodHidingFactor, AverageMethodsPerClass, SLOC, NumberEmptyLines, NumberLinesWithOnlyBrackets, NumberCommentLines, NumberDecisions, NumberDistinctOperators, NumberDistinctOperands, NumberOperatorInstances, NumberOperandInstances, NumberGotos, CloneLines, LCOM, NumberClasses, NumberSubclasses, NumberAttributes, NumberHiddenAttributes, NumberMethods, NumberHiddenMethods, ITD, SwitchesFromGotos, UnconvertibleGotos, ConditionalOperators, CBO, SLOCWithoutSQL, ELOCWithoutSQL, NumberAttributesAccessed, SumDistinctAttributesAccessed, NumberSQLLines} |

Figure 15.6: *Calculate Derived Metric* service modeled for Q-MIG's composite and aggregate metric calculation activity.

*SupportedMetrics* also models the distinction between composite and aggregate metric calculation, since each metric is either one or the other.

Even though the long list of capabilities representing all the metrics required in SENSEI might seem a bit unwieldy, compared to a single service per metric it is actually quite compact. More importantly though are the benefits in terms of flexibility afforded by the capability model: it allows domain experts to concisely specify which metrics are actually needed in a given orchestration, without having to model the potentially extensive control flow structures for deciding when to use which service – SENSEI will do this for them. Conversely, tool developers specify which metrics their tools support, which enables SENSEI to mix and match components that together satisfy the given requirements – again with no need for manual intervention.

Because the calculation is not based on the analysis of a software system, a capability class to model programming languages, which is present on *Calculate Metric*, is not needed here. For the same reason, the capability class for supported granularity levels is omitted: Even though these levels are still present in the Q-MIG data model, they are uniform, i.e. the actual granularity level does not change the manner of metric calculation.

## 15.3 Service Orchestration

Having filled the service catalog, orchestrations modeling the project activities to be automated can be created. With SENSEI, it is easy to create, modify, and extend orchestrations. This means there does not have to be a single toolchain that supports *everything*. Rather, orchestrations can be created on demand, and can be evolved as the project progresses. The following orchestrations must therefore be considered selected examples and "snapshots" at a certain point in time. Had SENSEI been used in Q-MIG from the outset, the orchestrations would have evolved incrementally, just as the actual, "manually" crafted tool support did. The orchestrations created for this case study cover the tool support that had been developed by the end of the project, but go beyond it in terms of integration and automation, as will be shown in the following.

In the following, orchestrations are presented for the *parse, migrate, and measure* (Section 15.3.1) and the *consolidate data* activities (Section 15.3.2), continuing from the previous section. In addition, an example of an orchestration spanning multiple activities is shown to demonstrate the high degree of automation fostered by SENSEI. This orchestration activities to *parse, migrate, measure*, *consolidate date*, *calculate composite and aggregate metrics*, *add traceability links*, and *browse, query, visualize*, to produce a migration quality comparison report from a COBOL software system. Further orchestrations can be found in Appendix A.1.2.

### 15.3.1 Orchestrations to Parse, Migrate, and Measure

Figure 15.7 shows an orchestration to calculate base metrics – similar to the *base metric calculation example* used throughout the thesis, with two differences: There is an additional capability class *SupportedGranularityLevel* in service *CalculateMetric*, as mentioned in Section 15.2.1, and the *Parse* service is omitted from this orchestration. This makes it more reusable, as it can now be used to calculate base metrics on an AST generated by parsing source code, but also on ASTs produced by a language migration, for example. The new orchestration induces the service *QMIGCalculateBaseMetrics*, which can be instantiated in other orchestrations, to hierarchically model more and more complex processes without sacrificing clarity and consiceness.

Another simple orchestration to be used as a building block is depicted in Figure 15.8. It models the reengineering sub-activity identified in the decomposition tree

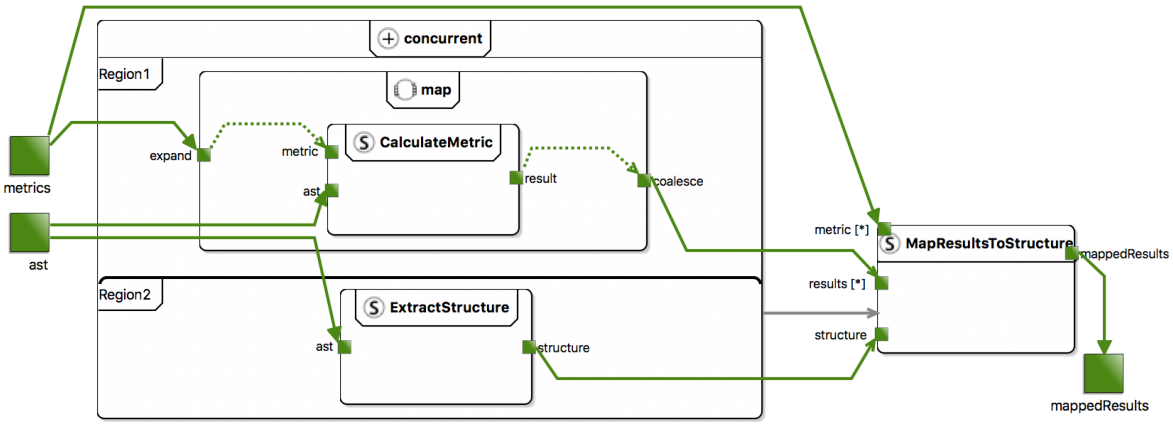



Figure 15.7: Q-MIG base metric calculation orchestration, without parsing (*QMIGCalculateBaseMetrics*).

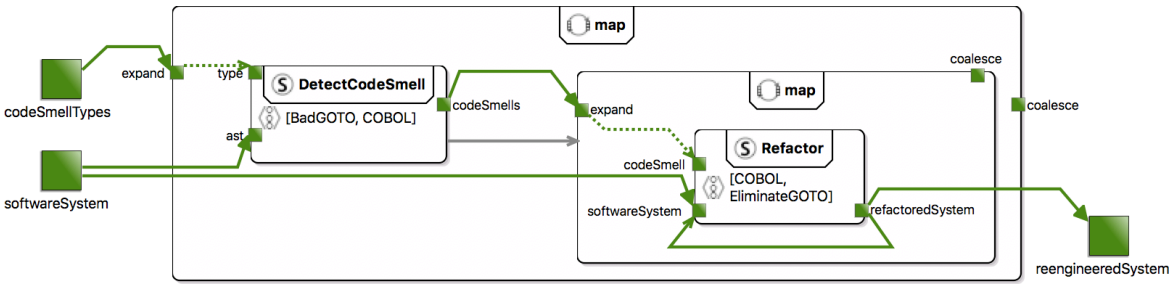



Figure 15.8: Q-MIG reengineering orchestration (*QMIGReengineer*).

shown earlier in Figure 15.2. The service induced by it is stored as *QMIGReengineer* in the service catalog.

The orchestration has only two service instances, one of *DetectCodeSmells*, and one of *Refactor*. The outer *map* loop iterates over all types of code smells provided as input. For each type, the provided software system is searched for instances of that code smell. The resulting list of code smells is iterated over by the nested *map*, and each one is removed by applying an appropriate refactoring.

Note that the *coalesce* ports are not utilized in this orchestration. This is because the changes are applied consecutively on the same software system. To do this, the output of the *Refactor* service instance is looped back to its input. SENSEI's data flow semantics make this work, as outgoing data flow edges are processed after each execution of the corresponding service instance, by copying the output data to the flow's destination, potentially overwriting any data that was previously stored at that port.

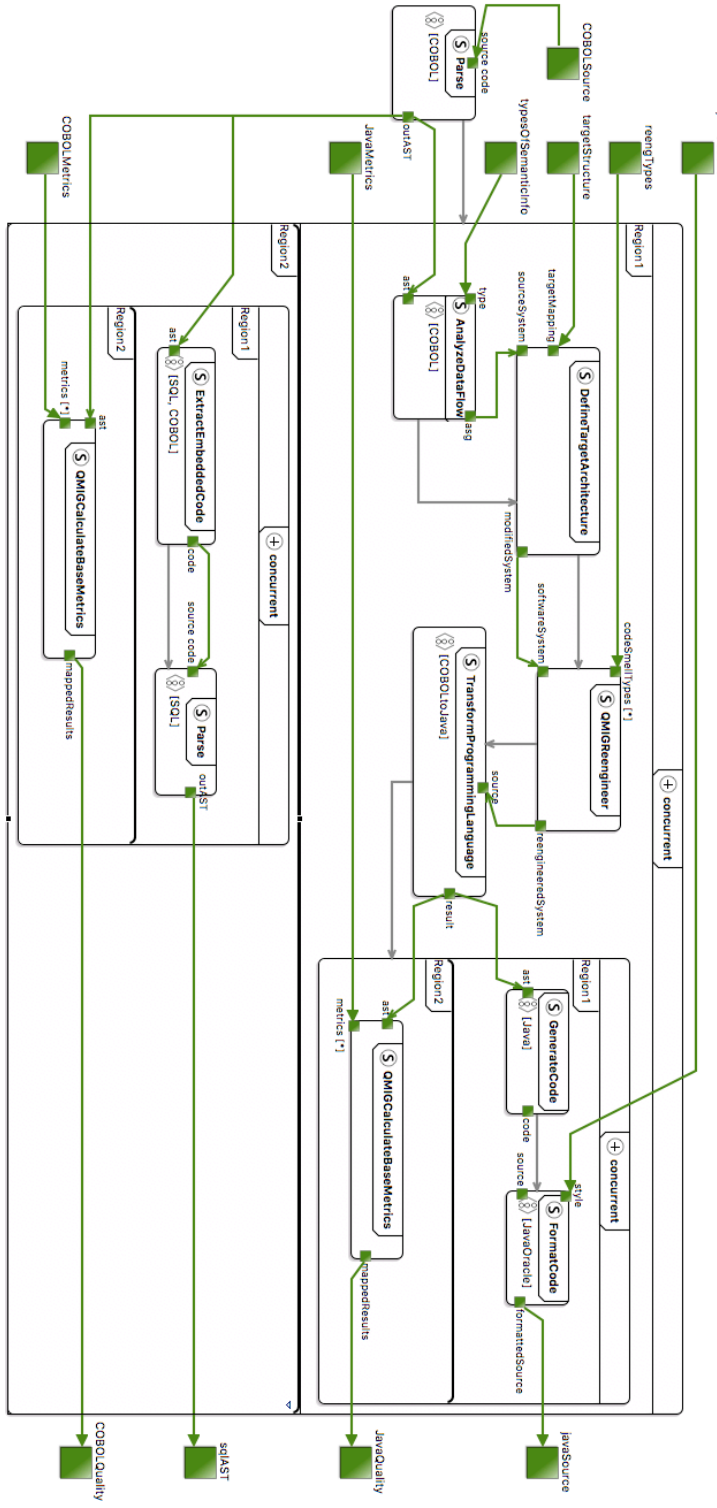



Figure 15.9: Orchestration covering parsing, migration, and quality measurement of COBOL and migrated Java systems.

These two orchestrations are reused in Figure 15.9, a much larger orchestration that covers the complete *parse, migrate, measure* activity of Q-MIG. As such, it certainly comes close to the upper size limit, both in terms of what can be fit on a screen (or page), as well as what can still be considered clear enough to be easily comprehensible. The orchestration could easily have been broken down further into sub-orchestrations, though. However, the goal here was to provide an example of a markedly more complex orchestration than has been shown up until now. Just like all SENSEI artifacts in this chapter, it was created with the SENSEI editor (Chapter 13), demonstrating its ability to handle large orchestrations, as well.

The orchestration models an activity that starts with *parsing* the source code of a COBOL software system. The resulting AST is passed on to three processes: the top half of the figure models the pre-migration analysis followed by the actual migration to Java. In the bottom half, the AST is analyzed for *embedded SQL* code, which is then *parsed*, and concurrently, the *base metrics* are calculated. This is an instance of the *QMIGCalculateBaseMetrics*, modeled previously as orchestration (Figure 15.7).

The pre-migration steps start with an instance of *data flow analysis*, which enriches the AST with semantic information. The resulting ASG is further decorated with a mapping of its element to a *target Java system structure*. Also in preparation for migration, the legacy system is cleaned up by applying *reengineering*, e.g. removing GOTO statements to ease migration to Java. Then, the actual *COBOL-to-Java migration* is performed. Following this, the resulting Java AST is passed to two concurrent processes: the one on the top *generates code* and then *formats* it, yielding source code of a Java software system as the final migration result. On the AST produced by the migration, base metrics are calculated, as well, reusing the *QMIGCalculateBaseMetrics* again.

This orchestration fully integrates and automates multiple sub-activities. It therefore has quite a lot of inputs (seven) and outputs (four). One minor shortcoming of the current SENSEI tooling is that there is no convenient way to specify constant values for input ports (also pointed out by Küpker [2015, p. 65]). This would be useful, e.g. for the *javaCodeFormat* input port. Then again, it is not much effort to provide that constant value, and having an input port exposed allows for potentially more flexible reuse.

### 15.3.2 Orchestrations to Consolidate Data

The data consolidation tasks are orchestrated by simply chaining them one after the other. For this activity, the orchestrations have been broken down exactly according to the corresponding decomposition tree (Figure 15.4): all leaf nodes had been identified as services in Section 15.2.2. Now, the root and the inner nodes are modeled as orchestrations.

Starting at the top this time, Figure 15.10 shows the high-level orchestration of the complete activity. First, the quality data sets are *imported* into an existing repository (or a new repository, if the corresponding port is left open). This orchestration assumes that

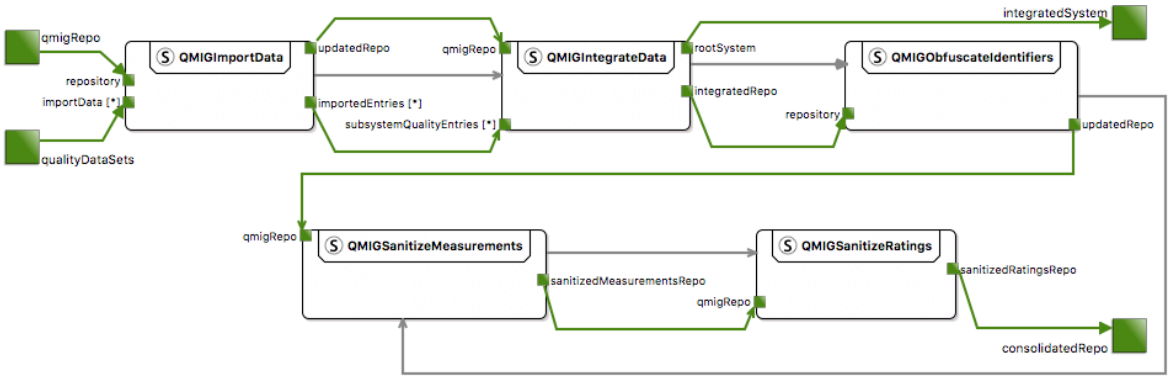



Figure 15.10: Q-MIG orchestration consolidate data (*QMIGConsolidateData*).

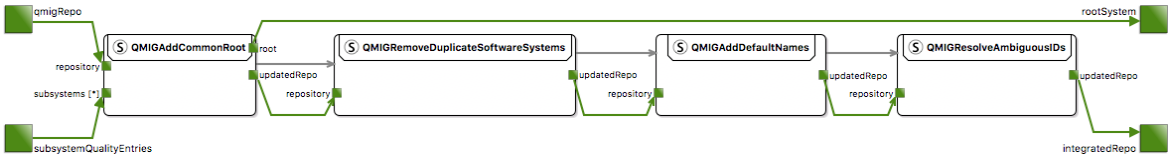



Figure 15.11: Q-MIG orchestration to integrate data (*QMIGIntegrateData*).

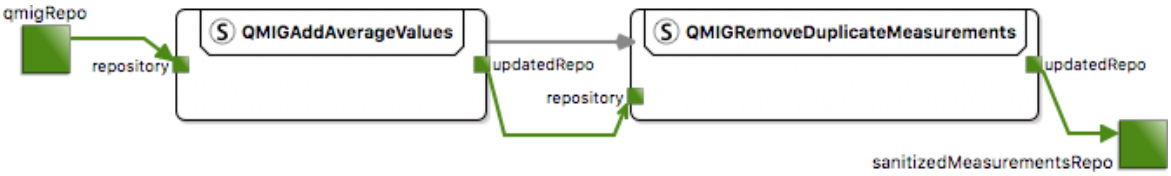



Figure 15.12: Q-MIG orchestration to sanitize measurements (*QMIGSanitizeMeasurements*).

all the quality data sets being imported are actually sub-programs belonging to a single software system. In the next step, the data is therefore *integrated* into a single entry. The data is then *obfuscated* according to the project's confidentiality requirements. Lastly, both the measured metric values and the quality ratings are *sanitized*.

The data integration step is further subdivided: the service instantiated in the top-level orchestration is induced by another orchestration, one level down in the activity decomposition shown in Figure 15.11. The service instances correspond to the services identified previously (refer to Section 15.2.2 for an explanation of what they do).

The two service instances of *QMIGSanitizeMeasurements* and *QMIGSanitizeRa-*

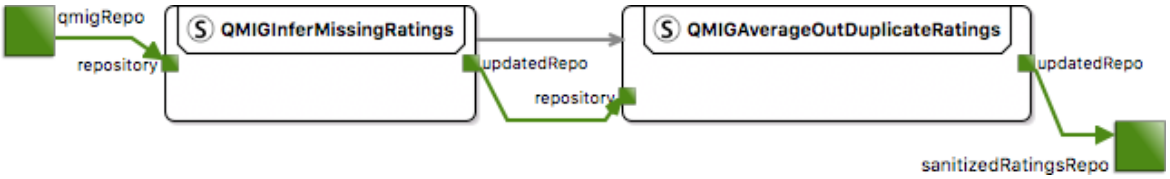



Figure 15.13: Q-MIG orchestration to sanitize ratings (*QMIGSanitizeRatings*).

*tings*, respectively, also correspond to a sub-orchestration, each. The former is depicted in Figure 15.12. The latter is shown in Figure 15.13.

The whole activity can be "flattened", by inserting the sub-orchstrations in place of their corresponding service instances in the main orchestration, which would yield one very long chain of service instances. The logical sub-division makes for less clutter, but it also serves to perform sub-activities independently.

For example, the data sanitation tools evolved later in the project, when the Q-MIG repository had already been filled with a lot of metric measurement data. This data might need to be sanitized, but the data import and integration activities do not have to be repeated (in fact, that could lead to duplications in the data, and subsequent erroneous analysis results).

It is also easy to change the top-level orchestration, or recombine the service instances and sub-orchestrations to form new orchestrations, e.g. to leave out some activities that are not always necessary. Finally, the clear separation into individual tasks that each handle a single problem with the data, that is facilitated by the service model, also greatly simplifies evolving the activity as the project progresses. For example, if the root cause for the presence of ambiguous IDs were to be fixed in the "upstream" tooling, the corresponding service instance can simply be removed from the orchestration.

### 15.3.3 Orchestrations to Generate a Quality Comparison Report

Several main activities have not yet been covered with orchestrations: the *calculation of derived metrics* and the *addition of traceability links* each only led to a single service being identified – they are either used on their own (which makes for rather trivial orchestrations), or as part of orchestrations that span multiple activities. The *rating of quality characteristics* is supported by an interactive tool. There is no place or need for automation by a SENSEI toolchain here. Finally, *browsing, querying, and visualizing* the quality data are actually three related activities, each of which corresponds to a single service with similar usage patterns as the services for derived metric calculation and traceability. The service descriptions that have not been introduced in Section 15.2 can be found in Appendix A.1.1.

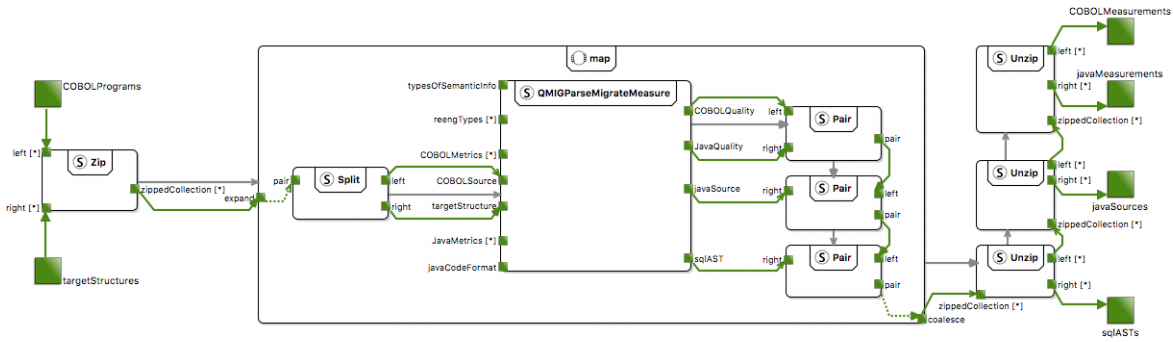



Figure 15.14: Q-MIG orchestration to parse, migrate, and measure multiple software (sub-)systems (*QMIGParseMigrateMeasureMultiple*).

To demonstrate performing these activities with SENSEI-based tool support, an orchestration that spans multiple activities is presented in the following. It represents an actual use case of the Q-MIG project: the quality analysis of both an original COBOL system (consisting of multiple programs) and the migrated Java system, resulting in a browsable quality comparison report that lists measured and derived metric values of corresponding COBOL and Java subsystems side by side.

Another activity-spanning orchestration was modeled, to perform quality analyses of COBOL systems, with subsequent prediction of expected metric values were it to be migrated to Java, and visualization of the results as bar charts or scatter plots. This orchestration is included in Appendix A.1.2.

For the former use case, a "helper" orchestration has been modeled, first. Figure 15.14 shows an orchestration to parse multiple COBOL programs (belonging to the same, overall software system), measure base metrics for each one, migrate them, and measure base metrics on each resulting Java (sub-)system, as well. In essence, this is just *QMIGParseMigrateMeasure* (Figure 15.9) embedded in a loop. However, there are multiple inputs and outputs, and the current orchestration language constructs neither provide an elegant way to iterate multiple collections in sync, nor to collect multiple, individual results into separate lists.

To solve this without extending SENSEI, four helper services have been modeled: *Pair* combines two data items, and *Split* pulls such pairs apart again. *Zip* and *Unzip* do the same for collections, i.e. create a single collection of pairs from to individual collections, and extract the original collections from a single collection of pairs, respectively.

As is obvious from looking at Figure 15.14, this leads to a very high degree of clutter in an orchestration that should otherwise be fairly straightforward. More powerful constructs for specifying data flow could fix this, for example having a *map* loop construct

with arbitrary many *expand* and *coalesce* ports. However, the orchestration language of SENSEI has been kept deliberately small, as its design was not a central objective of this thesis. Therefore, the helper services are necessary for what must be considered a makeshift solution.

Conversely, it should also be pointed out that it actually *is* possible to model and use the described orchestration with SENSEI. The helper services can be modeled quickly, and their implementation is straight-forward. Creating this functionality *within* the framework of SENSEI, instead of extending it in a more invasive manner, blends in seamlessly and naturally, following the "*everything is a service*" paradigm. A more concise way to specify such semantics would be highly desirable, but from a purely technical point of view, such extensions would be merely *syntactic sugar* – its absence has no fundamental impact on the soundness of SENSEI's foundations.

The orchestration induces the service *QMIGParseMigrateMeasureMultiple*, which is instantiated in the orchestration shown in Figure 15.15. This service instance is executed first, producing multiple COBOL and Java measurements. The SQL ASTs and the generated Java code is not used any further, here. The orchestration could be extended, either directly or by creating another orchestration that wraps around this one, to analyze SQL, as well, for example. Alternatively, a variant of *QMIGParseMigrateMeasure* could be used that skips SQL parsing and Java code generation, to avoid wasting resources on these when they are not actually necessary. For the sake of simplicity, the existing, previously introduced orchestrations were used as is.

Both the sets of COBOL and Java quality measurements are passed into instances of *QMIGConsolidateData*, which integrates each of them under a single root element. This is done sequentially, so the later service instance can work on the modified repository data produced by the earlier invocation. Although the respective changes should be completely disjoint, this avoids any chance for concurrent modification and potential conflicts.

Next, an instance of *QMIGAddTraceLinks* inserts traceability information that allows navigating between COBOL and corresponding Java systems and subsystems, and associated measurements. Following this, derived metrics are calculated: the semantics of this service specify that it always traverses the complete repository, calculating and inserting metric values wherever they are missing.

The last step produces a human-readable quality comparison report, incorporating all quality information available for the pair of COBOL and Java system. Since the service *QMIGGenerateReport* expects a list of measured software systems, another helper service is needed: *Concat* takes two elements or collections of the same, arbitrary type, and joins them into a single collection. In this case, since *QMIGGenerateReport* is project-specific anyways, its signature could also have been modified to better fit this use case. Another alternative would have been to create an additional service with basically the same semantics, but a different interface (implementing components would be able to provide both variants).

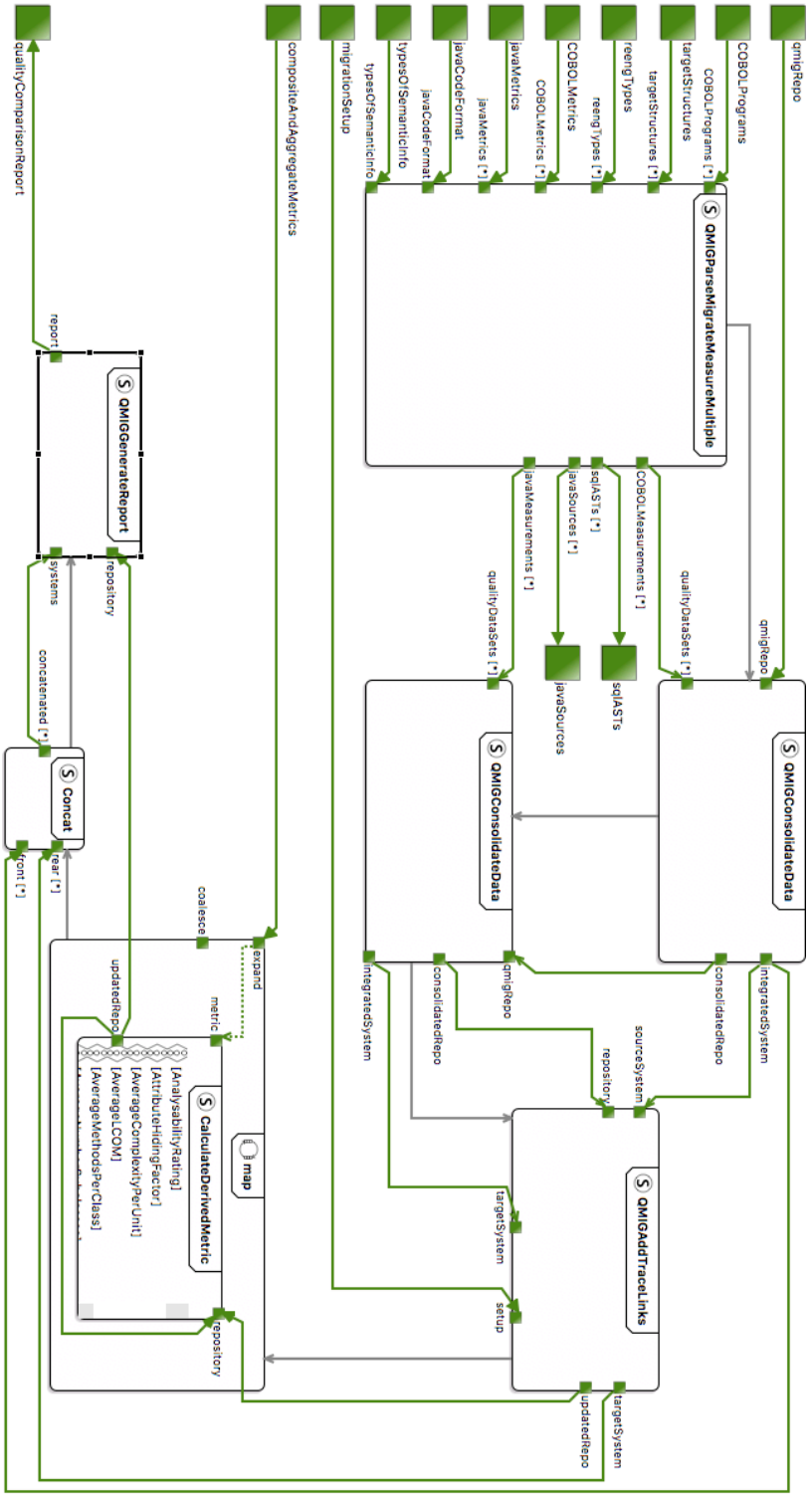



Figure 15.15: Orchestration to generate-quality-comparison-report.

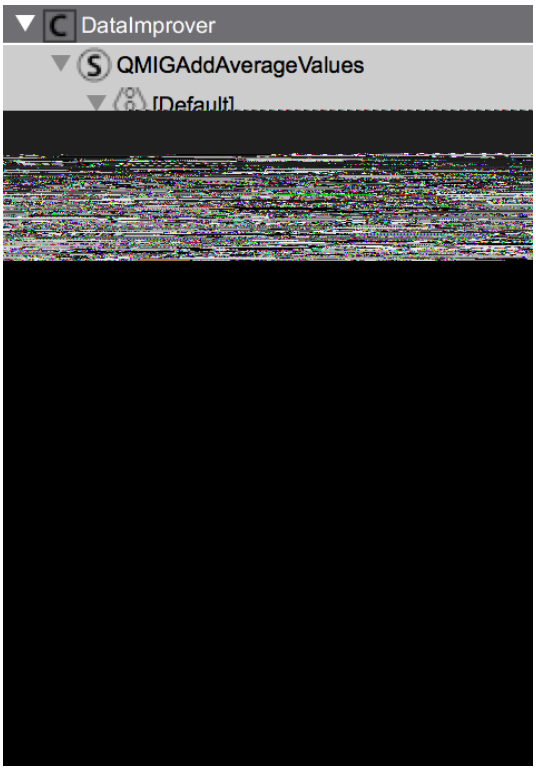



Figure 15.16: Screenshot of a component registry view in the SENSEI editor.

## 15.4 Service-Component Matching

This next phase in the toolchain-building process is about searching for appropriate tools to realize the tasks that have been identified and coordinated in the previous steps (Section 9.4). Using SENSEI, the process of finding implementations for services is partly automated, as described in Chapter 12. The remaining manual part, carried out by tool developers, is filling a component registry with the necessary information for service-component match-making.

In Q-MIG, a lot of project-specific services were identified, for which corresponding components were created specially for the project. While the ratio between usable standard components and necessary custom-built components may vary, the expectation is that most projects will require at least some individual logic.

A list of all the service-to-component mappings from the registry created for this case study is available in Appendix A.1.3. This list omits data definitions, as they would bloat it considerably with information that is required for technical reasons, but would provide little additional insight to the reader. Here, examples of the registry entries for two components are given to illustrate how this is modeled using the SENSEI editor.

| Property | Value |
|---|---|
| Data Definition de.unioldenburg.qmig.datamodel.AbstractSof... | |
| Format | de.unioldenburg.qmig.datamodel.AbstractSoftwareSystem |
| Model | |
| Protocol | |
| Representation | |
| Scope | [Default] |
| Target | Parameter root |
| Data Definition de.unioldenburg.qmig.datamodel.DataModel | |
| Format | de.unioldenburg.qmig.datamodel.DataModel |
| Model | |
| Protocol | |
| Representation | |
| Scope | [Default] |
| Target | Parameter repository |
| Data Definition de.unioldenburg.qmig.datamodel.DataModel 2 | |

Figure 15.17: Screenshot of the SENSEI editor's properties view, showing data definitions.

Figure 15.16 shows a screenshot of the SENSEI editor's component registry view. It shows the *DataImprover* component, which implements all services identified for the data consolidation activity. For these project-specific services, capabilities play a subordinate role. Therefore, all of the data definitions (see Section 12.1.3) for them are associated with a provided capability tuple consisting of the single capability *Default* from the capability class of the same name. SCAffolder only requires the *format* field, using it to bind SENSEI's conceptual data structures to concrete Java types. Two examples can be seen in Figure 15.17: these data definitions bind the parameters *root* and *repository* of the *QMIGAddCommonRoot* service to types defined by the Java classes *AbstractSoftwareSystem* and *DataModel*, respectively. The classes have to be referred to by their fully qualified name.

Another example, which makes use of the capability mechanism, is shown in Figure 15.18. Almost the complete base metric calculation was implemented within two custom tools in Q-MIG, the *Java Metric Calculator* and the *COBOL Metric Calculator*, the latter being realized by the project's industry partner *pro et con*. The only exception was for the *CloneLines* metric, for which the existing code clone detection tool *DuDe* [Wettel and Marinescu, 2005] was reused. In the original Q-MIG implementation, it was actually embedded within the Java Metric Calculator. For this case study, to achieve better reusability and flexibility, it was factored out again as a separate component, as is visible in the screenshot of the component registry view. Using *provided capabilities*, this component will only be taken into consideration for calculating clone lines on a directory-level, for either COBOL or Java systems.

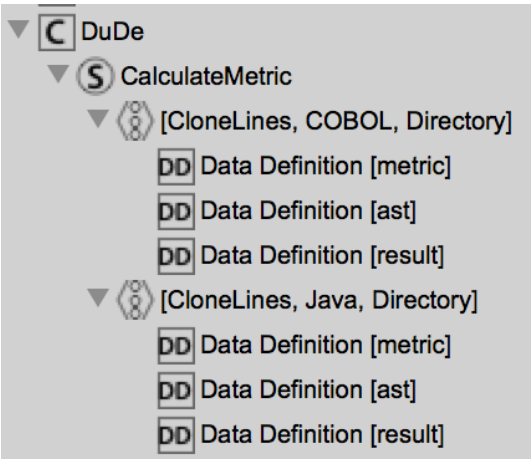



Figure 15.18: *DuDe* [Wettel and Marinescu, 2005] registered as component to provide the *CalculateMetric* service for the *CloneLines* metric.

## 15.5 Adapter Creation

Adapter creation falls to tool developers in SENSEI. For any given service that is being used, and that requires appropriate components implementing them, several cases can be distinguished:

1. There are one or more SENSEI-compatible components readily available. This is the best-case scenario: the tool developers have already taken care of adapting their component to SENSEI, and have added corresponding entries in the component registry. Such components will automatically be discovered, and there is no additional work to be done, i.e. the adapter creation step can be skipped.
2. There are one or more existing tools readily available, which have not been designed for SENSEI, however. In this case, an adapter component has to be created, which provides the interface required by the particular SENSEI processor. In the Q-MIG case study, SCAffolder is used, which supports adapter creation by generating the necessary boilerplate code from component registry entries. The required effort to implement such adapters is highly dependent on the tools to be adapted, and the interfaces they offer.
3. There are no tools available at all, or those that are available offer no interfaces that can be adapted to serve as SENSEI component (at least not with reasonable effort). In this case, the component as a whole has to be developed from scratch, not just an adapter.

In the original run of Q-MIG, a very large degree of the tooling was created specifically for the project (Case 3.). A major reason for not using more existing tools was

their lack of interoperability, e.g. inconvenient interfaces and too little customizability. At the time of the case study, this custom tooling has been implemented, i.e. the tools are available, but since SENSEI had not been available during Q-MIG, they do not feature the necessary adapters, yet (Case 2.). Obviously, since this case study was the first application of SENSEI on a larger scale, there were no SENSEI-ready components available (Case 1.).

Because of most tools being custom, in-house developments, their source code was available for this case study. On the one hand, this arguably simplified adapter creation, as it allowed to directly modify particularly unwieldy component interfaces. On the other hand, there were also cases in which the custom-build tool support was also the reason for more adapter creation effort, as some tools had not been build with individual reusability in mind. For example, the Java Metric Calculator implementation fuses generic, reusable functionality (corresponding to the *CalculateMetric* service) with very project-specific code, which made it harder to cut out component interfaces.

In the following, three examples for Q-MIG SENSEI adapters are given:

1. The *Java Frontend* adapter binds to the command-line interface of its tool.
2. The *DuDe* adapter uses an (undocumented) Java API.
3. The *Java Metric Calculator* adapter is built directly into the original code.

### 15.5.1 Java Frontend Adapter

The *Java Frontend* by *pro et con* provides the *ParseJava* service. It was available as a Microsoft Windows executable, only; without access to source code, and no exposed API, its command-line interface was adapted.

SCA can be extended with custom component implementation types, which can be used to support different programming languages, but also specific frameworks – Tuscany SCA, for example, supports Spring and BPEL out of the box [Laws et al., 2011, pp. 175ff]. In a similar manner, an implementation type for common styles of command-line interfaces could be created. For the Java Frontend adapter, the more direct route of using Java's *Process* API to invoke the tool was taken.

Figure 15.19 shows code snippets from an internal method of the adapter implementation. The complete adapter code is 229 lines long; the corresponding Java class implements the SCAffolder-generated interface *ParseJava*.

The first excerpt shown here (Lines 58 to 65) iterates over the source file names that were provided as input, and writes them to a "name file". This file is then passed to the *Java Frontend* instead of passing all file names individually (which can overwhelm the Microsoft Windows console). The command to be invoked, along with all parameters to be passed, is built in the second code snippet (Lines 78 to 82). The name file previously written to disk is passed as an argument in Line 81. The third code snippet (Lines 88 to 101) sets up the working directory, sets output streams to redirect to files, and then starts the process. When the process returns without abnormal interruption,

```
58      nameFileWriter = new OutputStreamWriter(
59              new FileOutputStream(NAME_FILE));
60      for (File localJavaFile : localJavaFiles)
61      {
62          nameFileWriter.write(localJavaFile.getAbsolutePath());
63          nameFileWriter.write("\n");
64      }
65      nameFileWriter.flush();
[...]
78      ProcessBuilder javaFEProcessBuilder = new ProcessBuilder(javaFEPath,
79              CLASS_PATH_PARAMETER, classPathString.toString(),
80              CONFIG_FILE_PARAMETER, configFile,
81              NAME_FILE_PARAMETER, NAME_FILE.getAbsolutePath(),
82              OUT_FILE_PARAMETER, OUT_FILE.getAbsolutePath());
[...]
88      javaFEProcessBuilder.directory(WORKING_DIRECTORY);
89      javaFEProcessBuilder.redirectError(new File(ERROR_OUT_FILE));
90      javaFEProcessBuilder.redirectOutput(new File(STD_OUT_FILE));
91      Process javaFEProcess = javaFEProcessBuilder.start();
92      try
93      {
94          javaFEProcess.waitFor();
95          return actualOutFile;
96      }
97      catch (InterruptedException ex)
98      {
99          Logger.getLogger(JavaParser.class.getName()).
100                 log(Level.SEVERE, null, ex);
101     }
```

Figure 15.19: Excerpts from the *Java Frontend*'s adapter implementation.

the file it wrote to disk, containing an XML representation of the abstract syntax tree is returned. As this is an internal method, this does not transfer control back to the component's invoker, directly. The result file's contents are read into memory, first, which are then transferred back as the component's response.

### 15.5.2 DuDe Adapter

The Java Frontend was only executed manually during the Q-MIG project's original run, and was not actually integrated into the rest of the toolchain. The clone detector *DuDe*, however, was integrated with the custom Java Metric Calculator tool, so an adapter connecting the tools' respective interfaces to each other had been created, already. The core of the necessary adapter logic is shown in Figure 15.20.

To perform the clone detection process, *DuDe* provides the *Processor* class, an instance of which is created and set up in Lines 70 to 73. The API requires to start the

```
68      public long evaluate(Entity[] dudeEntitites)
69      {
70          Processor dudeProcessor = new Processor(dudeEntitites,
71                  null, new LevenshteinMatchingStrategy(THRESHOLD));
72          dudeProcessor.setParams(DUDE_PARAMS);
73          dudeProcessor.setName("Dude Clone Detector (Levenshtein
    ↪ Matching)");
76          synchronized (dudeProcessor)
77          {
80              dudeProcessor.start();
83              try
84              {
87                  dudeProcessor.wait();
90              }
91              catch (InterruptedException ex)
92              {
93                  LOG.log(Level.SEVERE, "DuDe processor was interrupted
    ↪ ("
94                          + System.identityHashCode(dudeProcessor) +
    ↪ ")", ex);
95              }
96          }
97          //getMatrixLinesLength apparently returns the "relevant lines
    ↪ of code"
98          //see class lrg.dude.gui.GUI, method "showStatisticsAction".
99          long duplicateLines =
    ↪ dudeProcessor.getNumberOfDuplicatedLines();
100         return duplicateLines;
101     }
```

Figure 15.20: Excerpt from the *DuDe* adapter implementation (lines with fine-level logging omitted).

processor in its own thread (Line 80). Since this is not really desired here, the adapter simply waits for the processor to finish (Line 87). Provided that no exceptions occurred, the number of clone lines can then be retrieved from the processor instance.

In this code snippet, the input data – the software system under study – is represented by the *dudeEntities* parameter, and the output is a *long* number. This does not correspond to the types expected by the Java Metric Calculator, into which DuDe had been embedded for Q-MIG using this code. There is further glue code *transforming* between these different representations of data before and after invoking the tool, which has been omitted here. That is because, in accordance with the principles of SENSEI, this code constitutes *transformers*, not adapters. The transformer code, and the necessary refactorings to the original implementation which had it fused to the adapter, will be discussed in Section 15.6.

```
350 @Override public DataModel calculateAll() throws GraphIOException,
351         FileNotFoundException, JAXBException {
352     MetricCalculationResult result = new
    ↪ MetricCalculationResultImpl();
353     for (GranularityLevel level : GranularityLevel.values()) {
354         Set<Metric<SoftwareSystem>> metrics =
    ↪ registry.getMetrics(level);
355         for (Metric m : metrics) {
356             if (system.isLanguage(ProgrammingLanguage.JAVA)
357                     && m.isEvaluableOver(ProgrammingLanguage.JAVA)) {
358                 String metricName = m.getName();
359                 Map<Integer, Number> metricResultMap =
    ↪ m.evaluate(system);
360                 result.addMetricResults(metricName, metricResultMap);
361             } else {
362                 LOG.log(Level.INFO,"Metric "+m.getName()+" will not be
    ↪ "+
363                         "evaluated because the system does not have
    ↪ the "
364                         + "correct Programming Language");
365             }
366         }
367     }
368     DataModel dm = structureBuilder.build((SoamigGraph)
    ↪ system.getTGraph());
369     return baseValueRecorder.writeResults(dm, result);
370 }
```

Figure 15.21: Central method of the original Java Metric Calculator.

### 15.5.3 Java Metric Calculator Adapter

Like DuDe, the Java Metric Calculator is itself implemented in Java. Being a custom tool developed specifically for Q-MIG, the source code was readily available. Therefore, the corresponding SENSEI adapters were implemented as more immediate and natural extensions of the existing tool.

The original metric calculator tool only has a command-line interface with the following signature:

```
java -jar MetricCalculator-0.5.3.jar -e [--log <logLevel>] -s
↪ <SourcePath> -t <TargetPath> [-r <systemID> <setupID>]
```

Without going into details, it should be noticed that this interface offers no control over which metrics to calculate, or on which granularity levels. That is because the Java Metric Calculator always calculates *all* the metrics it implements. The "heart" of the tool is the *calculateAll* method shown in Figure 15.21. With SENSEI, this method, including the logic of some of the methods it invokes, is basically replaced by an orchestration, or more precisely, the *Composer* code that is generated based on it.

```
170 @Override
171 public Map<Integer, Number> calculatemetricSLOCJavaJavaClass(String
    ↪  metric,
172         byte[] ast)
173 {
174     LOG.info("Calculating SLOC of Java classes.");
175     return calculateMetric(metric, ast, JAVA, CLASS);
176 }
    [...]
242 private Map<Integer, Number> calculateMetric(String metric,
243         byte[] ast, ProgrammingLanguage lang, GranularityLevel level)
244 {
245     LOG.info("Calculating metric.");
246     try
247     {
248         SoftwareSystem sys = deserializeAST(ast);
249         MetricRegistry registry = registry();
250         LOG.info("Metric registry initialized");
251         Metric execMetric = registry.getMetric(metric, lang, level);
252         Map<Integer, Number> result = Collections.emptyMap();
253         if (execMetric != null)
254         {
255             LOG.info("Retrieved metric from registry");
256             result = execMetric.evaluate(sys);
257             LOG.info("Evaluated metric on specified system");
258         }
259         else
260         {
261             LOG.severe("Metric not found! (name='"+metric+"',
    ↪  language='"+lang+"', level='"+level+"')");
262         }
263         return result;
264     }
265     catch (IOException | XMLStreamException | ClassNotFoundException
    ↪  | GraphIOException ex)
266     {
267         throw new IllegalStateException(
268                 "metric registry could not be initialized.", ex);
269     }
270 }
```

Figure 15.22: Excerpts from the *Java Metric Calculator*'s adapter implementation.

The tool also directly outputs a complete Q-MIG data model instance, which was perfectly adequate for this project, but further entangles otherwise reusable parts (the calculation of a single metric on a specific part of a software system under study) with project-specific ones. So in essence, this implementation realized the complete base metric calculation (without parsing, see Figure 15.7) monolithically.

The new adapters shown in Figure 15.22 expose SENSEI and SCAffolder-compatible interfaces which makes its basic functionality available more directly, and independent from a single, project-specific use case. The interfaces auto-generated by SCAffolder define a separate method for each provided capability tuple declared in the component registry. These methods (Lines 170 to 176 show an example) all delegate to a single method *calculateMetric* (Lines 242 to 270), which initializes the required objects (Lines 248 to 252), deserializes the input data (Line 248)[1], and then invokes the actual metric evaluation (Line 256). The Java Metric Calculator also implements the *ExtractStructure* and *MapResultsToStructure* services; appropriate SENSEI adapters exposing their functionality for use with SCAffolder are implemented within the same class.

## 15.6 Transformer Creation

As most of the tools used for the Q-MIG project were custom-build, there is a limited number of required transformers: the tools that were built from scratch are all well-aligned to each other, and use compatible data formats. Some examples of transformers include the following:

- A transformer to turn the XML-based exchange format to represent COBOL and Java abstract syntax trees, as output by the parsers of *pro et con*, into the TGraph-based format used by the Metric Calculator.
- A transformer to turn an abstract syntax tree representation – either the XML- or the TGraph-based format – into the format expected by the DuDe clone detector.
- A transformer to turn the TGraph-based format of the Q-MIG data model into the XML exchange file format defined within the project, and another transformer to do the reverse.

Even though they had not been fully integrated into fully-automated toolchains, the original tool implementations already contain all the necessary code for these transformers. However, some refactorings were needed to properly cut the transformer functionality out.

One common issue was that the invocation of transformations happened nested inside of the methods implementing the functionalities. For example, the first step in the original code to calculate the *CloneLines* metric using DuDe was to transform the Metric Calculator's own software system representation to the format expected by DuDe. Fusing adapter and transformer logic in this manner impedes reuse, as now the input data has to be provided in one data format, only to be immediately transformed again. If the available input data is already in a DuDe-compatible format, it will have to be pointlessly transformed back and forth.

Another, related problem was concerning the interfaces defined to access data transformation logic. In most cases, the transformer code was held in their own classes (even

[1] Deserialization is *not* to be factored out into transformers, as will be explained in Section 15.6.

```java
package de.unioldenburg.ses.components;

import de.unioldenburg.qmig.datamodel.DataModel;
import de.unioldenburg.qmig.datamodel.xml2tgraph.TGWriterImpl;
import de.unioldenburg.qmig.exchange.RootType;

public class QMIGExchange2DataModelTGraphTransformer implements
        transformDataDefinitionQMIGExchange2DataModelTGraph
{
    @Override
    public DataModel transformdatadefinitionQMIGExchange2DataModelTGraph(
            RootType input)
    {
        return new TGWriterImpl().xml2tg(input);
    }

    @Override
    public boolean is_input(RootType input)
    {
        return input != null;
    }
}
```

Figure 15.23: Simple transformer implementation, relying on existing functionality from the Q-MIG tooling.

though they were coupled too closely with other classes, as stated above). But their interfaces often only defined methods that were convenient for one particular use case, but did not expose the "raw" data types to transform to and from. For example, the transformation logic to turn Q-MIG data models from XML into TGraphs was hidden behind convenience methods, taking the input data as *File* objects, or lists of them (to import multiple files at once), as well as further control parameters to trigger additional steps to be taken prior or after the actual data transformation. While these methods represent reasonable use cases of the Metric Calculator as a whole, and simplified the high-level logic close to the user interface, they buried the pure transformer code.

Issues like this are not at all *inherent* in object-oriented code. In fact, these problems arguably correspond to code smells and violate well-known principles and practices of object-oriented design, such as the single responsibility principle [Martin, 2003, p. 95]. However, issues like this are also probably quite common, in general. The contribution of SENSEI here is a stronger enforcement of such principles to facilitate reusability.

Figure 15.23 shows the implementation of the transformer taking data in the Q-MIG exchange file format and turning it into corresponding TGraph representations. As can be seen, the transformer, like all transformers that were created for Q-MIG, is fairly simple, as it is only a very thin wrapper that delegates to pre-existing code to perform the actual transformation. Since all transformers look basically like this, only this one

example is given. The logic behind these wrappers is not considered, either, as it has nothing to with using SENSEI, and would not yield any additional insights.

The transformer's input and output parameters have been bound to the Java types *RootType* and *DataModel*, respectively. The former represents the root of an in-memory representation of an XML file. The data conforms to the model defined in an XML schema, from which Java classes (like *RootType*) have been generated using JAXB. The *DataModel* type, along with further dependent classes, represent a TGraph. These classes have similarly been generated, based on a TGraph schema defined using grUML [Ebert, Riediger, and Winter, 2008]. The actual transformation is delegated to an instance of class *TGWriterImpl*, which is part of the existing Q-MIG tooling, but was modified to provide the *xml2tg* method, using an *extract method* refactoring.

Note that loading input data into memory, or deserializing it into objects should *not* be factored out into transformers. Transformers are special kinds of components, and SENSEI components never interact directly, but exclusively via a composer component. Depending on the target platform, SENSEI processor implementation, and deployment type, input and output data will have to be serialized and deserialized when being transferred from component to composer, and then back to a different component. In a distributed deployment, a transformer might even be residing on a different, physical host than the component that will eventually receive the transformed data. Therefore, a transformer that actually only loads or deserializes data, which it then has to immediately serialize again to return its results to a composer is completely pointless.

## 15.7 Composer Creation

The creation of the coordination logic, insertion of appropriate data transformers, and wiring up inter-component dependencies, is all done (almost) fully automatic in this last step. At this point, components have been registered (Section 15.4), and adapters created (Section 15.5), for all services defined (Section 15.2) and instantiated in orchestrations (Section 15.3). Now, for each of these orchestrations, *composers* can be generated by SCAffolder.

For simple, non-distributed, standalone toolchains, the only manual step required is the provision of user interfaces. A simple command-line interface can be implemented in only a few lines of code – SCAffolder does not yet support generating this, though it should be fairly straight-forward to add such a feature. In Q-MIG, several user interfaces already existed, both command-line interfaces such as that of the Java Metric Calculator, as well as graphical user interfaces, such as the Q-MIG GUI, which provides access to most of the individual Q-MIG tools. The latter does little to automate common processes, leaving it to the user to manually specify data produced by one tool as input of another, and invoke tools in the right order.

For this case study, the original functionality behind existing user interfaces was simply swapped for corresponding functionality provided by generated composers.

```
1 NodeFactory factory = NodeFactory.newInstance();
2 Node composerNode = factory.createNode();
3 composerNode.start();
4 result = composerNode.getService(ExecuteToolchain.class,
  ↪ "Composer/ExecuteToolchain")
5     .execute(consolidatedDataModel, individualDataModels);
```

Figure 15.24: Code to initialize a Tuscany SCA node and invoke the SCA service exposed by the generated *ConsolidateData* composer.

By doing so, some screen masks corresponding to intermediate steps became obsolete. Using Apache Tuscany SCA as standalone SCA runtime, invoking an SCA service exposed by a generated composer only requires a few lines of code, as seen in Figure 15.24. A single node[2] is instantiated and started in Lines 1 through 3. In such a setup, SCAffolder takes care of also generating all the necessary dependency information (through Maven), so that Apache Tuscany will be able to load all components, resolve SCA service references, and wire them up appropriately. In Line 4, the SCA service representing the entry point for a generated composer, which always provide an *ExecuteToolchain* interface, is retrieved, and subsequently invoked. This example is of the composer generated from the *ConsolidateData* orchestration (see Figure 15.10 on page 278). The method's arguments correspond to that orchestration's input parameters: a reference to a data model repository acting as target of the operation, and a collection of individual data models to be integrated and cleaned up.

The implementation and usage of SCAffolder has already been covered in Chapter 14. Since the basic structure of generated composers is generally the same, a single example, based on the data consolidation activity, is used for illustration in Section 15.7.1. This toolchain was setup to run locally in a standalone environment. In Section 15.7.2, the *parse, migrate, measure* activity (specifically the original base metric calculation scenario) is used as an example of a more complex deployment, with individual tools hosted on different platforms and network nodes in a distributed computing environment.

### 15.7.1 Integrated Tool Support for Data Consolidation

The complete SCA composition defining the toolchain corresponding to this activity is shown in Figure 15.25. It contains four composers, generated from the four orchestrations presented in Section 15.3.2, and the *DataImprover* composite component, which provides all the "atomic" data consolidation services introduced in Section 15.2.2. The

[2] Tuscany's node concept is more meaningful in distributed domains, where multiple nodes can exist, each representing an instance of an SCA runtime, that together form the overall SCA domain. For details, see e.g. Laws et al. [2011, pp. 309ff].

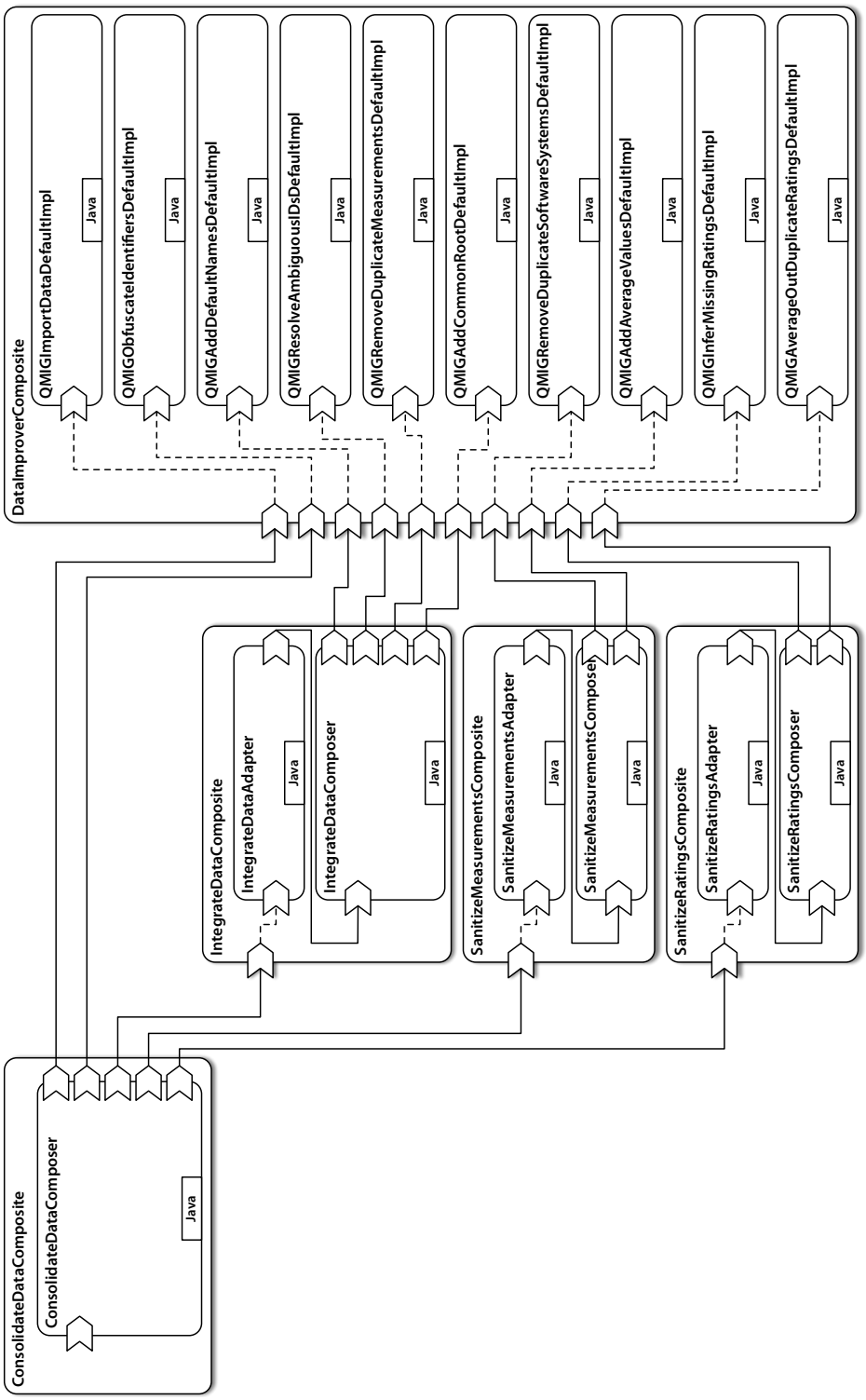



Figure 15.25: Overall SCA composition, consisting of four generated composers, and a single component providing all services.

```
9  @Service({QMIGSanitizeRatingsDefault.class})
10 public class SanitizeRatingsAdapter implements
   ↪ QMIGSanitizeRatingsDefault
11 {
12     @Reference
13     private ExecuteToolchain generatedComposer;
14
15     @Override
16     public de.unioldenburg.qmig.datamodel.DataModel
   ↪ qmigsanitizeratingsDefault(
17             de.unioldenburg.qmig.datamodel.DataModel qmigRepo__2691) {
18         return (DataModel) generatedComposer.execute(qmigRepo__2691);
19     }
20
21     private static final Logger LOG = Logger.getLogger(
22             SanitizeRatingsComposer.class.getName());
23
24     @Override
25     public boolean is_qmigRepo(DataModel qmigRepo) {
26         return qmigRepo != null;
27     }
28 }
```

Figure 15.26: The adapter for the *QMIGSanitizeRatings* component. Only the highlighted lines were added manually.

single SCA component that is normally generated by SCAffolder has been broken up into individual SCA components, each providing only a single service, for greater clarity.

In addition, the three SCA composites on the intermediate level all define both a composer component, and an adapter component. Each composite also corresponds to a separate Maven project. The mid-level projects are *QMIGIntegrateData*, *QMIGSanitizeMeasurements*, and *QMIGSanitizeRatings*, each of which include SCAffolder in their build setup to generate a composer *and* a stub for an adapter. The latter produces a proper SENSEI adapter stub for exposing the service induced by the respective orchestrations. SCAffolder cannot yet provide this fully automatically. Instead, it generates generic SCA service interfaces for all components. To use generated composers as SENSEI components, this small, additional step has to be performed manually for now. For the *SanitizeRatingsAdapter*, shown in Figure 15.26, the code that has to be added to the generated stub for this only takes up four lines (the ones highlighted). Of course, more sophisticated checks of the input (here, only a simple null check is performed in Line 26) may be required in some cases. In addition, a single line also has to be added to the generated SCA composite file to declare and wire the reference to the actual composer.

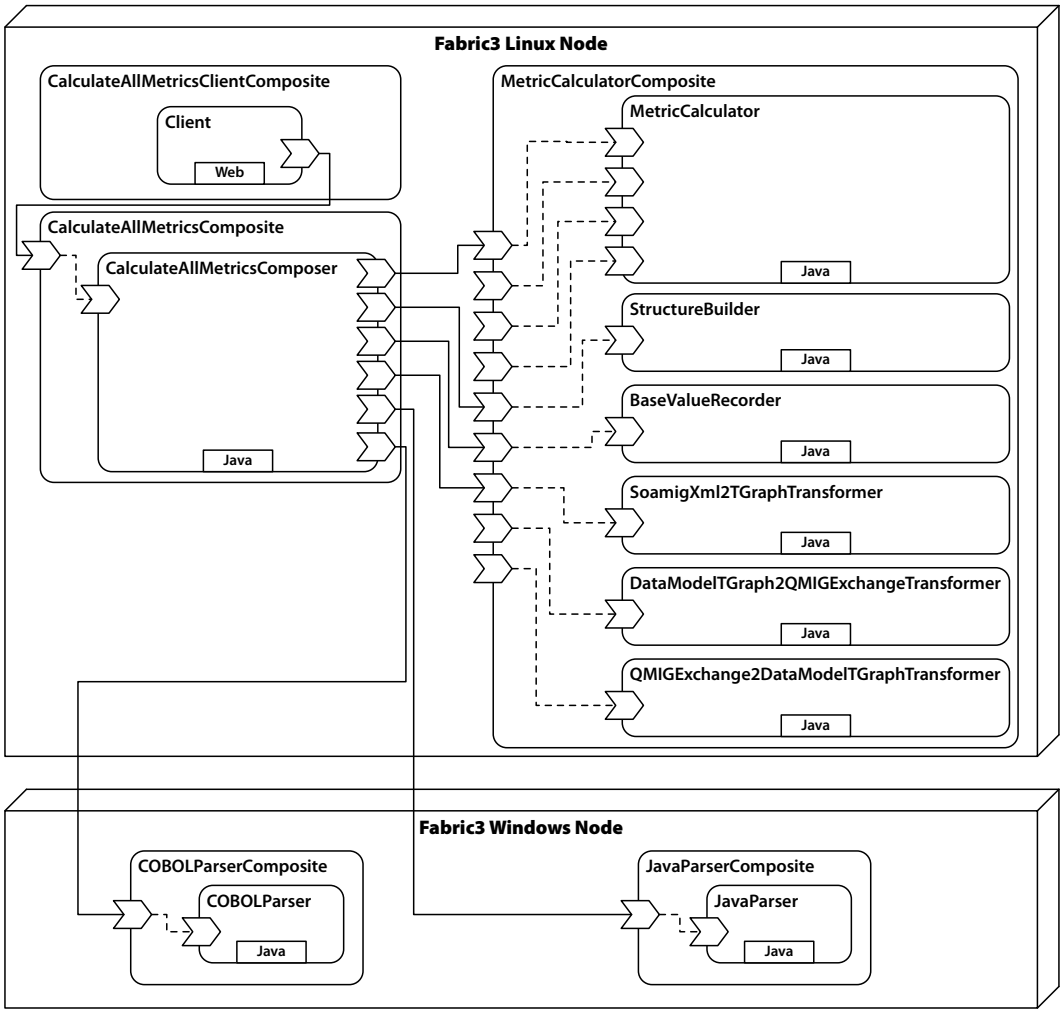



Figure 15.27: Overall SCA composition, distributed over two Fabric3 nodes.

### 15.7.2 Integrated Distributed Cross-Platform Tool Support

As the goal here was to demonstrate toolchain deployment to a cross-plattform, distributed computing environment, the original base metric calculation process was chosen as a suitable, minimal example. The orchestration (see Figure 11.7 on page 189) uses four service instances: *Parse*, *CalculateMetric*, *ExtractStructure*, and *MapResultsToStructure* (defined in Section 10.1.1). The generated composer relies on three components to provide these services: the *COBOL Frontend* and the *Java Frontend* by *pro et con* provide the two different parsing capabilities, while all other services are provided

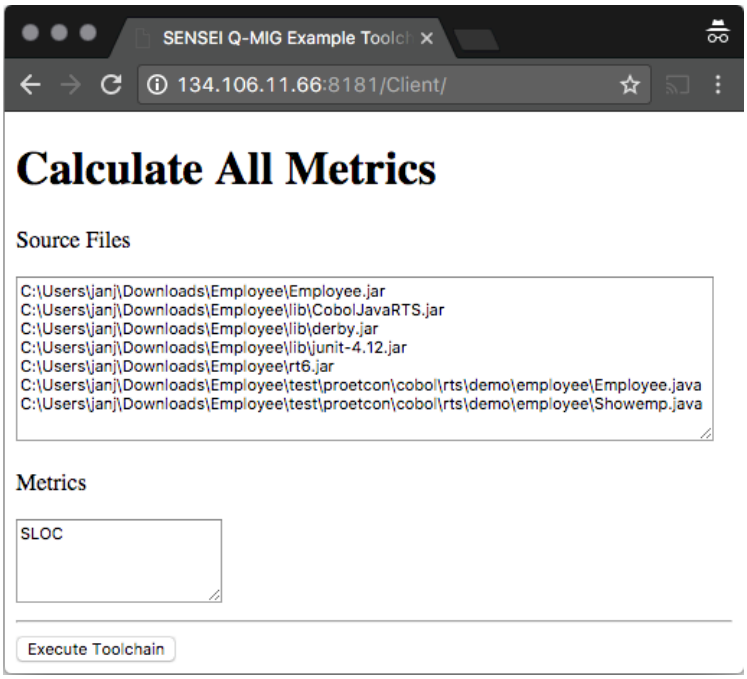



Figure 15.28: A primitive web interface for the base metric calculation toolchain.

by a single component, the Java Metric Calculator[3].

Figure 15.27 shows the resulting SCA composition, corresponding to this orchestration, and the components required to provide the services' functionalities. A very simple web client, depicted in the top-right, has been added manually. As always, the composer component (shown below it), is fully auto-generated. It has six SCA references, which are wired to SCA services provided by *MetricCalculator*, *COBOLParser*, and *JavaParser*.

*MetricCalculatorComposite* provides the most services. SCAffolder generates a single SCA component providing all the services registered in the SENSEI model. The initial, auto-generated boilerplate code was manually altered, and the provided SCA services were split onto six SCA components, as shown in the figure[4].

The *MetricCalculator* SCA component provides four SCA services, corresponding to a single SENSEI service (*CalculateMetric*), and four provided capabilities. The or-

[3]Despite the name, this tool also provides capabilities to calculate some metrics on COBOL code, namely those that are mostly programming language-independent, e.g. *SLOC*. For this particular setting, the required capabilities were reduced to a set that could be handled by the Java Metric Calculator alone, to keep the example simple.

[4]Alternatively, individual components could have been created in the SENSEI registry. SCAffolder would directly generate separate SCA components from such a model, but they would also each be encased in their own composite and Maven project.

chestration from which this composer was generated was simplified to declare only a single required capability on this service (representing the ability to calculate the SLOC metric), which is why only one SCA service is actually wired up.

*StructureBuilder* and *BaseValueRecorder* provide the *ExtractStructure* and *MapResultsToStructure* services, respectively. In addition, there are three transformers: *SoamigXml2TGraphTransformer* outputs TGraph-based representations of Java and COBOL ASTs from XML representations that *pro et con*'s parsers produce (and that was originally defined during the SOAMIG project [Fuhr et al., 2012], hence the name). *DataModelTGraph2QMIGExchangeTransformer* and *QMIGExchange2DataModelTGraphTransformer* convert between a TGraph-based and an XML-based representation of the Q-MIG data model. Due to how the data definitions were chosen for the component adapters, they were not required here (refer back to Section 15.6 for details on the transformers used for the Q-MIG application).

The components were deployed to a distributed environment consisting of two nodes: one machine running *Microsoft Windows Server 2008 R2* hosted the two parsing frontends, whereas the Metric Calculator, as well as the Composer ran on Ubuntu GNU/Linux 16.04. Instead of Apache Tuscany SCA, Fabric3, Version 2.5.3, was used as the target framework, after failing to create a working, distributed setup with the former. Since code generated by SCAffolder only uses standard SCA features, it was possible to deploy all components without having to change anything. For convenience however, SCAffolder was slightly extended to better support different packaging standards and deployment procedures, by having individual Maven profiles generated[5].

For this setup, both machines were on the same network to allow for node discovery via IP multicasting over UDP. Fabric3 uses *JGroups* [2020] for cluster creation and communication between nodes, which can also be configured for use in a wide area network setting, e.g. using TCP and a static list of initial nodes. Since such aspects are well outside of the scope of this thesis, the simpler variant was chosen.

Figure 15.28 shows a screenshot of the web client that was created manually. As can be seen, it allows to enter a list of code files to be parsed, and a list of metrics to be evaluated over the software system represented by said code files. The web form is defined in a few lines of HTML, backed by a similarly simple Java servlet to handle the HTTP POST request that gets sent when a user clicks on the *execute* button. The client is seamlessly integrated into the overall SCA composition using the *web* implementation type (see Marino and Rowley [2009, pp. 311ff] or Laws et al. [2011, pp. 232ff]). This makes the servlet into an SCA component, able to reference the generated composer and invoke the SCA service it exposes to run the toolchain.

[5]Later extensions to SCAffolder, such as more sophisticated SCA namespace handling, have revealed some issues with Fabric3: it failed to resolve namespace imports and exports across node boundaries. Also, the evolution of Fabric3 itself has made SCAffolder-generated artifacts incompatible, as the framework has, as of Version 3.0.0, seemingly broken away from standard conformance, probably due to the failure of the OASIS standardization efforts (see Section 14.2.3)

```
234 if (data.get("source_code__33") instanceof String[] && parsecobol.
235         is_source_code((String[]) data.get("source_code__33")))
236 {
237     LOG.info("[0] InvokeAndReturnImpl (390)");
238     byte[] outAST__32 = parsecobol.parseCOBOL((java.lang.String[])
    ↪  data.
239             get("source_code__33"));
240     LOG.info("result data saved as 'outAST__32'");
241     setResult(data, "outAST__32", outAST__32);
242 }
243 else if (data.get("source_code__33") instanceof String[] &&
    ↪  parsejava.
244         is_source_code((String[]) data.get("source_code__33")))
245 {
246     LOG.info("[1] InvokeAndReturnImpl (394)");
247     byte[] outAST__32 = parsejava.parseJava((java.lang.String[])
    ↪  data.
248             get("source_code__33"));
249     LOG.info("result data saved as 'outAST__32'");
250     setResult(data, "outAST__32", outAST__32);
251 }
```

Figure 15.29: Excerpt from the generated composer code, deciding which parser component to invoke.

The composer, in this example, was running on the same node as the client. After initialization, the first thing it does is checking which parser component to use. The corresponding, generated code is shown in Figure 15.29. It first checks to see if the COBOL parser can handle the input data, by asking it whether it thinks the data represents source code according to its own definition (i.e. COBOL code).

As an example, the input visible in the screenshot of Figure 15.28, representing files of a small Java software system, and the name of a single metric to be calculated, SLOC, was run through the toolchain. To illustrate the sequence of events and interactions between the nodes, fine-grained logging has been incorporated in all tool adapters / components. As shown in the composer code excerpt, SCAffolder also generates logging statements, although the messages are somewhat abstract, owed to the fact that SCAffolder has no knowledge of the concrete significance of its generated coordination logic. Therefore, it reports on the kind of statements that are getting executed, with (arbitrary, but unique) numerical IDs to trace each log message back to the originating line of code.

The log messages output by either node while executing a run through the complete toolchain, using the above-mentioned example input, are shown in Figure 15.30 and Figure 15.31. The former shows the Linux host's log, hosting the web client, the composer, and the metric calculator, while the latter shows the Windows host's log on which the parsers resided.

```
1  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [1] CopyStatementImpl (521)
2  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [2] CopyStatementImpl (522)
3  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [3] CopyStatementImpl (523)
4  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [4] SwitchInvokeAndReturnImpl (532)
5  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [1] InvokeAndReturnImpl (394)
6  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: result data saved as 'outAST__32'
7  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [3] ConcurrentStatementImpl (529)
8  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: starting to execute concurrent
   ↪  block (530).
9  INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [1] ForEachStatementImpl (524)
10 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [1] SwitchInvokeAndReturnImpl (535)
11 INFO de.unioldenburg.ses.components.MetricCalculator.calculatemetricSLOCJavaFile: Calculating
   ↪  SLOC of Java files.
12 INFO de.unioldenburg.ses.components.MetricCalculator.calculateMetric: Calculating metric.
13 INFO de.unioldenburg.ses.components.QMIGAdapter.deserializeAST: Abstract syntax graph (SOAMIG)
   ↪  de-serialization.
14 INFO de.unioldenburg.ses.components.QMIGAdapter.deserializeAST: de-serialization successful.
15 INFO de.unioldenburg.ses.components.QMIGAdapter.registry: Initializing metric registry.
16 INFO de.unioldenburg.qmig.metricregistry.MetricRegistryImpl.loadMetricsFromServiceLoader: Loading
   ↪  metric services
17 INFO de.unioldenburg.ses.components.QMIGAdapter.registry: Registry initialized with the following
   ↪  metrics:
18 INFO de.unioldenburg.ses.components.QMIGAdapter.registry: CloneLines [PROJECT]

[...]

28 INFO de.unioldenburg.ses.components.QMIGAdapter.registry: SLOC [FILE]

[...]

59 INFO de.unioldenburg.ses.components.QMIGAdapter.registry: NumberOperandInstances [METHOD]
60 INFO de.unioldenburg.ses.components.MetricCalculator.calculateMetric: Metric registry initialized
61 INFO de.unioldenburg.ses.components.MetricCalculator.calculateMetric: Retrieved metric from
   ↪  registry
62 INFO de.unioldenburg.ses.components.MetricCalculator.calculateMetric: Evaluated metric on
   ↪  specified system
63 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: result data saved as 'collect_36'
64 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: finished executing concurrent block
   ↪  (530).
65 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: counting down latch (530).
66 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: starting to execute concurrent
   ↪  block (531).
67 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [1] SwitchInvokeAndReturnImpl (534)
68 INFO de.unioldenburg.ses.components.StructureBuilderAdapter.extractstructureNoChoice: Extracting
   ↪  data model structure from abstract syntax graph.
69 INFO de.unioldenburg.ses.components.QMIGAdapter.deserializeAST: Abstract syntax graph (SOAMIG)
   ↪  de-serialization.
70 INFO de.unioldenburg.ses.components.QMIGAdapter.deserializeAST: de-serialization successful.
71 INFO de.unioldenburg.ses.components.QMIGAdapter.serializeTGraph: TGraph serialization.
72 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: result data saved as
   ↪  'structure__47'
73 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: finished executing concurrent block
   ↪  (531).
74 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: counting down latch (531).
75 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [2] SwitchInvokeAndReturnImpl (533)
76 INFO de.unioldenburg.ses.components.BaseValueRecorderAdapter.mapresultstostructureNoChoice:
   ↪  Writing metric values into data model structure.
77 INFO de.unioldenburg.ses.components.QMIGAdapter.deserializeTGraph: TGraph de-serialization.
78 INFO de.unioldenburg.ses.components.QMIGAdapter.serializeTGraph: TGraph serialization.
79 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: result data saved as
   ↪  'mappedResults__51'
80 INFO de.unioldenburg.ses.calculateallmetrics.Composer.execute: [2] CopyStatementImpl (520)
```

Figure 15.30: Log messages emitted by the Linux node while running an example through the toolchain.

```
1  Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.COBOLParser is_source_code
2  INFO: Checking whether the input data is COBOL source code...
3  Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.COBOLParser is_source_code
4  INFO: Unknown file extension: C:\Users\janj\Downloads\Employee\Employee.jar
5  COBOL Frontend cannot handle this input.
6  Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser is_source_code
7  INFO: Invoked is_source_code
8  Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
9  INFO: invoked 'parseJava'
10 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
11 INFO: #source code files: 7
12 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
13 INFO: Processing file C:\Users\janj\Downloads\Employee\Employee.jar
14 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
15 INFO: Recognized file extension: jar
16 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
17 INFO: ... adding to class path.
   [...]
48 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
49 INFO: Processing file
   ↪  C:\Users\janj\Downloads\Employee\test\proetcon\cobol\rts\demo\employee\Showemp.java
50 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
51 INFO: Recognized file extension: java
52 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
53 INFO: ... adding to source files.
54 Mai 22, 2017 7:52:04 PM de.unioldenburg.ses.components.JavaParser parseJava
55 INFO: Parsing...
56 Mai 22, 2017 7:52:13 PM de.unioldenburg.ses.components.JavaParser parseJava
57 INFO: Done (success).
58 Mai 22, 2017 7:52:13 PM de.unioldenburg.ses.components.JavaParser parseJava
59 INFO: Reading result file for byte stream serialization...
60 Mai 22, 2017 7:52:13 PM de.unioldenburg.ses.components.JavaParser parseJava
61 INFO: Success (10 MB).
```

Figure 15.31: Log messages emitted by the Windows node while running an example through the toolchain.

On the Linux machine, Lines 4 through 7 correspond to the code snippet shown in Figure 15.29. All of the log messages shown from the Windows node were emitted as a result of executing these composer lines, which, for this concrete example input data, remotely invokes three methods on the parser components.

First, the COBOL parser's *is_source_code* method is called, which checks the input data and rejects it based on the file names' extensions (Lines 1 through 5). Next, the same method is invoked on the Java parser component – it does not feature detailed logging (Lines 6 and 7), but based on the following steps one can conclude that it deemed the input data valid, since the composer proceeds to call the actual service method *parseJava* (Lines 8 and 9).

The Java parser adapter will then sort through all the input files to separate source files and binaries (Java class and jar files representing dependencies of the software system to be parsed)[6]. This process corresponds to the log messages of Lines 10

[6]The SENSEI service *Parse* only has a single parameter for source code, as it must abstract from technical details and different interfaces of concrete implementations. But the Java Frontend by *pro et con* requires a second input parameter, so the adapter needs to account for this.

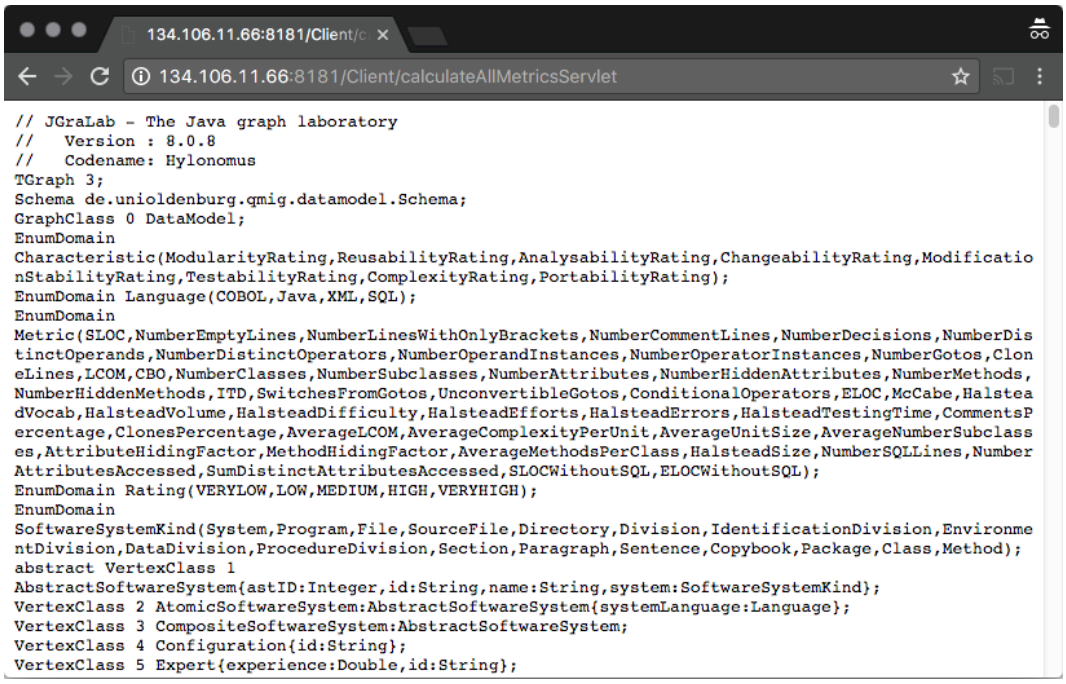



Figure 15.32: Output of the web client after running a simple Java software system through the toolchain.

through 53, most of which have been omitted here, as they are all of the same kind.

Finally, the adapter invokes the actual parser, and then serializes its output for sending it back to the composer across the network (Lines 54 through 61). This returns control back to the composer (Line 6 in Figure 15.30), and also concludes all the communications between the two nodes, as all the components required for subsequent service invocations are deployed on the same node as the composer. The remaining steps can be traced by the log messages emitted on the Linux machine:

After parsing is completed, two concurrent blocks of execution are started (Lines 7 through 65, and Lines 66 through 74 – to avoid confusion stemming from concurrent execution, the log messages have been ordered into two sequential blocks like this for this listing).

The first block loops through the metrics to be calculated (Line 9; here, only the single SLOC metric will be calculated), and invokes the *CalculateMetric* service provided by the metric calculator component (Line 10). Lines 10 through 60 stem from this component's initialization, i.e. loading the provided AST, and all available metric calculation rules. Most lines regarding the initialization of the internal metric registry have been omitted – the metric calculator loads a total of 42 metrics, even though only

a single, specific one is being requested each time. This is a remnant of the original implementation that simply has not been changed for the application of SENSEI to Q-MIG (see Section 15.5.3). The actual retrieval of the requested metric from the registry, and its subsequent evaluation over the provided software system is logged in Lines 61 through 63.

In the second block, the hierarchical decomposition structure of the software system is extracted from the provided AST. This is recorded in Lines 67 through 72. Finally, the calculated metric values are associated to their corresponding nodes in the system hierarchy, e.g. the files, classes, or methods on which the metrics were evaluated (Lines 75 through 78). The composer receives this result, and copies its reference into the result variable to be returned to the invoking client.

The client simply displays the TGraph representation of the Q-MIG data model it receives, shown in Figure 15.32, which contains the hierarchical representation of the analyzed software system, with the calculated metric values embedded. While TGraph files are plain text-based, they are not really meant to be read and comprehended by humans, directly. Instead of having a client operate this toolchain alone, it would be more sensible to pass its results through subsequent analysis and visualization steps, as modeled by the orchestrations presented in Section 15.3. Therefore, this output only serves as proof of a functioning toolchain distributed over a network of nodes using different operating systems.

## 15.8 Results

The main objective of this case study was to demonstrate that SENSEI scales well beyond the toy examples shown earlier throughout the thesis by applying it to a real-world research and software modernization project. In particular, two goals were defined in Section 15.1:

1. Automation of activities and comparison with original tooling.
2. Cross-activity, cross-platform, and cross-network node automation.

Regarding the **first goal**, SENSEI was successfully used to replace the integration logic of the original toolchain, while reusing the individual tools. Some of the original unit tests were run against SENSEI-based toolchains without any regressions, indicating functional equivalency. The original toolchain implementations also served as test oracles, as several Q-MIG activities were repeated using both the original and the SENSEI-based tooling, yielding the same results.

While no exact runtime comparisons were performed, no significant loss of performance was noticed, either. There can certainly be an overhead due to more indirections and the initialization of utilized frameworks (e.g. SCA middleware), but this is dominated, by far, by the execution time of the individual tools for all but the simplest, smallest sets of data, and so is of no real consequence.

The much more decisive factor is the time to develop toolchains. Again, there is an initial overhead for defining services and registering components. Once a core set of services and their implementations is available, the SENSEI-based tooling is arguably easier to modify and extend. This case study produced some quite complex orchestrations, some of which have been shown in the previous sections. They are, however, far more concise, and thus easier to comprehend and modify, than the corresponding integration code of the original toolchain written in Java. As part of the NEMo case study (Chapter 16), further vivid evidence for the flexibility of SENSEI will be presented.

Regarding the **second goal**, *cross-activity* integration is no special feat for SENSEI, but a natural fit for its service integration capacities. It was demonstrated, e.g. by modeling complex orchestrations like the ones presented in Section 15.3.3, and by generating corresponding, fully-functional integrated tooling using SCAffolder. However, applying SENSEI for modeling highly complex processes spanning multiple Q-MIG activities has highlighted the need for decomposing orchestrations hierarchically. SENSEI's metamodel supports this in essentially two different ways:

1. orchestrations *induce services*: their input and output parameters determine service parameters, and their data and control flow defines the intended semantics. The induced services can then be instantiated in other orchestrations (or even within the same orchestrations to model recursive processes). Composers generated by SCAffolder can be registered as components implementing the orchestration-induced services. Deriving capability classes and capabilities from orchestrations (by analyzing *orchestration trails*, see Section 12.3) is not yet supported, and requires further research, but was not needed during this case study, either.
2. orchestrations *are nested directly*. The SENSEI metamodel supports this quite naturally, but so far, the editors only use this to allow modeling structured control flow. It should be possible to extend both the editors to allow direct nesting of orchestrations, as well as SCAffolder, e.g. by having it preprocess SENSEI models and simply flatten orchestrations with nested sub-structures. To an interpreter like SNOrcInS, nested orchestrations would come even more natural.

The latter approach works on the level of orchestrations, only, while the former spans all three basic levels of SENSEI (service catalog, orchestrations, and component registry). This can pose a problem, because changes to an orchestration may induce changes to corresponding services, which in turn may affect other orchestrations, as well as components providing these services.

The tool support for either approach can be further improved, e.g. to alleviate the potential downsides of the first approach, and to provide direct support for the second one. Feasibility in general was shown by this case study, though. It used the first method, only requiring some manual effort to implement proxy components as frontends for generated composers, so they would properly expose the appropriate SENSEI services.

*Cross-platform* and *cross-network* automation has been shown in this case study using fully integrated tool support for the *parse, migrate, measure* activity. During the original run of the Q-MIG project, the lack of automation of this activity presented one of the biggest challenges, and was a source of several issues, that took weeks to sort out. The individual tools were not properly integrated because of a divide in both platform and deployment site: the tools of industry partner *pro et con* were Microsoft Windows-based, while the Software Engineering Group's infrastructure consisted largely of machines running UNIX derivates. Because of confidential input data in form of COBOL programs to be analyzed, the first steps of the analysis, up to the base metric calculation, had to be performed on the partner's site. Integrating the tooling across both platform and network boundaries manually was considered far too complex, particularly given the project's short duration of only fifteen months.

This case study has demonstrated that SENSEI models can be mapped to heterogenous, distributed hard- and software infrastructure *completely transparently*, i.e. without the need to modify the models at all. The actual mapping is done by a SENSEI processor, in this case SCAffolder, which utilizes SCA features to invoke components on different platforms and machines across the network. Of course, such a deployment could just as well have been put in place manually. In fact, the actual deployment of individual tools, as well as launching and connecting individual network nodes, is not directly supported by SENSEI or SCAffolder. The benefits of using SENSEI here are

1. the facilities for distributed, cross-platform toolchain integration have to be realized only once, as opposed to once for every toolchain, and
2. the fact that SENSEI models, in particular the orchestrations, can be used and modified regardless of these concerns, retaining complete flexibility through full platform- and location-transparency.

CHAPTER 16

# The NEMo Mobility Platform

Software Evolution Services and the SENSEI approach have originally been conceived to support the modeling and integration of toolchains in the context of software evolution projects. The application domain is expressed in the *Comprehensiveness* requirement (defined in Chapter 3), which demands universal applicability to support arbitrary software evolution techniques. This has guided the design of SENSEI to be generic.

Due to its generality, SENSEI is actually not restricted to being used in the contexts of toolchain-building and software evolution. At its core, it is an approach for service-oriented modeling of processes, and subsequent, automatic mapping to implementing software components, and generation (or interpretation) of the integrating coordination logic. Therefore, SENSEI can also be used, for example, to model business logic of arbitrary software applications, conveying the same benefits of improved reusability, flexibility, and overall productivity during software development and evolution.

This chapter is aimed at providing evidence for this hypothesis, by describing the results of a successful application of SENSEI for the creation of a "regular" software application, the *NEMo*[1] *Mobility Platform*. The NEMo project is aimed at improving sustainable mobility in rural areas, and will be further described in Section 16.1. Its innovative nature, and a need for extendability and sustainability (longevity) demand a highly flexible software architecture that does not erode in the face of frequent changes. The principles of SENSEI (Definition 9.1, page 161) support exactly that. Section 16.2 describes how the approach can be utilized in NEMo, and gives examples of SENSEI having already been applied in this project, successfully. Section 16.3 provides further evidence of the flexibility gained by using SENSEI, by describing different *flexibility*

[1] NEMo stands for "**N**achhaltige **E**rfüllung von **Mo**bilitätsbedürfnissen im ländlichen Raum", meaning "Sustainable Fulfillment of Mobility Needs in Rural Areas". The project is funded by the Ministry for Science and Culture of Lower Saxony, Germany, and the Volkswagen Foundation (VolkswagenStiftung) through the "Niedersächsisches Vorab" grant programme (grant number VWZN3122).

*scenarios*. A summary is provided in Section 16.4, highlighting and delimitating the application scope of SENSEI in normal software application development, and sketching a concept for transferring the principles of the approach to modeling *interactive*, user-centric application behavior.

## 16.1 The NEMo Project

More than 60% of the total population of Germany live in rural areas [Statistisches Bundesamt, 2015, p. 29], yet medical facilities, shopping centers, cultural and recreational offerings, as well as job, training, and educational opportunities are primarily found in urban centers. This is giving rise to diverse mobility needs, which are not, or not sufficiently, addressed by public transport offers, especially since, in general, local public transport coverage of rural areas has been declining.

The interdisciplinary research project NEMo aims at the sustainable fulfillment of mobility needs in rural areas, and opts for a holistic view, considering social, demographic, accessibility, legal, economic, and ecological conditions and objectives. NEMo wants to facilitate the provision of novel mobility services based, for example, on social self-organization, and develop business models that increase utilization of private transport, while reducing the overall number of vehicles on the streets. Information technology is viewed as key enabler to support these objectives by means of software systems implementing a mobility platform that is accessible through various devices and media (web, mobile, automated hotlines, ticket kiosks, etc.) and addresses communication, announcement, notification, reservation, and compensation requirements regarding mobility needs and services of both providers and consumers. A cross section of this functionality is planned to be realized in prototypical manner as a major outcome of the overall project.

Like any software system, the NEMo mobility platform will need to evolve to keep up with new or modified requirements, e.g. changing legal regulations, or new business models for the provision of mobility services. Unless specific counter-measures are taken, continuously adapting the mobility platform to support unanticipated use cases, for which it was not originally designed for, will lead to an ever more complex and less maintainable software system [Lehman, 1996].

Due to the innovative nature of the NEMo project, a flexible software architecture is of particular importance, to allow for experimentation, and simple and fast integration of new features to test and support research hypotheses and results, respectively. Independent of the scientific context, the application domain of novel rural mobility service provision also stresses the need for highly flexible software support: the NEMo mobility platform should be able to support all kinds of mobility needs and scenarios, modes of transportation, and business models. It should facilitate the recombination of existing mobility services to provide enhanced services, as well as completely new, unanticipated usage scenarios, and the development of corresponding business models.

Besides the technical implementation issues that may arise due to a required software change, the semantic gap between the process-oriented business view of use cases to be realized on the one hand, and mostly object-oriented software architecture views on the other hand, remains a major challenge of software engineering, in general [Combemale et al., 2016; Schmidt, 2006].

Finally, with the overall goal of NEMo being *sustainability*, it is only appropriate to strive for it in terms of software design. A rigid, monolithical software system would lead to high maintenance costs, and ultimately to its phaseout, closedown [Rajlich and Bennett, 2000], and forced replacement. To be sustainable, the NEMo mobility platform must make architectural provisions for flexibility, evolvability, and longevity. This aligns neatly with the objectives of SENSEI (Chapter 1), which led to the decision to try to employ it beyond its original field of application. In NEMo, SENSEI is used to model the business logic of its mobility platform in terms of services and service orchestrations, to achieve the same flexibility, reusability, and productivity gains that the approach confers for software evolution toolchain building.

NEMo builds on the outcomes from previous projects, *ICT Services* and *ICT Platform*, which were carried out in the context of the *Electric Mobility Showcase* program [*ICT Services* 2016; Wagner vom Berg, 2015]. The ICT infrastructure created in this context serves as the basis for the NEMo project. A major use case, that is expected to play an important role in NEMo is *inter-modal route planning*. In the following, this use case is presented as an example of the kind of ICT support that is needed in NEMo (Section 16.1.1). The way this has been realized in the preceding projects is then contrasted with the aspired functional extensions, and particularly the new architectural foundations to facilitate them, to be realized in NEMo (Section 16.1.2).

### 16.1.1 Inter-Modal Routing

The NEMo mobility platform is expected to serve both service providers and consumers, i.e. businesses or people seeking to either offer or make use of mobility services, which is transporting people or goods from one place to another. One exemplary use case for the platform is to find routes: given a point of origin and a destination, it should provide possible routes. Combining different modes of transport, e.g. walking, riding a bike, take a bus or a train, driving a private car, or joining a car pool, makes this *inter-modal routing*.

The activity diagram depicted in Figure 16.1 shows a possible breakdown of this use case into process steps. A client wishing to travel inter-modally sets off the process by requesting a route to his destination. The mobility platform first tries to find reasonable stopovers to possibly change the mode of transport. Then, available mobility services, providing transportation by different traveling means, are searched for to instantiate the individual sub-routes. The results are integrated into complete traveling itineraries, and returned to the user, who chooses from them. One special case is considered: if the

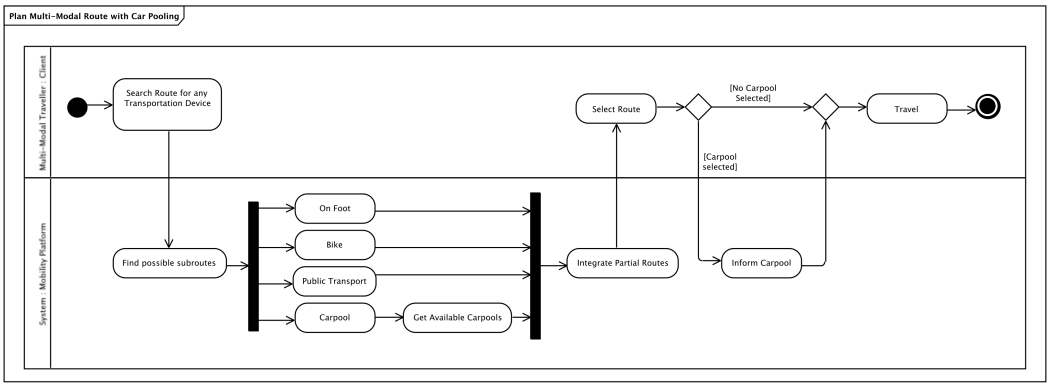



Figure 16.1: Exemplary mobility business process to get inter-modal route directions.

selected route utilizes a car pool, its driver – as provider of a mobility service another user of the mobility platform – must be informed. Of course, this is an extremely simplified and incomplete example, but it conveys the general idea.

### 16.1.2 Challenges

The existing infrastructure and functionality from the ICT Platform and ICT Services projects is already able to support the use case of inter-modal routing to a large extent. The ICT Platform project developed an eponymous software system providing an infrastructure to host software components based on *WSO2* [2020], and a market place to showcase and acquire the software services they provide. On top of this platform, the ICT Services project build another software system to combine its basic software services, and offer value-added services in support of designated business processes.

Due to the different focus of these projects (electric mobility), it is not tailored specifically towards rural mobility needs. Other improvements are being considered, e.g. providing inter-modal routes that allow joining a car pool at a convenient point along its planned route, instead of having to travel to and from its points of origin and destination, respectively, as it is currently implemented.

In addition, the current infrastructure has not been designed with a particular focus on flexibility and evolvability. The business processes underlying use cases, such as for inter-modal routing, are “hard-wired” into the ICT Services’ software system, and the system is therefore hard to adapt and extend. This is not a good fit for NEMo, given its sustainability objective, and the need for the mobility platform to sustain, facilitate, and mirror the project’s progress, innovative nature, and evolving requirements. In summary, the goals for the NEMo mobility platform are therefore as follows:

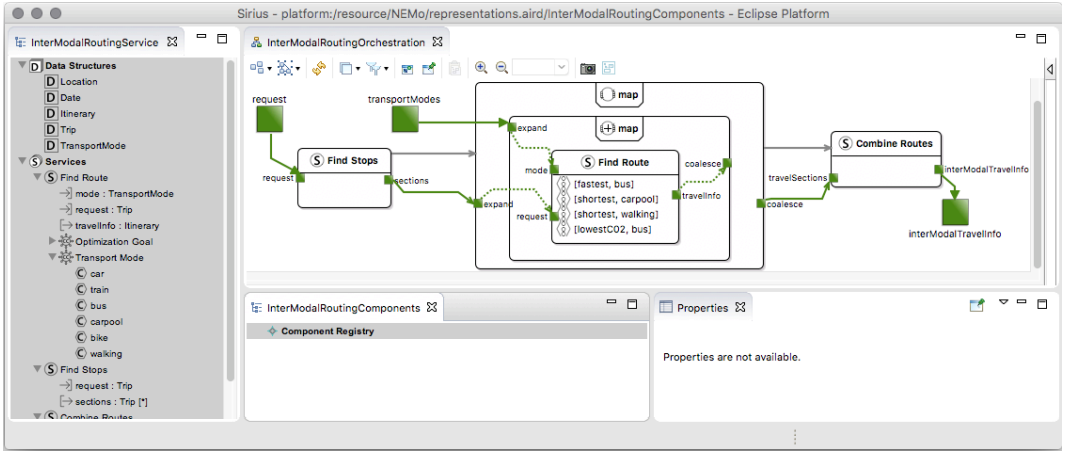



Figure 16.2: Inter-modal routing service catalog and orchestration in the SENSEI editor.

1. Enhance, extend, and modify the existing functionality based on the research findings within the NEMo project, to support, for example, community-driven, self-organized mobility, novel business models, and legal constraints.
2. Facilitate these, and future changes, by providing an architectural framework that incorporates the existing functionality, but highlights flexibility, adaptability, and long-term sustainability.

While the exact nature of the first challenge will only be known once the project has progressed further, the solution proposed to tackle the second challenge is adopting SENSEI to build the mobility platform.

## 16.2 Application

Using SENSEI for modeling and integrating general software applications encompasses basically the same steps that also make up the toolchain-building process (Section 3.1). *Tool integration* now becomes *application integration*, but other than this purely terminological change, everything stays the same.

Figure 16.2 shows a screenshot of the SENSEI editor, with a model of the inter-modal routing example opened. The steps of the corresponding business process (Figure 16.1) are mapped to three services: *Find Stops* decomposes a routing request into requests for sub-routes, *Find Routes* provides traveling information (e.g. itineraries or driving instructions) for a single route, and *Combine Routes* fuses these together into integrated informations for the whole trip. For the *Find Route* service, capabilities are used to model a variability in terms of supported transport modes, with capabilities being *car*,

*train*, *bus*, etc. Another capability class represents supported optimization goals (shortest or fastest route, cheapest connection, lowest $CO_2$ footprint, etc.).

For NEMo, the comprehensive, heavy-weight middleware WSO2 had been designated from the outset as the target infrastructure to build the mobility platform on, since this had been used for the *Electric Mobility Showcase* projects. SCAffolder could have been modified or extended to support WSO2 in addition to SCA, or the WSO2 application server could have been configured to host an SCA runtime environment. Instead, it was decided to develop a new SENSEI processor from scratch, to take this opportunity and investigate an interpreter approach to contrast with SCAffolder's generator approach. An outcome of these efforts is the SENSEI interpreter SNOrcInS, which was realized by Küpker [2015]. SNOrcInS has been briefly described in Section 14.5.

Küpker also applied SENSEI with the newly-created interpreter to different variants of a scenario from the NEMo project: for the basic variant, a simple route planning process for using public busses was modeled. The corresponding orchestration is depicted in Figure 16.3. Using the desired origin and destination locations as input, first, the bus stations closest to either one are retrieved (*NearestStationFinder*). Then, three partial routes are calculated (*RouteFinder*):

1. walking directions from the origin to the bus station closest to it,
2. bus connections from that station to the one closest to the destination, and
3. walking directions from there to the final destination.

To complete the process, these three routes are combined into a single, inter-modal route (*RouteConcatenator*).

This simple bus planning orchestration was then slightly modified to optimize its results in terms of travel time [Küpker, 2015, pp. 59ff]: the first variant of the orchestration might produce less than optimal solutions in this regard, because the nearest stations might not offer the best connections. Additional walking time incurred by having to get to or from stations further away from the origin or destination location may well be offset by far better bus connections. A revised orchestration, taking this observation into account, is shown in Figure 16.4. Instances of *NearestStationsFinder* (notice the extra "s") are now used to retrieve not just the single closest bus station to a particular location, but a set containing a number of stations (that number being determined by input port *count*). Using two nested *map* constructs, routes are calculated for all combinations of bus stations. All resulting routes are passed into an instance of a new service, *RouteSelector*, which simply chooses a single element based on a metric determined by a capability: here, the *TIME* capability requires an implementation to select the element with the shortest overall travel time.

Modifying and extending the original scenario to get to the optimized one took about half a day, according to Küpker [2015]. This included the specification of the new service, RouteSelector, the implementation of a corresponding component, as well as the modification of NearestStationFinder into NearestStation**s**Finder, and the adaptation of the implementing component. The latter could also have been designed

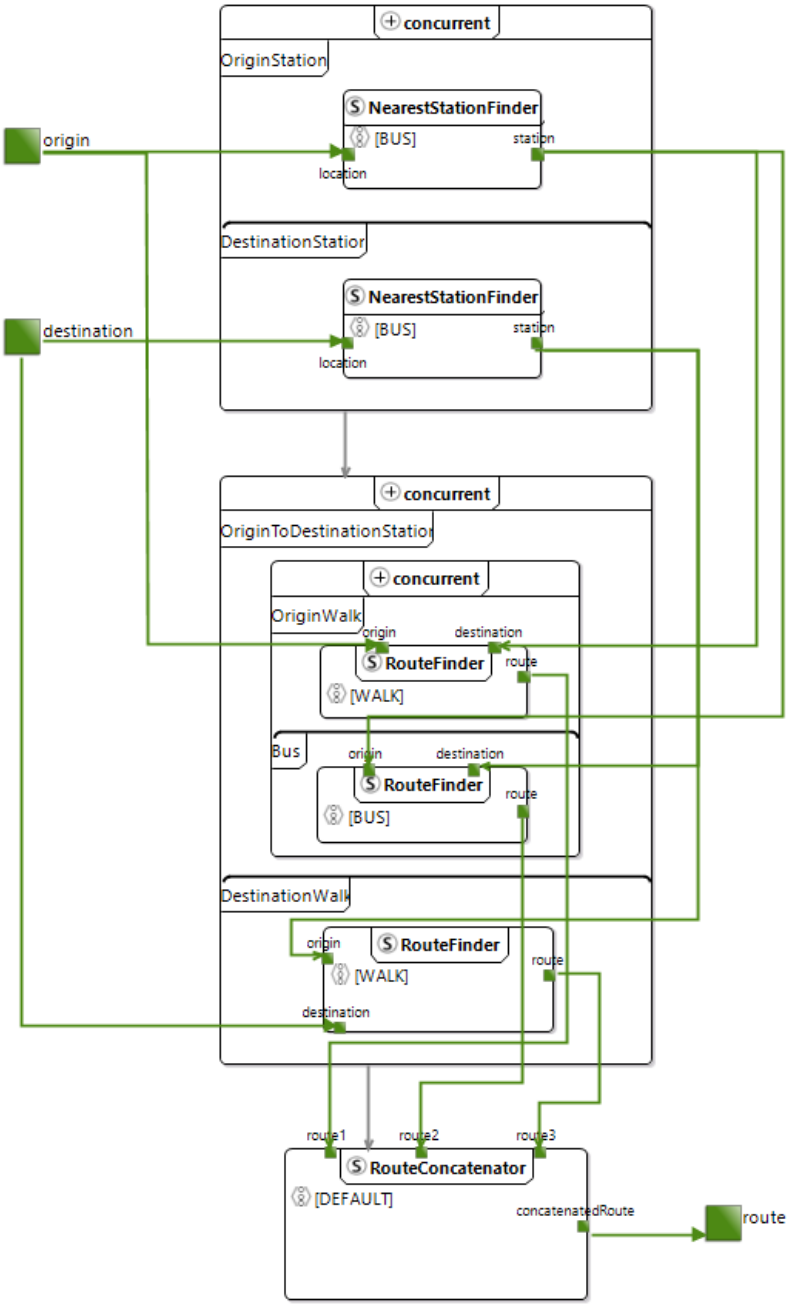



Figure 16.3: Bus planning orchestration, taken from Küpker [2015, p. 72].

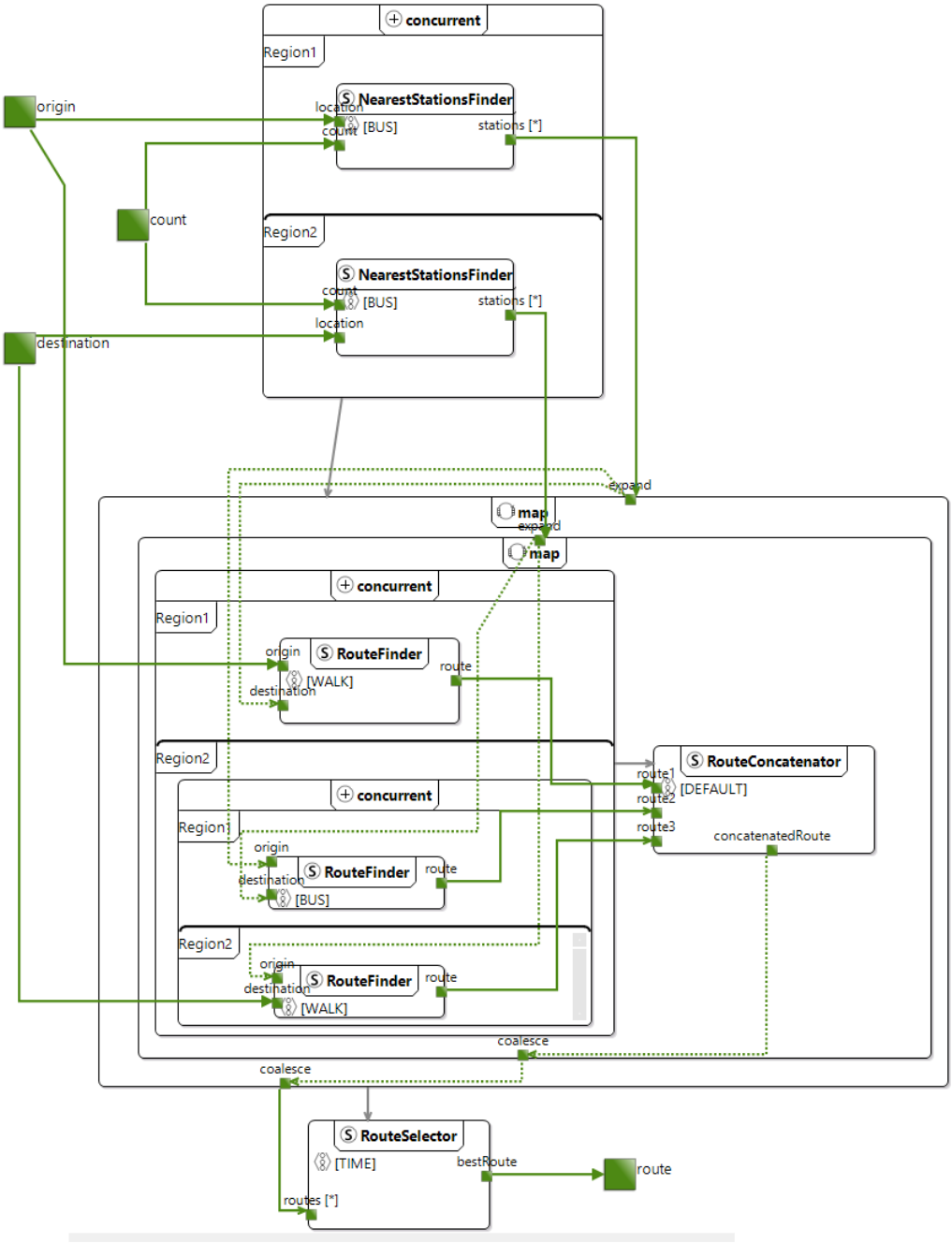



Figure 16.4: Optimized bus planning orchestration, taken from Küpker [2015, p. 73].

that way from the start, as setting the input port *count* to 1 yields the behavior of the simpler variant. This was intentionally *not* done, to avoid the impression of an overly contrived example.

Still, with a little foresight and properly generic services, this modification effort can be avoided. The same would be true if there was already a reasonably comprehensive service catalog available, either due to up-front, top-down specification, or because SENSEI has been used for some time, resulting in a catalog filled from the bottom up, with incrementally refined services (see Section 9.4 for a discussion of top-down and bottom-up service discovery and description). Subtracting these overheads, which in SENSEI should only occur once due to reusability, changes like this can be realized in under an hour.

In total, only five services had to be defined for these (admittedly simple) examples. An overview is given in Figure 16.5, essentially showing the complete service catalog for this application of SENSEI. It should be pointed out that the *RouteConcatenator* is modeled somewhat awkwardly, being limited to combining exactly three partial routes. This was done in order to keep the examples simple. A slightly more elegant alternative would be to use two subsequent instances of a service appending on route to another.

Besides defining the required services and modeling the orchestrations, implementing components, using Google's Places and Directions APIs [*Google Maps Directions API* 2020; *Google Places API* 2020] and targeting the WSO2 platform, have also been created. Using SNOrcInS, the bus planner orchestrations have successfully been used to calculate various routes for the city of Oldenburg, also confirming that the optimized variant often provides routes with lower overall travel times.

## 16.3 Flexibility Scenarios

The central aim of using SENSEI in the NEMo project is to gain flexibility, to easily integrate software support for new and innovative mobility services into the mobility platform. The bus planner example provided in Section 16.2 already gave an impression of how SENSEI can support evolving a software system quickly and easily, without compromising the overall architecture (which is dictated by SENSEI, and, in this case, SNOrcInS). In the following, further examples from the NEMo application of SENSEI are given, to evince the flexibility qualities of the approach.

Figure 16.6 shows a very simple orchestration that will serve as base case for the following scenarios. It uses only a single service, *FindRoute*, to realize basic route planning functionality. The service catalog defines *FindRoute* to require two inputs, the desired *mode of transportation*, and the *trip request*, which encodes starting point, destination, time of departure or arrival, and potentially further traveling constraints. The service's output contains the *traveling information*, e.g. a list of driving directions. The orchestration nests the service instance within a *map* control flow construct. As

| | | |
|---|---|---|
| **Service** | Name | NearestStationsFinder |
| | Description | Finds the nearest stations for a transportation mode as specified by capabilities |
| | Input | location : Location<br>count : Integer |
| | Output | stations [*] : Station |
| | Capability Classes | TransportationMode = {BUS} |
| **Service** | Name | RouteConcatenator |
| | Description | Concatenates 3 routes to one route |
| | Input | route1 : Itinerary<br>route2 : Itinerary<br>route3 : Itinerary |
| | Output | concatenatedRoute : Itinerary |
| | Capability Classes | Default = {DEFAULT} |
| **Service** | Name | NearestStationFinder |
| | Description | Finds the station with the minimal distance to a location |
| | Input | location : Location |
| | Output | station : Station |
| | Capability Classes | TransportMode = {BUS} |
| **Service** | Name | RouteFinder |
| | Description | Finds a route between two locations using a specific transport mode |
| | Input | origin : Location<br>destination : Location |
| | Output | route : Itinerary |
| | Capability Classes | TransportMode = {BUS, WALK} |
| **Service** | Name | RouteSelector |
| | Description | |
| | Input | routes [*] : Itinerary |
| | Output | bestRoute : Itinerary |
| | Capability Classes | QualityCriteria = {TIME} |

Figure 16.5: Services modeled by Küpker [2015] for use in NEMo.

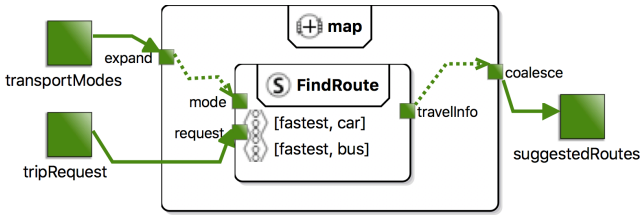



Figure 16.6: Simplistic SENSEI service orchestration for basic route planning.

a result, this orchestration calculates a route for each of the specified transport mode, and returns them as collection of *suggested routes*.

*FindRoute* has two *capability classes*: an *optimization goal* (to find the route that is *fastest*, *shortest*, *cheapest*, etc.), and a *transport mode* (*car*, *bus*, *walking*, etc.). As shown in Figure 16.6, the *FindRoute* instance used in the orchestration is required to support finding *fastest* routes for private *cars* and public *busses*.
Three basic flexibility mechanics can be identified in SENSEI:

1. Adding or modifying required capabilities.
2. Extending or modifying existing orchestrations, or re-orchestrating existing services.
3. Mapping orchestrated services to different components, or partitioning functionality in components in different ways.

For each of these mechanics, an example scenario is given in the following to provide details of their respective workings.

### 16.3.1 Adding Capabilities

The NEMo project aims at supporting mobility in rural areas in a sustainable, environmental-friendly manner. Mobility services advertised on the mobility platform may include "classic" public transport offers (busses, trains, etc.), more recent ones, like car pooling, car sharing, electric bike rental stations, as well as completely new mobility services based on innovative business models or being completely community-driven and non-profit. All of them would benefit if the platform's (inter-modal) route planning functionality would also provide *shortest* routes, instead of *fastest* routes (as modeled in Figure 16.6), assuming the former are generally more ecological than the latter due to lower carbon emissions.

Figure 16.7 shows an orchestration that has been extended to support such a use case. The only difference to Figure 16.6 are additional required capabilities, which demand that for private cars, *shortest* routes will also be provided. The *shortest* capability can, of course, be combined with capabilities for other transport modes in the same manner.

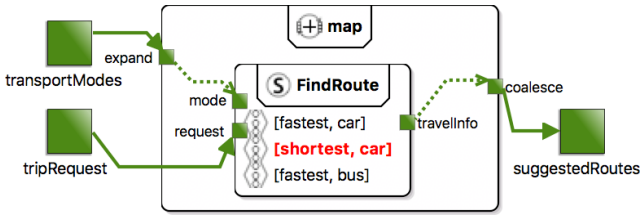



Figure 16.7: Basic route planning with an additional required capability.

In the optimal case, this minimal change is all that is required to add this functionality. SENSEI will automatically integrate additional components as needed. In simple cases, as with this example, it is not unlikely that the originally used implementation might also already support this additional functionality. Recall that SENSEI does not depend on a single component to provide all the required capabilities, but is able to map a single service instance to multiple components (see Section 11.2 and Chapter 12).

In the same manner, support for additional transport modes can be added. What kind of features can be added or modified like this depends on how services are modeled, i.e. what capability classes they define. Of course, this flexibility mechanism assumes that the actual functionality is already available, implemented in registered components, so that SENSEI will be able to integrate them into the modified software solution. Even if such components are missing and have to be implemented, another benefit of using SENSEI is that the standardization provided through the service catalog facilitates reuse: Once a service is implemented in a component, it becomes available for future usage in different contexts. In the long run, the set of components will grow, reducing the need for manual implementation and the effort involved. This makes this scenario more viable over time.

### 16.3.2 Extending Orchestrations

An important consideration in NEMo is the combination of multiple mobility services to provide comprehensive, needs-based mobility options. The basic route planning example is insufficient for this: instead, the ability to provide *inter-modal routing* is needed, meaning the combination of multiple modes of transportation within a single route [Jelschen et al., 2016].

A straight-forward way to evolve the SENSEI-integrated software system that provides the route planning functionality is to extend the existing orchestration with additional service instances. Figure 16.8 shows an orchestration for *inter-model route planning*. It corresponds to the orchestration shown in the SENSEI editor screenshot in Figure 16.2, with those elements highlighted that were added to the basic orchestration shown in Figure 16.6. At its core, it still contains basic route planning, but has two additional service instances, and uses another *map* control flow construct.

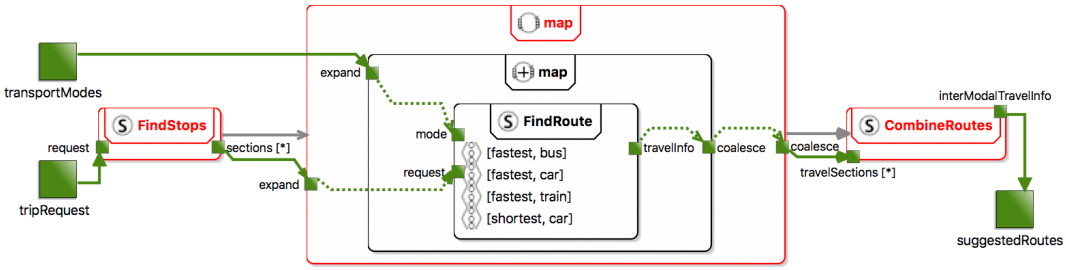



Figure 16.8: Inter-modal route planning orchestration.

The first step is now performed by an instance of the *FindStops* service, which takes the trip request and tries to partition it by determining reasonable places to switch transport mode, e.g. bus stops and train stations. This results in a set of sub-requests, with the discovered stops as new points of origin or destinations. The outer *map* construct iterates these, so that the *FindRoute* instance will now be invoked once for each pairing of sub-request and transport mode, yielding multiple, partial routes. The last step is to stitch these partial routes together again, so that they satisfy the original travel request, which is done by *CombineRoutes*.

In the best case, service definitions and corresponding components implementing them already exist. If not, using SENSEI incurs an initial overhead from having to explicitly model services, and adapt components to the framework. As with the first flexibility mechanism, this is thought to be compensated for during long-term use, due to a growing number of services and components, and hence reusability.

The high abstraction level of orchestrations simplifies the task of defining and evolving the desired processes. Orchestration designers do not have to address different interface or binding technologies, disparate data formats, and technical incompatibilities between components of different vendors or providers. SENSEI shifts the burden of providing interface adapter and data transformation logic from software integrators to component developers, and imposes a structure that promotes its reusability, as opposed to fusing it to individual components. The fact that integration logic is either auto-generated and thus “discardable”, or provided at runtime by an interpreter, prevents SENSEI-based software systems from becoming entangled in hard-wired dependencies.

### 16.3.3 Changing Component Mappings

To realize the functionality modeled in orchestrations, SENSEI maps the instantiated services to components providing them. There is an *n*-to-*m* relationship between services and components: a single service can be realized by a combination of multiple components implementing different capabilities, and the opposite is also possible, with a single component providing multiple, different services.

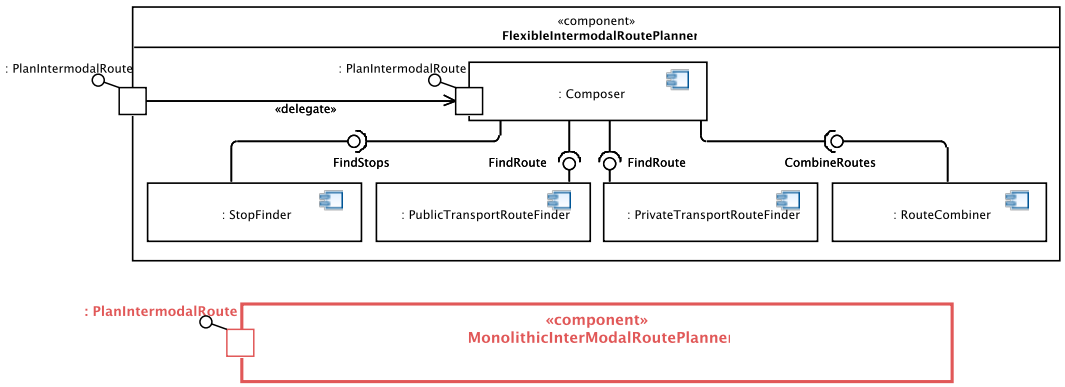



Figure 16.9: Composition of combined components, and a monolithic component, both implementing the *PlanIntermodalRoute* service.

SENSEI makes no assumptions in this regard, i.e. invoking services on components always assumes that they have no knowledge of the overall process they are contributing to, and that they cannot and will not communicate directly with other components. All information goes through a central hub, the *composer*, as can be seen in the top half of Figure 16.9. This component diagram shows a component composition that realizes inter-modal route planning, as specified by the orchestration in Figure 16.8. The *FindStops* service is mapped to the component *StopFinder*, and CombineRoutes is mapped to *RouteCombiner*. *FindRoute* is associated to two components: *PublicTransportRouteFinder* provides the capability for bus and train routes, while *PrivateTransportRouteFinder* provides routes for driving a private car and walking.

The whole composition of composer and service-providing components provides the *PlanIntermodalRoute* service, which is also induced by the corresponding orchestration. This service can be instantiated in other orchestrations, forming a hierarchy of services and orchestrations.

This architecture of completely isolating individual components is essential to delivering SENSEI's stated goal of a sustained, high level of flexibility and reusability. Allowing direct communication between components and inter-dependencies would quickly undermine this objective. However, these benefits are traded in for a potential curtailment of other attributes, e.g. run-time performance. While a smart composer, and the overall SENSEI infrastructure, could certainly try to put optimizations in place, for example to reduce the data traffic through caching, in general, there will be an overhead.

It is also a matter of finding the right level of granularity for services: more fine-grained services may be more reusable, but more coarse-grained ones may allow for more internal cohesion, and optimizations that would otherwise incur a steep performance penalty because of the higher communication overhead.

Because orchestrations are also service instances (*PlanIntermodalRoute* in the example), they can also be mapped as a whole to a single, monolithic component, as suggested in the bottom half of Figure 16.9. Such a component could, for example, realize a more complex route finding algorithm that requires its constituents to be more closely attuned to each other and share large amounts of data. For instance, it could perform multiple passes to incrementally find the best inter-modal route, which might not scale well using a loosely coupled architecture, because of a large communication overhead.

Like the individual service-providing components above, such a monolithic component is viewed by SENSEI as a black box, only having to adhere to the interface defined by the service, while the exact manner of implementation is left unconstrained. Specifically, a monolithic component does not have to follow the process prescribed by an orchestration. With this mechanism, SENSEI allows to trade performance and scalability against flexibility and reusability. The latter only has to be sacrificed for select parts of an overall software system, for which performance is critical. Also, such monolithic implementations fit into SENSEI seamlessly, being treated like any other component, just providing a more coarse-grained service.

## 16.4 Results

This chapter summarized the results of applying SENSEI outside of its originally intended application domain, namely for modeling and integrating the business logic of the NEMo mobility platform. As of this writing, the NEMo project, and the efforts to build a fully-fledged, highly flexible and sustainable mobility platform using SENSEI, are still ongoing. Progress so far has confirmed general feasibility of the approach, and first experiments indicate that it will be able to confer the desired benefits of increased flexibility, reusability, and productivity, for information system development and evolution, just as it does in the field of building and evolving software evolution toolchains. Flexibility in particular was demonstrated, using examples from NEMo to describe three distinct degrees of flexibility.

It should not go unmentioned, though, that SENSEI's scope applies only to the business logic layer of software systems, i.e. the middle layer in a classic three-tier architecture, which can be reasonably described in terms of processes. Above it, there is the presentation layer, which provides the interface to users, and is interactive and event-driven by nature. While services that provide user interfaces can be defined in SENSEI, this is awkward at best, as the approach provides no concepts to model user interaction. Below the business logic layer is the data layer, concerned with persistent state. Access to this layer is encapsulated within components, and thus remains hidden from SENSEI, as well. While no conceptual issues have arisen during the NEMo project as of yet, persistence inevitably introduces statefulness, which needs to be carefully considered in the context of SENSEI's services that usually assume statelessness.

To conclude this chapter, Section 16.4.1 presents some technical observations made during the NEMo experiments, that may guide further improvements of the overall approach. Section 16.4.2 very briefly sketches ongoing research activities to transfer the principles of SENSEI to the presentation layer, and derive a complementary solution that integrates with SENSEI at the boundary of these two layers, and confers similar benefits for modeling flexible, integrated user interfaces from reusable parts.

### 16.4.1 Technical Observations

Within the context of NEMo, the interpreter solution provided by SNOrcInS has proven itself viable, as its dynamic nature provides even more flexibility. This comes in handy when relying on heavy-weight middleware like WSO2, where regenerating, recompiling, and redeploying can be quite disruptive, and slow down development cycles, whereas the interpreter accepts new or changed orchestrations at runtime, with the effects becoming visible immediately. Conversely, it was found that implementing data transmission from and to components can be more challenging in an interpreter. A generator can make composers created from orchestrations aware of the kind of data that needs to be passed around. An interpreter must be designed to be completely agnostic of the types of data, only acting as "dumb router" between the components, because only they will be able to understand the data being passed in and out.

This issue became more pronounced when SNOrcInS was extended for the NEMo project to be completely location-transparent, which necessitated a lot of data serialization and deserialization for network transmission. While certainly not an unsolvable problem, common approaches using high-level frameworks, e.g. for REST-style service communication, might not be the best choice. The interpreter might also be easier to realize in a dynamically-typed programming language – SNOrcInS is written in Java, which is a statically, strongly-typed language.

Another observation made during the experiments with SENSEI in the context of NEMo is that aggregating individual data items, or unwinding collections of data items, can be awkward outside of *map*-constructs. Reasons for this include the simplistic overwrite-semantics of data flows and ports, and the fact that data flows do not trigger behavior (only control flow does), as well as the inherent statelessness of services. While there is no inherent limitation in terms of expressive power, orchestrations may require a large number of service instances to model such data-handling behavior, convoluting the orchestration.

Since it is not a fundamental limitation, it could be addressed by "syntactic sugaring" of the orchestration language, e.g. having the ability to annotate ports to express the intent to aggregate (rather than overwrite) data there, thus yielding a significantly more concise syntax. This can either be done by extending the metamodel, or by only extending the tooling (SENSEI editor). In either case, such a solution has the advantage that orchestrations can be "de-sugared", i.e. refactoring into a canonical form that only

uses the basic language constructs. This way, existing SENSEI processors do not have to be able to understand the extended syntax, and will not have to be modified.

In general, for self-contained software applications, the data integration challenges play a lesser role than for software evolution toolchains, assuming most of the business logic is developed from scratch. While such information systems usually use a lot of third-party software libraries and middleware frameworks, these mostly provide basic infrastructure, or support cross-cutting concerns. In these cases, the business logic is designed with a common data model from the ground up. In inter-application integration, or when relying more heavily on pre-built components providing the required services, these aspects become relevant, again, and SENSEI can adapt to either case through its transformer concept.

On relying on off-the-shelf components, SENSEI can be used to factor out common, generic functionality, by defining correspondingly abstract services and utilizing capabilities to make them configurable for particular applications. This maximizes reuse and reduces the amount of business logic that is actually application-specific and has to be purpose-built from scratch.

### 16.4.2 Interaction Modeling

On the conceptual level, a distinct observation is that SENSEI cannot sensibly be used to model *interactive* application, or at least not those parts that are interactive, i.e. user interfaces. This is not so much a limitation as it is out of the scope of the approach, by definition, since SENSEI was simply not designed to provide a solution for this. In the domain of software evolution toolchains, particularly those meant to automate complex software migration or reengineering processes, interactivity is not a major concern. However, there are more interactive activities in software evolution, too, e.g. for program comprehension or incremental refactorings that require constant user intervention.

These considerations have led to first research activities into designing and developing an approach complementary to SENSEI, founded its core principles (Section 9.5), but aimed at modeling and integrating interactive applications. The core idea is to replace the process-oriented orchestration language in SENSEI with an interaction modeling language based on state machines, which are suitable to describe systems that (only) react with a change of state to outside stimuli such as user-triggered events.

Using state machines to describe interaction models for user interfaces is not a new idea, at all. More recently, the IFML standard was published, describing the *Interaction Flow Modeling Language* in terms of a UML profile [*IFML: The Interaction Flow Modeling Language* 2017]. However, to the best knowledge of this author, no approach exists so far that uses such a state machine-based language in conjunction with the strict separation of specification and implementation that is a defining quality of SENSEI, or has concepts similar to SENSEI's service catalog, component registry, and

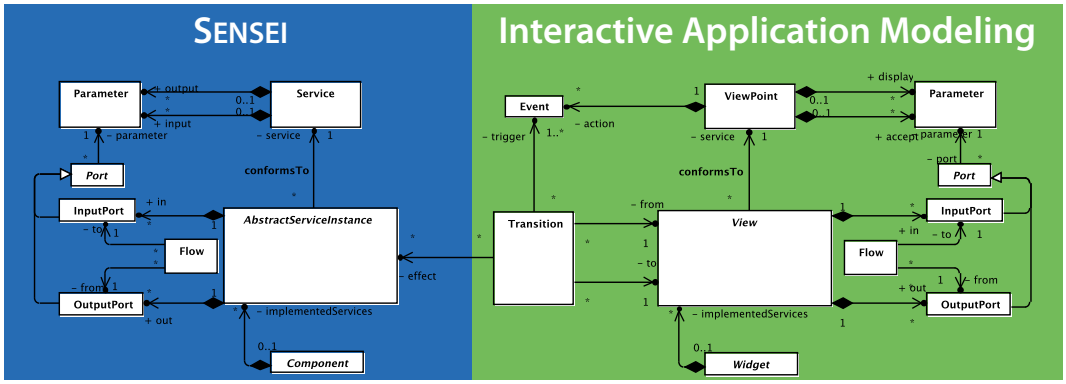



Figure 16.10: Core concepts of the SENSEI metamodel shown side-by-side with a possible, corresponding metamodel for interactive application modeling.

capabilities for automatic matchmaking. An important first step has been made with a feasibility study performed by Schlömer [2017]. Bischopink et al. [2018] have build on these foundations to create the interactive application modeling framework *DORI*.

Figure 16.10 shows the central concepts from the SENSEI metamodel on the left. On the right, they are paired with analogous concepts of what could become the metamodel of a framework to support modeling and integrating interactive applications. The core concepts all have "mirror" images:

- Specification in SENSEI is done in terms of services, representing functional units. On the same layer, interactive applications have *viewpoints*, which define what can be *displayed*, what inputs can be *accepted*, and what *actions* users can take.
- In place of service instances, interactive applications have *views* to represent concrete instances of viewpoints that can be used in interaction models.
- Instead of components, there are now *widgets*. While the former implement the functionality defined by a service, the latter provide concrete user interface implementations corresponding to a viewpoint, e.g. a web form, a part of a command line interface, or a window or dialog box of a desktop application.

Data is handled pretty much the same on both sides, although *parameters* play different, in a way, *reversed* roles for viewpoints than they do for services: inputs to viewpoints are displayed, or *output*, to users. Outputs of viewpoints are what users have *input* into form fields, for example. Otherwise, data flows between concrete views work much the same as between service instances in SENSEI.

The major difference is, however, that interaction models do not have control flow (which exists in SENSEI, but was omitted from the metamodel excerpt shown in Figure 16.10), but instead feature *transitions*. In an interaction model, the views take on

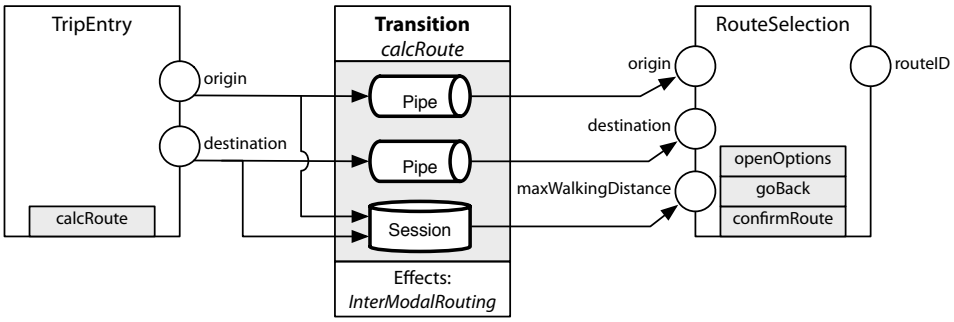


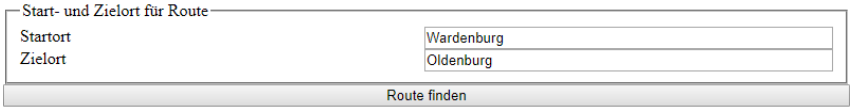

Figure 16.11: An excerpt of a small interaction model; adapted from [Schlömer, 2017, pp. 62, 65].

the roles of states in a state machine. A change of state is *triggered* by the occurrence of *events*, and governed by transitions. Therefore, in addition to parameters, viewpoints also define what *actions* users can take. If users take such an action, the associated event may trigger transitions from the current view to another. And, most importantly, transitions may have *effects*, meaning they may invoke application behaviors of the underlying business logic, and potentially display any data produced on the next view. These effects that transitions may have provide the link between interaction models and SENSEI orchestrations, which may be used to model those application behaviors.

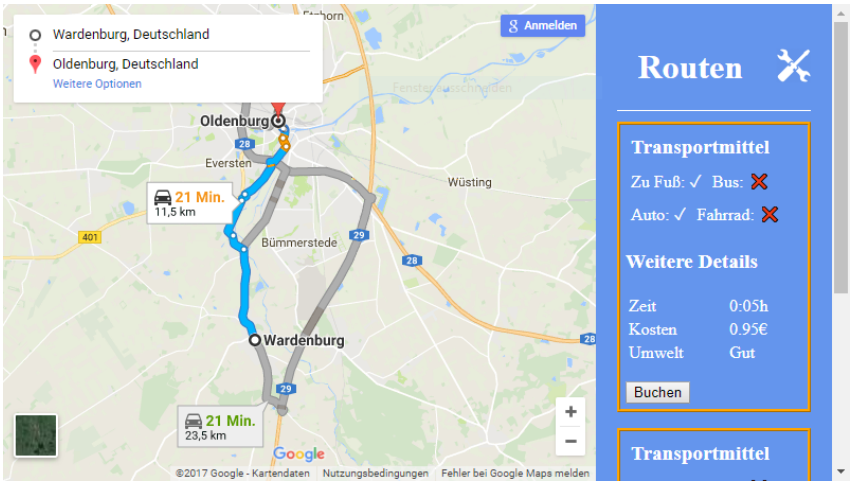
For the sake of simplicity, *guards*, which allow a transition to execute only if a certain condition holds, have been omitted here. Also, on a more general note, whether and how capabilities fit into this whole new picture has not yet been looked into, but will be the subject of further research in the context of the NEMo project.

The terminology for these concepts is still somewhat in flux, e.g. Schlömer uses the term *widget* for what was here described as *viewpoint* (analogous to services in SENSEI), while the widgets introduced here are referred to as *WidgetImpl* (analogous to components in SENSEI). The terms as they are introduced here have been revised to avoid such awkward names. Some inspiration has been taken from the IFML standard [*IFML: The Interaction Flow Modeling Language* 2017] and a user interface taxonomy introduced by Chignell [1990], though there might still be room for improvement. The challenge here is to try to stick to terminology already established in the field of user interface design, as long as the terms' common definitions are appropriate, while also avoiding terms that are similar or identical to those already used in SENSEI, unless they truly describe the exact same concept.

Schlömer [2017] has designed an initial metamodel for this SENSEI-like approach for modeling interactive applications, and implemented a prototype of an interaction model interpreter. He then applied the approach to model and integrate a user interface, once again using NEMo as application example. A small part of the corresponding interaction model is depicted in Figure 16.11: two views and a transition between them




(a) Screenshot of a widget implementing *TripEntry*; taken from [Schlömer, 2017, p. 63].




(b) Screenshot of a widget implementing *RouteSelection*; taken from [Schlömer, 2017, p. 64].

Figure 16.12: Example widgets corresponding to the interaction model of Figure 16.11.

are shown. The graphical modeling syntax shown here has been come up with by Schlömer for presentation purposes (meanwhile, Bischopink et al. [2018] have developed a graphical editor with a revised modeling language for their DORI framework). Views are depicted as white boxes, ports are shown as circles on their borders, and the events users can generate by performing actions (like clicking a button) appear at the bottom. Transitions appear as boxes with the names of the events that trigger them in italics at the top. In this design, transitions encapsulate data flows (called *pipes*), and there is a *session* concept to store and retrieve data that might not be needed in a direct successor view, but may appear on a different view, should the user later decide to navigate there.

The views contained in this example allow users to enter the origin and destination of a planned trip (*TripEntry*), and, after confirming the entry (generating the *calcRoute* event), to select one of the displayed routes (*confirmRoute*), make changes to the previously made entries (*goBack*; corresponding reverse transition not shown), or set additional parameters like the maximum distance one is willing to walk (*openOptions*). Widgets corresponding to these views might look like the ones shown in Figure 16.12(a) and Figure 16.12(b), respectively. These have been implemented as web-based user interfaces by Schlömer as part of his feasibility study.

Building on these foundations, a one-year student project group [Bischopink et al., 2018] has developed a more complete metamodel (e.g. considering nested or concurrent views), a correspondingly revised, graphical interaction model language, an interaction model editor, and fully-featured model-driven tooling to execute interactive applications. These results are used in NEMo, together with SENSEI, to build its mobility platform [Kuryazov, Winter, and Sandau, 2019].

The ultimate goal of these and follow-up research activities is to complement SENSEI in such a way that a comprehensive framework is provided, spanning all layers of three-tier systems, to support building, integrating, and evolving highly flexible software applications from reusable parts.

CHAPTER 17

# Achievement of Objectives

In the introduction to Part V, four evaluation goals have been defined, repeated here for convenience:

- **Feasibility:** Can the approach be technically realized?
- **Applicability:** Can the approach be practically applied in the intended domain, and on an industrial scale, while conferring its desired benefits?
- **Generalizability:** Can the approach be practically applied beyond its intended domain, to confer its desired benefits in a more general scope?
- **Utility:** Does the approach confer the desired benefits expressed by its objectives of *flexibility*, *reusability*, and *productivity*?

The case studies presented in Chapter 15 and Chapter 16 were designed to answer the associated questions. Attainment of the first three goals will be discussed briefly in the following. The *utility* goal is aimed at assessing whether the three key objectives of this thesis are satisfied by SENSEI, and to what extent: increasing *flexibility*, *reusability*, and *productivity*. Due to their significance for the overall thesis, each objective is dedicated a separate sub-section (Section 17.1, Section 17.2, an Section 17.3, respectively).

**Feasibility** had already largely been demonstrated in Part IV, through the fully-realized SENSEI metamodel, and by implementing the approaches' meta-tools SCAffolder, SNOrcInS, and the Composition Finder, as well as the SENSEI editor. Both the base metric calculation toy example used throughout this thesis, as well as the self-application of SENSEI to SCAffolder, showed the feasibility of actually using the SENSEI method and its associated tools in the intended manner. The detailed descriptions of applying SENSEI in the Q-MIG and NEMo projects further substantiate this.

**Applicability** refers to SENSEI's relevance for real-world use cases, and its ability to scale to real projects. Both case studies have illustrated this, though the Q-MIG application was more specifically aimed at demonstrating industry scalability. To this end,

Chapter 15 explicitly details going through the full toolchain-building process with SENSEI support. It presents several complex processes that were successfully modeled in terms of SENSEI services and orchestrations, described the adapters and transformers that had to be implemented, and showed fully-functional, integrated toolchains generated by SCAffolder.

**Generalizability** expands on applicability by requiring an application domain that is outside the originally targeted field of software evolution. Also, since the Q-MIG project was a major motivation for this thesis, it could be argued that evaluating the approach only against that same use case poses a threat to validity in that it leaves the possibility of SENSEI working only for this one project for some reason. The NEMo application eliminates that case, by using SENSEI to model business logic of its mobility platform, essentially demonstrating that it can effectively be used for application integration, in general. The self-application to SCAffolder is another example of this.

**Utility** goes further, as it requires a demonstration not only of whether SENSEI can in fact be realized and used, but also of whether doing so confers the advantages this thesis set out to achieve: increased *flexibility*, *reusability*, and *productivity*.

## 17.1 Flexibility

In the context of this thesis, *flexibility* of integrated solutions produced by SENSEI refers to effort required to perform changes to them – the less effort is required, the more flexible they are considered. This concept of flexibility is borrowed from Eden and Mens [2006], who try to quantify flexibility in terms of *evolution complexity* given in big O notation. This idea will be used in the following to analyze the flexibility scenarios presented as part of the NEMo case study (Section 16.3), and compare the evolution complexity of a change to a SENSEI-based integration solution to a manually integrated one. It must be noted that idealized assumptions are made for the SENSEI case, i.e. all necessary services, components, adapters, and transformers are already available for a given change in an orchestration. The implications of these assumptions will be discussed afterwards.

The first scenario presented was **adding capabilities** (Section 16.3.1, page 317). This is a very simple change in SENSEI, and under the assumptions made, is in $\mathcal{O}(n)$, where $n$ is the number of capability tuples to be added. The concept of capabilities is not present when not using SENSEI, but it corresponds to adding support for another "feature" to the overall toolchain. In the best case, one of the tools already integrated supports the new feature. It will potentially require changes to the integration logic, so the new feature can be used, and will be, given appropriate input data. In most cases, this means that the change is arguably easier when using SENSEI, because it automatically generates the integration code. The evolution complexity class is the same though.

However, in many cases requiring a new capability tuple will also require a new tool supporting it to be integrated (the number of tools backing a single service instance

is bound by its declared capability tuples, so every capability tuple potentially introduces a separate tool). This requires much more additional integration logic, but more importantly, it also requires additional adapter and transformers (which may not be neatly separated as they would be when using SENSEI). This is critical, because it leads to a significantly worse evolution complexity of $\mathcal{O}(m \cdot n)$, where $m$ is the number of tools in the toolchain the new tool needs to be integrated with. Viewed through the corresponding SENSEI orchestration, the number of tools would be found by tracing the data flows connected to the service instance with the added capability tuple. The sum of all capability tuples declared for the service instances found this way provides the upper bound the number of tools the new tool will need to be integrated with. This means that very concise-looking SENSEI orchestrations with only a few service instances, and each with a few capability tuples declared, translates to a potentially massive integration effort.

The next flexibility scenario is **extending orchestrations** (Section 16.3.2). It introduces two changes, adding service instances to an orchestration, and adding control flow structures. The latter requires to adapt the integration logic – a manual programming task when not relying on SENSEI, which is arguably more laborious than making the corresponding changes to the orchestration. But, there is no fundamental difference in terms of the evolution complexity, which is $\mathcal{O}(n)$, with $n$ being the number of control flow structures.

Similar to the first flexibility scenario, adding service instances is once again more complex without SENSEI, because they have to be integrated with $m$ other service instances. While SENSEI automates this, the glue code has to be manually rewritten, which leads to an evolution complexity of $\mathcal{O}(m \cdot n)$.

The last flexibility scenario is **changing component mappings** (Section 16.3.3). This corresponds to taking an integrated tool out of a toolchain and putting another one in its place. With SENSEI this is straight-forward, because the adapters of its components are standardized by corresponding catalog services. Without the approach, adapters, transformers, and integrative glue code cannot, in general, be assumed to be cleanly separated, so the whole integration logic has to be adapted or recreated. Once again, the evolution complexity is $\mathcal{O}(m \cdot n)$, as the required effort depends on the number of tools $m$ that connect to those tools being swapped out.

SENSEI achieves its flexibility by means of abstraction, enforcing clean separation of concerns, and automation enabled by these definite structures. Manual integration has, in general, lots of accidental complexity, which SENSEI eliminates. In theory, very disciplined manual integration can achieve the same, but that would require to set up the same or similar rules as SENSEI does, which essentially amounts to an ad-hoc reinvention of the approach, and adhering to its principles without any tool support. In any real-world software project, this would be virtually impossible to sustain.

It must also be noted that there are limitations to the flexibility of SENSEI. For once, flexibility increases are potentially traded with a decrease in toolchain runtime perfor-

mance – this can be controlled, though, as discussed in Section 16.3.3. Also, considering changes to service catalogs reveals high evolution complexity: changing service definitions is only easy as long as there are no orchestrations using them, and no components implementing them. Otherwise, the change will require to adapt all affected orchestrations and components, as well, leading to prohibitively high effort and correspondingly low flexibility in this regard. However, SENSEI services should almost never have to change: many changes are due to the evolution of underlying implementation technologies, and do not influence the functional level. Functional change requests can be addressed by adding capabilities to existing capability classes, or adding completely new service definitions, which incurs no increased costs. A remaining issue would be cluttering of the catalog in use cases which sees services evolve frequently, leading to many similar services. One simple way to address this would be a versioning mechanism for services, which SENSEI currently does not include.

To conclude the discussion, the initial assumptions must be considered, as they seem to have shifted the odds unfairly in favor of SENSEI. If the assumptions are lifted, all scenarios must account for having to define missing services, register needed components, and implement necessary adapters and transformers. Having to amend service catalogs and component registries is a real overhead imposed by SENSEI, but it is a fairly low one compared to the effort required to implement adapters, transformers, and integration logic, and thus should be easily offset by more substantial productivity gains conferred by SENSEI, such as never having to manually write or adapt integration logic.

Having to create adapters to make existing tools usable within SENSEI, or implement additional transformers to make tools interoperable, is much more expensive. However, this is not a downside of SENSEI, at all, as the same adapter or transformer logic would have to be implemented without the approach, as well. The difference in that case would be that such adapters and transformers would not be inherently reusable: SENSEI may, in the worst case incur the same integration costs, i.e. devolve into the same evolution complexity class, as manual integration, but it is never worse. Because SENSEI ensures reusability, these costs can be expected to diminish over time, as will be explored in the following section.

## 17.2 Reusability

Manual tool integration is a repetitive task, especially for continuously evolving toolchains. As discussed in the previous section, adding a tool, or swapping one out for another, can impact large portions of the integration code, so it will need to be adapted frequently, and some parts will repeatedly have to be scrapped and replaced completely. To avoid this, this thesis aimed at increasing the reusability of the integrated tools and all the interconnecting glue code.

As with flexibility, the underlying principles allowing SENSEI to achieve this reusability include clean separation of specification and implementation, and establishing clear

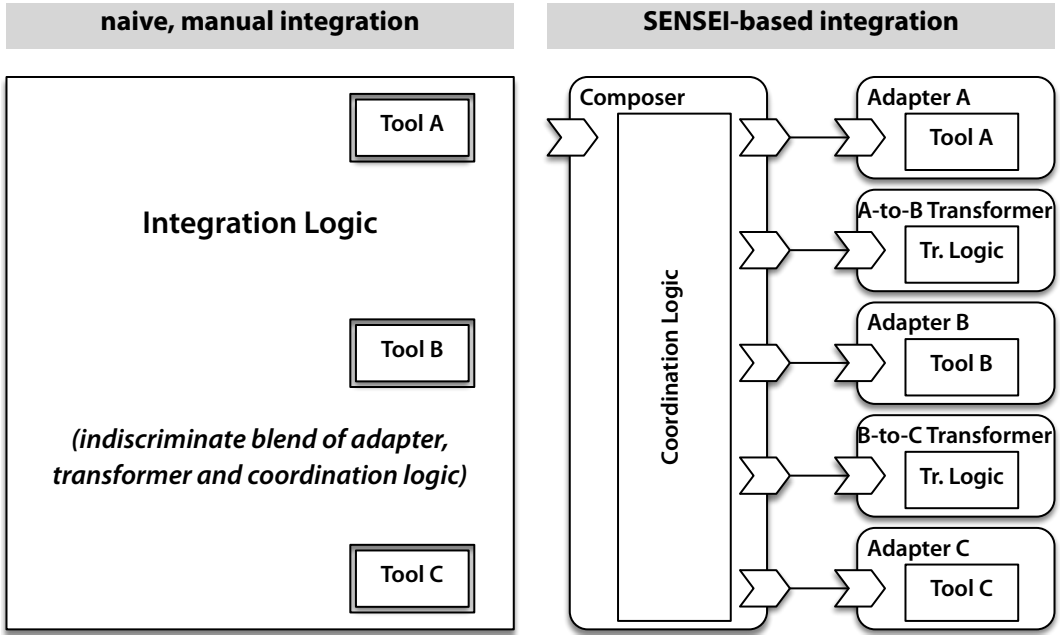



Figure 17.1: Comparison of manual integration with SENSEI.

boundaries traversed only across well-defined interfaces. As a basis for discussion, Figure 17.1 shows an abstract side-by-side depiction of what a manual integration of three arbitrary tools looks like compared to a corresponding SENSEI-generated toolchain.

The manual integration picture is a worst-case scenario, in which the whole glue code bleeds together with adapters and transformers, and no obvious cutting lines can be made out. If the integrated solution was static, this would be an unfair representation, but since this thesis aims at supporting evolving toolchains, any structures that might have been established at the outset are, without a framework keeping them up, likely to erode away in accordance with Lehman's laws [Lehman, 1980].

Toolchains generated using SENSEI enforce clear boundaries: the service catalog standardizes interfaces, which a tool like SCAffolder consistently maps onto a particular target framework. All manually written code must be encapsulated in components and is hidden behind such interfaces, so nothing can bleed through, ensuring loose coupling. The high-level abstractions provided by the SENSEI metamodel and accessed through its editor shield users from technical aspects. This, and the advantage of code generation gained from adhering to SENSEI, strongly incentivize against forcibly breaking the established structures by manual "hacks". More importantly, since such structural violations in the manual integration case are usually done unintentional, SENSEI makes the boundaries clearly visible.

Through these measures, adapters and transformers in SENSEI are fully reusable. Regarding adapters, another important difference may be noted: in Figure 17.1, there is one *more* adapter for the SENSEI integration than for the manual integration, because in the latter case, there is not necessarily a central hub corresponding to a composer in SENSEI. Instead, adapters connect tools to each other directly. This requires one adapter per connection, instead of one adapter per tool. While for simple, sequential toolchains this is marginally better – $n - 1$ adapters against $n$ adapters in SENSEI – for arbitrarily interconnected toolchains, the manual case requires $n^2$ adapters in the worst case, whereas SENSEI remains at $n$. This is not as significant as it seems, as the worst case can be assumed to be rather unrealistic. The real advantage of SENSEI are *reusable* adapters: when a tool is added or swapped out, the manual integration will most likely require changes to its adapter; it may even have to be replaced entirely. Similarly, when a tool that was previously integrated in a toolchain is to be used in another toolchain, a new adapter has to be created. In contrast, SENSEI adapters must only be created once, and can always be fully reused.

Transformers are different than adapters, as SENSEI requires just as many as are needed in a manual integration. But since they are treated as regular components, they never get blended in with the coordination logic, and are inherently reusable, as well. As the number of registered transformers increases with the time of using SENSEI, the more likely it becomes that changes to an evolving toolchain can benefit from reusing existing ones, and it will be integrated automatically and transparent to the user.

What remains is the coordination logic that glues toolchains together. Isolating adapters and transformers from it makes this remaining code fully determined by toolchain specifications, i.e. SENSEI service orchestrations. Changes that require programming in manually integrated toolchains are instead made much easier on this higher level, only. The coordination logic is generated from this. Instead of reusable, it is fully *disposable*, i.e. it can be recreated at will with minimal effort, so no need for reuse ever arises.

## 17.3 Productivity

Overall, a toolchain-building framework is required to increase productivity, i.e. to reach the same results using less effort. SENSEI achieves this in several manners: First, the increased flexibility of the tool support conferred by SENSEI also boosts productivity. More flexible toolchains require less effort to be adapted, and as a result, development cycles become shorter, and projects become more agile. Second, increased reusability saves the effort of having to recreate parts of the integration logic on each change. And third, those parts of the integration logic that are not reusable, are generated fully automatically by SENSEI, reducing the corresponding manual effort considerably, by allowing domain experts to focus on the essential complexities of their processes, modeled

as orchestrations, and eliminating the accidental complexities caused by the underlying technology.

These productivity gains refer to creating and evolving toolchains more efficiently. When using SENSEI to integrate the business logic of a software application that is the actual product being developed within the overall project, such as was the case in the NEMo case study, this is the predominant advantage. If SENSEI is instead applied to create integrated toolchains that support the project, but are not among its main outcome, like in the Q-MIG case study, further advantages can be identified: not only do the improvements free resources that can be spent on the actual project objectives, supported by the SENSEI-integrated toolchains – a more efficient toolchain-building process also means that processes can be automated that would otherwise have been performed manually due to prohibitively high integration costs. In Q-MIG, the greatest loss of productivity is thought to have been due to the measurement process having a manual gap, which resulted in numerous miscommunications regarding how the process should have been performed, as well as mistakes being made while manually performing it.

## 17.4 Summary

Supported by implementations of SENSEI's meta-tools (recall Section 9.1), including the SENSEI editor, SCAffolder, SNOrcInS, and a composition finder, the application of the approach in the context of the Q-MIG and NEMo projects has demonstrated comprehensively the achievement of the objectives of this thesis. This chapter has provided further discussion of the increased *flexibility*, *reusability*, and *productivity* conferred by SENSEI. The reason why the approach is able to provide these benefits can be attributed to its underlying core principles, described concisely back in Section 9.5. These principles represent a significant contribution of this thesis beyond its original objectives, as they can be, and already have been (see Section 16.4.2), used as guidance to derive frameworks constructed analogous to SENSEI that confer similar advantages in areas other than toolchain integration.

CHAPTER 18

# Conclusion

Software integration has become an increasingly important topic for software engineering. In the field of software evolution, the integration challenge is manifested as a need to create automated, flexible toolchains from the large body of existing, but non-interoperable tools. In general, software systems are being modularized to manage their increasing size and complexity, necessitating their integration. Decomposing monolithic systems into individual building blocks is also motivated by a constant need to evolve them, to account for changing requirements and fast-paced technological advancements. At the same time, new trends like the *internet of things* and *industry 4.0* imply highly heterogenous, distributed software-intensive systems of unprecedented scale.

To address the need for integrating software systems from reusable parts in a flexible manner, this thesis has presented the SENSEI approach. Providing a framework and method for the creation of toolchains, and more generally, integrated software solutions implementing specified processes, it has been build on comprehensively researched foundations, and has been evaluated by extensive practical application.

Concluding this thesis, this chapter summarizes its major contributions (Section 18.1), describes limitations (Section 18.2), and provides an outlook towards future research and applications (Section 18.3).

## 18.1 Contributions

The central contribution of this thesis is SENSEI, a **conceptual framework and method** for building *flexible*, integrated toolchains from *reusable* tools. The approach separates concerns and identifies corresponding roles and responsibilities, to yield an *integration method* that provides the toolchain-building process with a clear, perspicuous structure.

The conceptual framework of SENSEI is formalized in terms of its *layered integrated metamodel*. It is the basis for the model-driven aspects of SENSEI, facilitating the description, storage, and exchange of SENSEI, and their processing by conforming tools.

From the metamodel, implementations of SENSEI can be derived. Defined in MOF, the metamodel is compatible with most tools on the market like model-driven code generators, model transformation languages, and language workbenches. This enables the creation of further research prototypes, as well as product-mature SENSEI tool suites viable for commercial use. As authoritative basis, the metamodel also ensures that, on a fundamental level, these tools will all be compatible with each other.

In addition, a dedicated *orchestration language* has been created, incorporating SENSEI concepts such as capabilities, service instantiation, and the strict separation of data and control flow, which existing languages cannot provide.

Another distinguishing feature of SENSEI that provides several unique advantages of the approach is its *capability model*: capabilities aid service modeling and selection, simplify orchestrations and component registration by providing a concise, declarative means for specification, and enable to automatically map services to implementing components, and generating potentially complex integration logic.

The method followed in this thesis has put an emphasis on laying strong foundations for the design of SENSEI to ensure upfront that it addresses the most relevant issues within the overall challenge of toolchain integration, and utilizes the most appropriate means to achieve its objectives. To this end, this thesis has introduced and subsequently analyzed the *toolchain-building process*, and performed a comprehensive literature review of the field of tool integration in general, as well as concrete, existing integration approaches. These works are considered a contribution in their own right, representing a differentiated view of tool integration that may help to place, compare, and assess different integration approaches, and guide future developments and reveal further research opportunities.

Similarly, the integrated view of component-based, service-oriented, and model-driven software engineering paradigms not only lays the groundwork for SENSEI, but also provides a consistent terminology, demarcates the different application areas, and may serve as a comparison framework for existing approaches, as well as as a blueprint to derive new ones. While SENSEI relies on widely accepted doctrines of both the component-based and the model-driven paradigms, the service concept adopted by this thesis has a more precise meaning than is generally found in the literature (relying more on ideas that predate SOA), and plays a very well-defined role in the overall framework. The complementary and synergistic nature of these three paradigms shows in their combination in SENSEI, which further elaborates the identified, basic concepts, and adds the capability model.

In fact, the most basic tenets underlying the presented approach have been condensed into the *core principles of SENSEI* (Definition 9.1). These can serve as guidelines for creating similar frameworks for the development and integration of software systems

that impart the same benefits as SENSEI does, i.e. more flexible solutions, assembled from parts that are easier to reuse, with reduced overall effort. The approach for interactive application modeling described in Section 16.4.2 was derived in this manner from these core principles of SENSEI.

The feasibility of the SENSEI approach has been demonstrated through concrete implementations of an editor, a composition finder, and two distinct tools, SCAffolder and SNOrcInS, to generate toolchains from SENSEI specifications and to interpret them at runtime, respectively. This already demonstrates the technology independence and overall versatility of the approach.

Besides providing a platform to jump-start SENSEI adoption efforts, these tools have enabled the application of SENSEI in two very different fields: to specify and integrate a software evolution toolchain in the Q-MIG project, and to model the business processes of the NEMo mobility platform, and automatically integrate corresponding components, as well. The former study has focused on comprehensively demonstrating soundness and scalability of the SENSEI method, while the latter showed generality and achievement of the intended improvement, particularly flexibility. Both applications utilize different tools and have different target platforms, proving its technology independence. As a further display of broad applicability, a third use case was provided by modeling the process of coordination logic creation in terms of SENSEI orchestrations, essentially applying the approach to itself, and, after bootstrapping, allowing SCAffolder to generate its own integration code.

The research results presented in this thesis have also been shared with the scientific community through publications and presentations at conferences and workshops. This spans from first ideas [Jelschen and Winter, 2011] and application scenarios [Jelschen and Winter, 2012], over individual topics such as service discovery and description [Jelschen, 2013], the metamodel [Jelschen et al., 2013], and capabilities [Jelschen and Winter, 2014], to the overall approach [Jelschen, 2014a, 2015] and its applications in software evolution [Jelschen, Meier, and Winter, 2015] and application development [Akyol et al., 2017; Jelschen et al., 2016]. Ongoing research without the author's direct involvement [Hebig et al., 2018; Kuryazov, Winter, and Sandau, 2019] has continued to rely on SENSEI, as well.

## 18.2 Limitations

As demonstrated in the previous chapters, the SENSEI approach satisfies its objectives of providing a toolchain-building support framework for increased flexibility, reusability, and productivity. Its design has been tailored towards these goals, and while SENSEI has also been shown to be very versatile, there are aspects of tool integration that are therefore out of its scope.

For one, SENSEI is specifically aimed at process integration, as this aspect had been identified as both the most relevant to the problem of toolchain-building, as well as one that is not covered by most existing approaches. Many previous works have focused on

data integration, instead. SENSEI offers data integration, as well, but on a comparatively basic level, only, excluding, for example, a common exchange format or data model. This makes it very flexible on the one hand, but also means that some manual effort has to be spent on data integration. This is softened by SENSEI's reusable transformers, reducing the required effort over time, as more transformers become readily available. In addition, the data flow concept has been kept very generic and extensible, so that it can be complemented with existing data integration frameworks.

The focus on process integration also excludes interactivity. This is not an issue when creating software evolution toolchains, which are specifically aimed at eliminating the need for human intervention by automating as much of the migration or reengineering processes as possible. In general application development, this means that the use of SENSEI is constrained to the central business logic, i.e. the middle tier in a classical three-tiered architecture, sandwiched by the presentation and data tiers.

It should also be mentioned that the benefits imparted by the use of SENSEI are not completely free. For example, its flexibility is, to some degree, a tradeoff against runtime performance, as it assumes all components to be completely decoupled, preventing performance optimizations that might be possible in more cohesive applications. For use cases in which the flexibility provided by SENSEI is desirable, yet individual parts in the overall process need to be optimized for performance, it is always possible to achieve that by choosing more coarse-grained services and making the optimizations on the level of implementing components.

SENSEI requires explicit service descriptions, component registration, and orchestration modeling, which presents an overhead not present in ad hoc tool integration. The application of the framework have shown that the benefits of flexibility and reusability quickly offset this. Small toolchains that are neither expected to need to evolve after their initial integration, nor contain parts that might be reusable in the future, will obviously not benefit from SENSEI, as this describes a use case diametrically opposed to the objectives of this thesis. As most software in use today is under constant pressure to evolve, it is also a very rare one.

An aspect of SENSEI that can actually be expensive to change are service definitions, as it would potentially impact all existing uses in orchestrations and implementations in components. This puts a lot of pressure on catalog maintainers to get services right the first time. While not observed as a problem during the case studies that applied SENSEI, this might become more of an issue in very large, long running projects.

## 18.3 Outlook

SENSEI is a framework for tool integration aimed at industrial application, but as the result of extensive scientific research as presented in this thesis, it also represents a launching platform for further studies. Coming to a close, this section highlights opportunities for future research, and sums up the continued relevance of the presented approach in the context of current software engineering industry trends.

### 18.3.1 Future Research

To start with, an aspect worth investigating is how to endow domain experts with more control over non-functional properties of the integrated solutions provided by SENSEI, for example to alleviate the need to trade off performance against flexibility. SENSEI already provides a foundation to address this without sacrificing the strong separation of specification and implementation: the capability model enables domain experts to declaratively specify requirements imposed on service-implementing components. They could be extended to also allow the declaration of non-functional, technical, and potentially orchestration-spanning requirements.

This would be somewhat comparable to *aspect-oriented programming*, in that it could address cross-cutting concerns, which the tooling of SENSEI would automatically weave into integrated solutions; a prime example of such concerns is *logging* to trace the progress of running toolchains. Further examples include technical capabilities that require the SENSEI-produced, integrated toolchain to provide *caching* strategies as one possible means for performance optimization, enable running toolchains to be *paused and resumed* at a later time, require certain data integration strategies like storing and retrieving all exchanged data to and from a central *repository*, or impose *transactionality*, enforce service properties such as *idempotency*, or have orchestrations mapped to an event-based architecture in which components are *signaling* each other (closer in style to service choreographies).

Another avenue of future research could explore ways to further improve the structure of the service catalog. SENSEI assumes that service definitions are highly stable, which is reasonable for most projects. However, for very large, long-running projects, and more generally, for very large service catalogs used in many projects (e.g. curated catalogs made available publicly), the need to evolve service descriptions, as well, may arise. One very straight-forward extension would be to enable *versioning* of services, immediately enabling their evolution without impacting existing uses. SENSEI facilitates such studies with its established metamodel, providing a solid foundation.

Related to these issues are questions regarding the improvement of managing very large orchestrations. In particular, the relationship of orchestrations with (induced) services, to simplify the decomposition of orchestrations into sub-orchestrations. This could result in metamodel extensions, but this might not be necessary, as SENSEI is already capable of handling nested orchestrations. Probably, an enhanced editor would be able to provide sufficient syntactic sugar to improve usability in these regards, while still mapping its models to the unchanged structures of the SENSEI metamodel.

In the NEMo project, the introduction of *quality of service* (QoS) attributes within the SENSEI framework has been considered [Jelschen et al., 2016]. This is an important issue when applying SENSEI in application development and integration, particularly for service-oriented and cloud-deployed applications. In this field, it is common to establish bounds for quality criteria that individual services need to provide, such as its availability for example, in *service-level agreements* (SLA; see Section 7.4). These qual-

ity properties are measured, especially for services provided by a third party, where the SLA is part of a legal contract, and falling below the agreed QoS may incur corresponding penalties.

With its capability model, SENSEI provides a powerful basis that could be extended to cover QoS attributes. QoS capabilities would allow to automatically find components for services with specific quality requirements, e.g. in terms of correctness, performance, and responsiveness. Besides the ability to express required and provided quality of service, a capability mechanism extended in this manner could be further exploited by generating composers that automatically measure QoS and check compliance to the corresponding SLA. If evaluated at runtime (by an interpreter like SNOrcInS), it could also enable highly context-sensitive applications, whose orchestrated services are mapped to different components, dynamically, based on the quality needs and provisions at that moment.

Another observation from NEMo was the need to complement SENSEI-integrated application code with interactive user interfaces. This has already led to research and the successful development of a framework for the modeling, integration, and generation of user interfaces and interactive applications, called DORI [Bischopink et al., 2018; Schlömer, 2017], which transfers the underlying principles of SENSEI and mirrors its basic concepts. Due to its analogous architecture, DORI can be considered a sibling of SENSEI, but it also integrates with SENSEI: users interacting with elements of DORI-provided user interfaces can trigger actions that lead to the execution of SENSEI-integrated toolchains, whose results may in turn lead to a state change in the user interface.

To return to the previously used three-tiered view of applications, this addresses SENSEI's confinement to the middle tier, with DORI already covering the top-level presentation tier. This leaves the bottom data tier – exploring whether the SENSEI principles can be applied here is a task for future research. This would be a major stepping stone towards a software development framework, based on the concepts presented by this thesis, that covers all layers of software applications. The core principles of SENSEI provide the necessary guidance and foundations for the development of such a comprehensive framework.

### 18.3.2 Practical Relevance

In terms of software development methodology, industry has long been striving for less rigid and more agile process models. Due to its conferred flexibility, using SENSEI to specify software functionality and automate integration can help keeping turn-around cycles short, so that changing requirements can quickly be accounted for.

With regards to the architecture of application landscapes, *microservices* are becoming the predominant paradigm being pursued. Here, using SENSEI can be a huge advantage: service catalogs and component registries are natural starting places for es-

tablishing central, company-wide reference repositories of all available microservices, and SENSEI orchestrations provide a way to model business processes and build up value-added services from more basic ones, resulting in potentially much more flexible applications and a corresponding competitive edge.

Microservices should not be directly equated with SENSEI services, as the former usually refers to both the conceptual service, its technical interface (often REST), and its implementation (the component). In terms of central properties, the two concepts do make a good match, as both demand well-defined boundaries based on functional, rather than technical considerations, and are ideally modeled stateless. The separation of service and component, and the abstraction from deployment concerns naturally enable the need for independent deployment of microservice implementations, and, of course, facilitate automatic integration.

Another opportunity for applying SENSEI is the modeling and integration of heterogeneous, distributed, software-intensive *cyber-physical systems* [Harrison, Vera, and Ahmad, 2016]. Systems like that are emerging with trends like the *internet of Things* (IoT) and *industry 4.0*. Due to their inherent complexity, their specification, development, and integration is expected to hugely benefit from the abstraction layer provided by SENSEI.

Model-driven techniques, in general, are considered a crucial ingredient in these fields [Artikov, Kuryazov, and Winter, 2019]. *Smart Modeling* [2020], a joint project of the University of Oldenburg and the Tashkent University of Information Technologies, with support from industry partners, is currently underway to investigate novel model-driven approaches to meet the challenges posed by cyber-physical systems. SENSEI is among those approaches as a framework for model-driven systems and services integration, to address the need for flexibility, raised especially by dynamic, fast-evolving IoT applications.

Lastly, it can be noted that legacy software systems remain to be a huge challenge for many corporations. Market pressure drives them to modernize their application landscapes, e.g. by adopting the aforementioned technologies such as microservices, streamline their development processes by trying to embrace agile methods and implement DevOps practices like continuous integration, delivery, and deployment, as well as transitioning at least partly into the cloud. The existing legacy applications have to somehow be taken along on these journeys, so software evolution and migration projects remain an important reality of the IT industry.

The original motivation for SENSEI to be applied for toolchain integration in the field of software evolution is therefore as relevant as ever. Adopting SENSEI in a software evolution project will help to ensure cost-effectiveness, by reducing the effort required to automate modernization processes as much as possible. Its flexible toolchains built from reusable components enable projects to remain agile in the face of rapidly changing requirements, conditions, and technologies.

# Appendices

APPENDIX A

# SENSEI Models

In the following, the full contents of the SENSEI models created for the Q-MIG project and the NEMo project are documented. This is provided as a source of reference; the application of SENSEI to these projects is described in Chapter 15 and Chapter 16 with detailed explanations. The outline for this chapter is given by the following table of contents:



## A.1 The Q-MIG SENSEI Model

Section A.1.1 lists the contents of the Q-MIG service catalog: Figure A.1 depicts all data structures as class diagram. In the following, all the services of the catalog are listed in alphabetical order. Section A.1.2 lists the orchestrations modeled for Q-MIG. Section A.1.3 lists the contents of the component registry.

### A.1.1 The Q-MIG Service Catalog

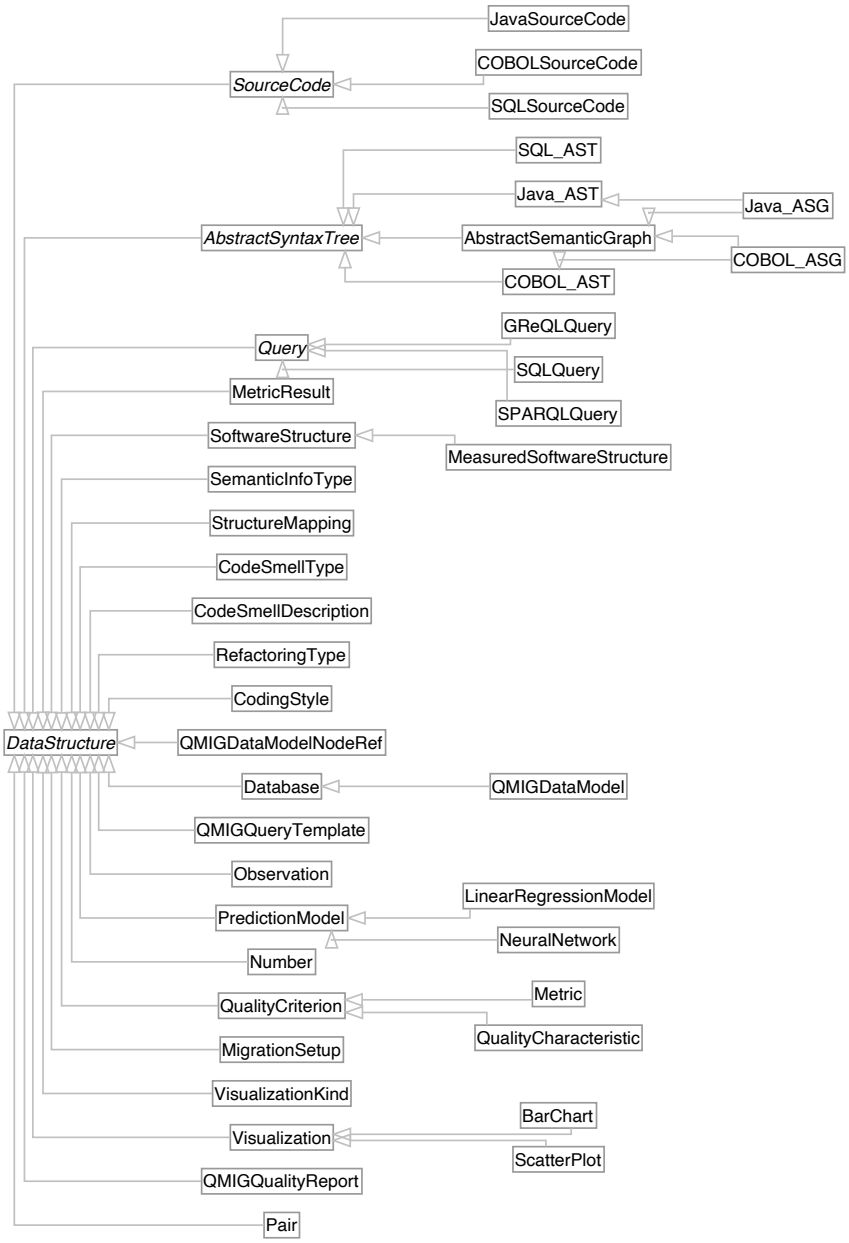



Figure A.1: Q-MIG data structures.

| | |
|---|---|
| Name | AnalyzeDataFlow |
| Description | Takes an abstract syntax tree of a software system and enriches it with data flow relations. |
| Input | ast : AbstractSyntaxTree<br>type : SemanticInfoType |
| Output | asg : AbstractSemanticGraph |
| Capability | ProgrammingLanguage = {COBOL} |
| Classes | |

| | |
|---|---|
| Name | CalculateAllMetrics |
| Description | |
| Input | metrics<br>sourceCode |
| Output | mappedResults |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | CalculateDerivedMetric |
| Description | Derives values for the specified composite or aggregate metric for all software systems and their subsystems found in the repository that miss the value, if the necessary base metric data is present. |
| Input | repository : QMIGDataModel<br>metric : Metric |
| Output | updatedRepo : QMIGDataModel |

| | |
|---|---|
| Capability Classes | SupportedMetrics = {ModularityRating, ReusabilityRating, AnalysabilityRating, ChangeabilityRating, ModificationStabilityRating, TestabilityRating, ComplexityRating, PortabilityRating, ELOC, McCabe, HalsteadVocab, HalsteadSize, HalsteadVolume, HalsteadDifficulty, HalsteadEfforts, HalsteadErrors, HalsteadTestingTime, CommentsPercentage, ClonesPercentage, AverageLCOM, AverageComplexityPerUnit, AverageUnitSize, AverageNumberSubclasses, AttributeHidingFactor, MethodHidingFactor, AverageMethodsPerClass, SLOC, NumberEmptyLines, NumberLinesWithOnlyBrackets, NumberCommentLines, NumberDecisions, NumberDistinctOperators, NumberDistinctOperands, NumberOperatorInstances, NumberOperandInstances, NumberGotos, CloneLines, LCOM, NumberClasses, NumberSubclasses, NumberAttributes, NumberHiddenAttributes, NumberMethods, NumberHiddenMethods, ITD, SwitchesFromGotos, UnconvertibleGotos, ConditionalOperators, CBO, SLOCWithoutSQL, ELOCWithoutSQL, NumberAttributesAccessed, SumDistinctAttributesAccessed, NumberSQLLines} |

| | |
|---|---|
| Name | CalculateMetric |
| Description | Evaluates the specified metric over a software system's abstract syntax tree and returns the result. |
| Input | metric : Metric<br>ast : AbstractSyntaxTree |
| Output | result : MetricResult |

| | |
|---|---|
| Capability Classes | SupportedMetrics = {ModularityRating, ReusabilityRating, AnalysabilityRating, ChangeabilityRating, ModificationStabilityRating, TestabilityRating, ComplexityRating, PortabilityRating, ELOC, McCabe, HalsteadVocab, HalsteadSize, HalsteadVolume, HalsteadDifficulty, HalsteadEfforts, HalsteadErrors, HalsteadTestingTime, CommentsPercentage, ClonesPercentage, AverageLCOM, AverageComplexityPerUnit, AverageUnitSize, AverageNumberSubclasses, AttributeHidingFactor, MethodHidingFactor, AverageMethodsPerClass, SLOC, NumberEmptyLines, NumberLinesWithOnlyBrackets, NumberCommentLines, NumberDecisions, NumberDistinctOperators, NumberDistinctOperands, NumberOperatorInstances, NumberOperandInstances, NumberGotos, CloneLines, LCOM, NumberClasses, NumberSubclasses, NumberAttributes, NumberHiddenAttributes, NumberMethods, NumberHiddenMethods, ITD, SwitchesFromGotos, UnconvertibleGotos, ConditionalOperators, CBO, SLOCWithoutSQL, ELOCWithoutSQL, NumberAttributesAccessed, SumDistinctAttributesAccessed, NumberSQLLines} SupportedProgrammingLanguage = {COBOL, Java}<br>SupportedGranularityLevel = {COBOLSection, COBOLDivision, File, JavaPackage, Directory, JavaClass, JavaMethod} |

| | |
|---|---|
| Name | Concat |
| Description | Takes two ordered collections of arbitrary length (including length of zero or one), and returns a single ordered collection, containing all elements of the front collection in their original order, followed by all elements of the rear collection in their original order. |
| Input | front [*] : DataStructure<br>rear [*] : DataStructure |
| Output | concatenated [*] : DataStructure |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | DefineTargetArchitecture |
| Description | Takes a source system and a mapping of structural source system elements to target system elements, and enriches the source system's AST with information to guide a later migration. |

| | |
|---|---|
| Input | sourceSystem : AbstractSyntaxTree<br>targetMapping : StructureMapping |
| Output | modifiedSystem : AbstractSemanticGraph |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | DetectCodeSmell |
| Description | Searches the provided software system for the presence of code smells of the given type, and returns a set of all found instances of that code smell. |
| Input | type : CodeSmellType<br>ast : AbstractSyntaxTree |
| Output | codeSmells : CodeSmellDescription |
| Capability | CodeSmellType = {BadGOTO} ProgrammingLanguage = |
| Classes | {COBOL} |

| | |
|---|---|
| Name | ExtractEmbeddedCode |
| Description | Extracts unparsed, embedded source code fragments from a host language AST. |
| Input | ast : AbstractSyntaxTree |
| Output | code : SourceCode |
| Capability | HostLanguage = {COBOL} EmbeddedLanguage = {SQL} |
| Classes | |

| | |
|---|---|
| Name | ExtractStructure |
| Description | Extracts a basic, hierarchical decomposition of a software system's AST. |
| Input | ast : AbstractSyntaxTree |
| Output | structure : SoftwareStructure |
| Capability | NoChoice = {NoChoice} |
| Classes | |

| | |
|---|---|
| Name | FormatCode |
| Description | Takes source code in a particular programming language and formats it according to the specified coding style. |
| Input | style : CodingStyle |

| | |
|---|---|
| | source : SourceCode |
| Output | formattedSource : SourceCode |
| Capability | CodingStyle = {JavaOracle} |
| Classes | |

| | |
|---|---|
| Name | GenerateCode |
| Description | Takes an AST representation of a software system and generates the corresponding source code files. |
| Input | ast : AbstractSyntaxTree |
| Output | code : SourceCode |
| Capability | ProgrammingLanguage = {Java} |
| Classes | |

| | |
|---|---|
| Name | LearnSupervised |
| Description | Takes a training set of observations (predictors / independent variables, paired with the corresponding desired outcome / dependent variable) and fits a prediction model to this data. |
| Input | trainingSet [*] : Observation |
| Output | model : PredictionModel |
| Capability | supervisedLearningTechnique = {LinearRegression, |
| Classes | NeuralNetworkBackPropagation} |

| | |
|---|---|
| Name | MapResultsToStructure |
| Description | Takes a hierarchical decomposition of a software system, as well as a series of metric values calculated on the same system, and combines the two by annotating the structural elements with their corresponding metric values. |
| Input | results [*] : MetricResult<br>structure : SoftwareStructure<br>metric [*] : Metric |
| Output | mappedResults : MeasuredSoftwareStructure |
| Capability | NoChoice = {NoChoice} |
| Classes | |

| | |
|---|---|
| Name | Pair |

| | |
|---|---|
| Description | Pairs two arbitrary data structures together. A pair can also be used in place of a collection (of exactly two elements). |
| Input | left : DataStructure<br>right : DataStructure |
| Output | pair : Pair |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | Parse |
| Description | Parses code files and returns an abstract syntax tree / graph (AST/ASG) representation. |
| Input | source code : SourceCode |
| Output | outAST : AbstractSyntaxTree |
| Capability | Programming Language = {COBOL, Java, SQL} |
| Classes | |

| | |
|---|---|
| Name | Predict |
| Description | Takes a set of predictors / independent variables, and returns the expected outcome / dependent variable according to the provided prediction model. |
| Input | model : PredictionModel<br>predictors [*] : Number |
| Output | prediction : Number |
| Capability | modelType = {LinearRegression, NeuralNetwork} |
| Classes | |

| | |
|---|---|
| Name | QMIGAddAverageValues |
| Description | For metrics that allow this, fills missing metric values of a software system with the average of values measured on its subsystems. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGAddCommonRoot |

| | |
|---|---|
| Description | Adds a new software system node to the repository, and adds all specified, existing subsystems as its children. |
| Input | repository : QMIGDataModel<br>subsystems [*] : QMIGDataModelNodeRef |
| Output | updatedRepo : QMIGDataModel<br>root : QMIGDataModelNodeRef |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGAddDefaultNames |
| Description | Where possible, fills missing name attributes of software systems with sensible default values. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGAddTraceLinks |
| Description | Adds trace links between the source and target software systems, and all their respective subsystems, assuming source has been migrated to target using the provided migration setup (containing necessary information about the migration process, mapping rules, etc.). If no target system reference is provided, it is created along with the trace links, in accordance with the migration setup. |
| Input | repository : QMIGDataModel<br>sourceSystem : QMIGDataModelNodeRef<br>targetSystem : QMIGDataModelNodeRef<br>setup : MigrationSetup |
| Output | updatedRepo : QMIGDataModel<br>targetSystem : QMIGDataModelNodeRef |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGAverageOutDuplicateRatings |
| Description | Removes duplicate ratings for a software system by the same expert, and replaces them with the average of the provided rating values. |
| Input | repository : QMIGDataModel |

| | |
|---|---|
| Output | updatedRepo : QMIGDataModel |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGCalculateBaseMetrics |
| Description | |
| Input | metrics [*] : Metric |
| | ast : AbstractSyntaxTree |
| Output | mappedResults : MeasuredSoftwareStructure |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGConsolidateData |
| Description | |
| Input | qmigRepo : QMIGDataModel |
| | qualityDataSets [*] : QMIGDataModel |
| Output | consolidatedRepo : QMIGDataModel |
| | integratedSystem : QMIGDataModelNodeRef |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGExtractData |
| Description | |
| Input | queryTemplate : QMIGQueryTemplate |
| | independentCriteria [*] : QualityCriterion |
| | systems [*] : QMIGDataModelNodeRef |
| | qmigRepo : QMIGDataModel |
| | dependentCriterion : QualityCriterion |
| Output | result : DataStructure |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGGenerateReport |
| Description | Generates a quality report that presents the data associated with the specified software systems contained within the repository in an easily browsable, human-readable format. |

| | |
|---|---|
| Input | repository : QMIGDataModel<br>systems : QMIGDataModelNodeRef |
| Output | report : QMIGQualityReport |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGImportData |
| Description | Merges all the provided data model instances into the specified, existing repository. |
| Input | repository : QMIGDataModel<br>importData [*] : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel<br>importedEntries [*] : QMIGDataModelNodeRef |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGInferMissingRatings |
| Description | Tries to infer missing rating values by exploiting certain redundancies in the hierarchical subsystem structure, and by using the average of subsystem ratings, if the parent system's rating value is missing. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGIntegrateData |
| Description | |
| Input | qmigRepo : QMIGDataModel<br>subsystemQualityEntries [*] : QMIGDataModelNodeRef |
| Output | integratedRepo : QMIGDataModel<br>rootSystem : QMIGDataModelNodeRef |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGObfuscateIdentifiers |

| | |
|---|---|
| Description | Replaces all software (sub-)system names with arbitrarily chosen identifiers. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGParseMigrateMeasure |
| Description | |
| Input | COBOLSource : COBOLSourceCode<br>typesOfSemanticInfo : SemanticInfoType<br>targetStructure : StructureMapping<br>reengTypes [*] : CodeSmellType<br>COBOLMetrics [*] : Metric<br>javaCodeFormat : CodingStyle<br>JavaMetrics [*] : Metric |
| Output | sqlAST : SQL_AST<br>COBOLQuality : MeasuredSoftwareStructure<br>JavaQuality : MeasuredSoftwareStructure<br>javaSource : JavaSourceCode |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGParseMigrateMeasureMultiple |
| Description | |
| Input | COBOLPrograms [*] : COBOLSourceCode<br>targetStructures [*] : StructureMapping<br>typesOfSemanticInfo : SemanticInfoType<br>reengTypes [*] : CodeSmellType<br>COBOLMetrics [*] : Metric<br>javaMetrics [*] : Metric<br>javaCodeFormat : CodingStyle |
| Output | sqlASTs [*] : SQL_AST<br>javaSources [*] : JavaSourceCode<br>javaMeasurements [*] : MeasuredSoftwareStructure<br>COBOLMeasurements [*] : MeasuredSoftwareStructure |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGPersistPrediction |
| Description | |
| Input | repository : QMIGDataModel<br>system : QMIGDataModelNodeRef<br>criteria [*] : QualityCriterion<br>values [*] : Number |
| Output | updatedRepo : QMIGDataModel |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGPrepareQuery |
| Description | Builds a query expression based on the provided template, by inserting constraints based on the specified software systems and quality criteria to be taken into consideration. |
| Input | queryTemplate : QMIGQueryTemplate<br>independentCriteria [*] : QualityCriterion<br>systems [*] : QMIGDataModelNodeRef<br>dependentCriterion : QualityCriterion |
| Output | query : GReQLQuery |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGReengineer |
| Description | |
| Input | softwareSystem : AbstractSyntaxTree<br>codeSmellTypes [*] : CodeSmellType |
| Output | reengineeredSystem : AbstractSyntaxTree |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGRemoveDuplicateMeasurements |
| Description | Eliminates measurement entry clones, leaving only one entry for a single software system node. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |

| | |
|---|---|
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGRemoveDuplicateSoftwareSystems |
| Description | Merges sets of clones representing the same subsystem into a single entry. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGResolveAmbiguousIDs |
| Description | Adds contextual information to ID strings to remove ambiguity and ensure repository-wide uniqueness of all software system IDs. |
| Input | repository : QMIGDataModel |
| Output | updatedRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGSanitizeMeasurements |
| Description | |
| Input | qmigRepo : QMIGDataModel |
| Output | sanitizedMeasurementsRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGSanitizeRatings |
| Description | |
| Input | qmigRepo : QMIGDataModel |
| Output | sanitizedRatingsRepo : QMIGDataModel |
| Capability Classes | Default = {Default} |

| | |
|---|---|
| Name | QMIGTrainAndPredict |
| Description | |

| | |
|---|---|
| Input | criteriaToPredict<br>trainingTemplate<br>testTemplate<br>independentCriteria<br>testSystems<br>trainingSystems<br>qmigRepo<br>predictedSystem |
| Output | repoWithPrediction |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | QMIGTrainPredictionModel |
| Description | |
| Input | queryTemplate : QMIGQueryTemplate<br>systems [*] : QMIGDataModelNodeRef<br>independentCriteria [*] : QualityCriterion<br>qmigRepo : QMIGDataModel<br>dependentCriterion : QualityCriterion |
| Output | model : PredictionModel |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | Query |
| Description | Queries the specified data repository of arbitrary kind using the provided query expression, and returns the result. |
| Input | repository : Database<br>query : Query |
| Output | result : DataStructure |
| Capability | QueryLanguage = {GReQL} |
| Classes | |

| | |
|---|---|
| Name | Refactor |
| Description | Removes the provided code smell instance from the input software system, and replaces it with semantically equivalent logic. |
| Input | codeSmell : CodeSmellDescription<br>softwareSystem : AbstractSyntaxTree |

| | |
|---|---|
| Output | refactoredSystem : AbstractSyntaxTree |
| Capability | Refactoring = {EliminateGOTO} ProgrammingLanguage = |
| Classes | {COBOL} |

| | |
|---|---|
| Name | Split |
| Description | |
| Input | pair : Pair |
| Output | left : DataStructure |
| | right : DataStructure |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | Transform |
| Description | Performs a semantic transformation of data. Note: This is a special service used by the CompositionFinder and must not be modified! |
| Input | input : DataStructure |
| Output | output : DataStructure |
| Capability | Transformations = {Identity} |
| Classes | |

| | |
|---|---|
| Name | TransformProgrammingLanguage |
| Description | Transforms a software system in one programming language into a semantically equivalent software system in another programming language. |
| Input | source : AbstractSyntaxTree |
| Output | result : AbstractSyntaxTree |
| Capability | LanguageTransformations = {COBOLtoJava} |
| Classes | |

| | |
|---|---|
| Name | Unzip |
| Description | |
| Input | zippedCollection [*] |
| Output | left [*] : DataStructure |
| | right [*] : DataStructure |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | Visualize |
| Description | Transforms the provided data into a visual representation of the specified kind. |
| Input | data : DataStructure<br>kind : VisualizationKind |
| Output | result : Visualization |
| Capability | SupportedVisualizations = {BarChart, ScatterPlot} |
| Classes | |

| | |
|---|---|
| Name | Zip |
| Description | Takes two arbitrary, ordered collections |
| Input | left [*] : DataStructure<br>right [*] : DataStructure |
| Output | zippedCollection [*] : Pair |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | transformDataDefinition |
| Description | Transforms the format, model, or representation of the provided data. Note: This is a special service used by the CompositionFinder and must not be modified! |
| Input | input : DataStructure |
| Output | output : DataStructure |
| Capability | Transformations = {Identity, QMIGSoftwareSystem2DuDeEntity, |
| Classes | DuDeCloneLines2QMIGMetricResult, SoamigXml2TGraph, QMIGExchange2DataModelTGraph, DataModelTGraph2QMIGExchange} |

### A.1.2 Q-MIG Orchestrations

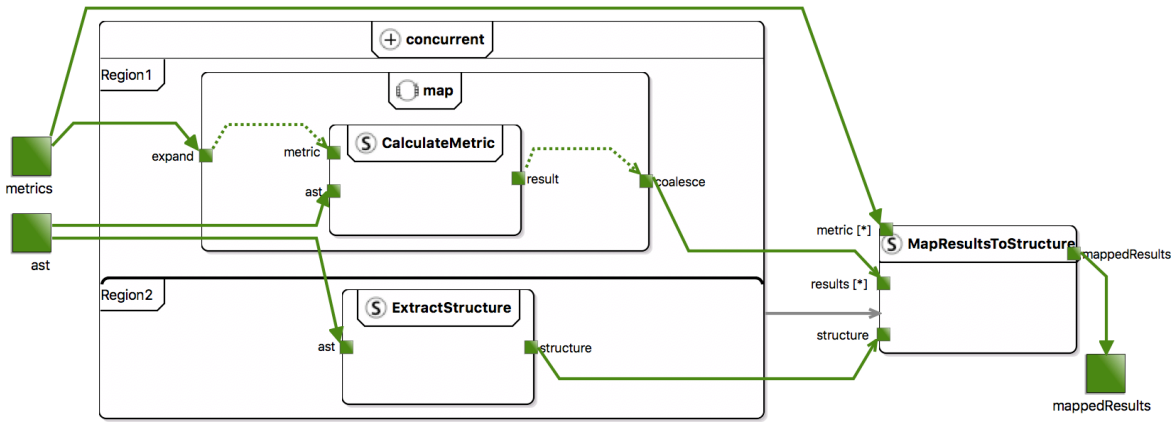



Figure A.2: Q-MIG base metric calculation orchestration, without parsing.

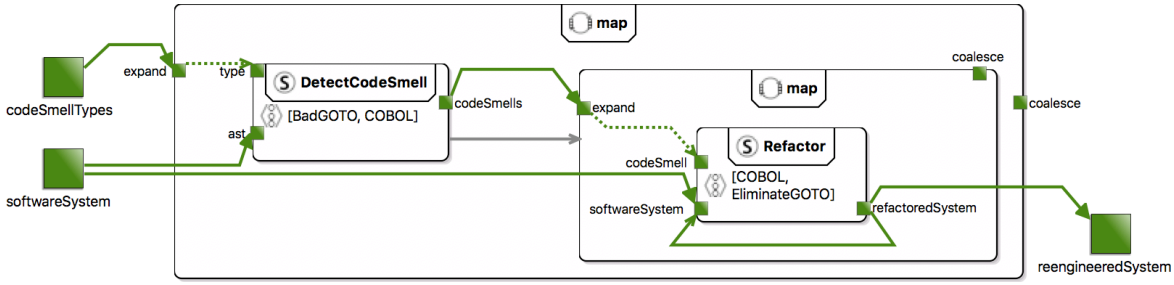



Figure A.3: Q-MIG reengineering orchestration.

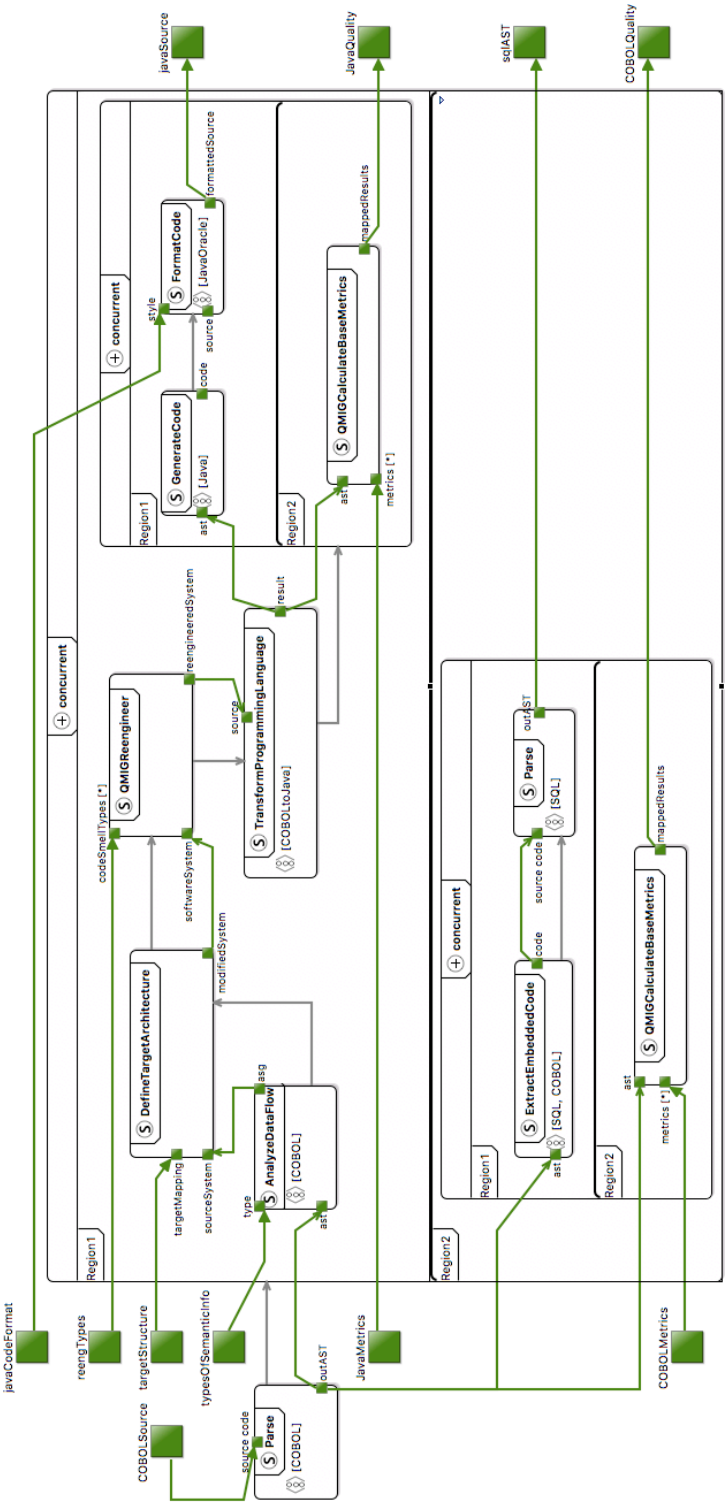



Figure A.4: Orchestration covering parsing, migration, and quality measurement of both COBOL and migrated Java systems.

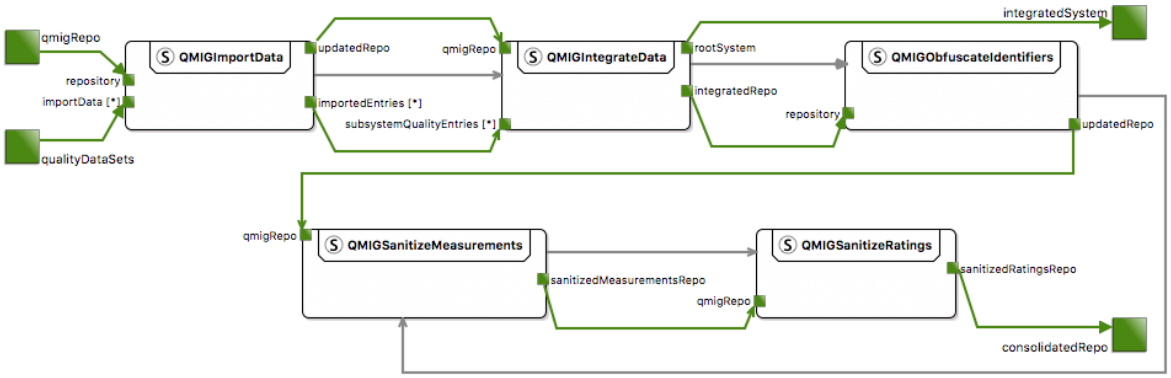



Figure A.5: Q-MIG orchestration to consolidate data.

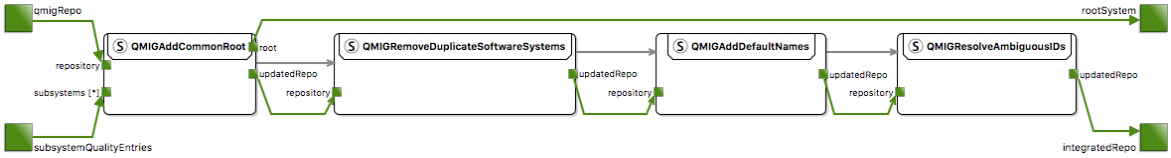



Figure A.6: Q-MIG orchestration to integrate data.

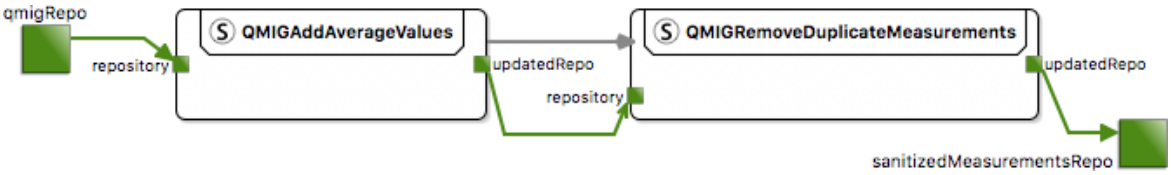



Figure A.7: Q-MIG orchestration to sanitize measurements.

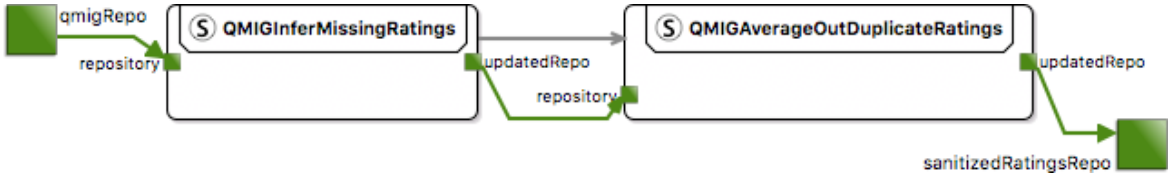



Figure A.8: Q-MIG orchestration to sanitize ratings.

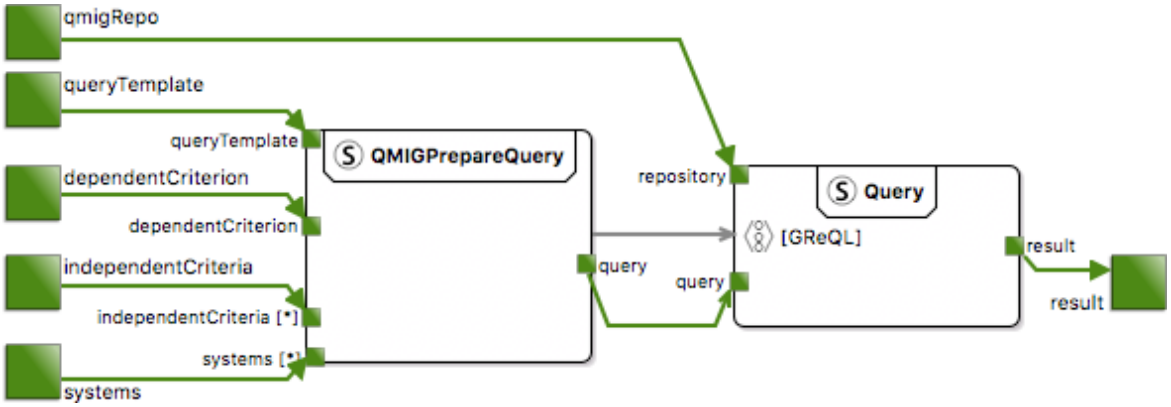



Figure A.9: Q-MIG orchestration to extract data.

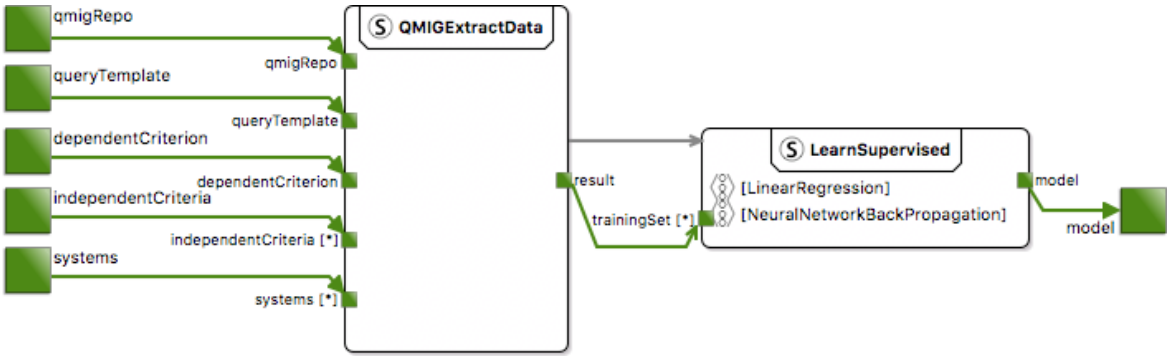



Figure A.10: Q-MIG orchestration to train prediction models.

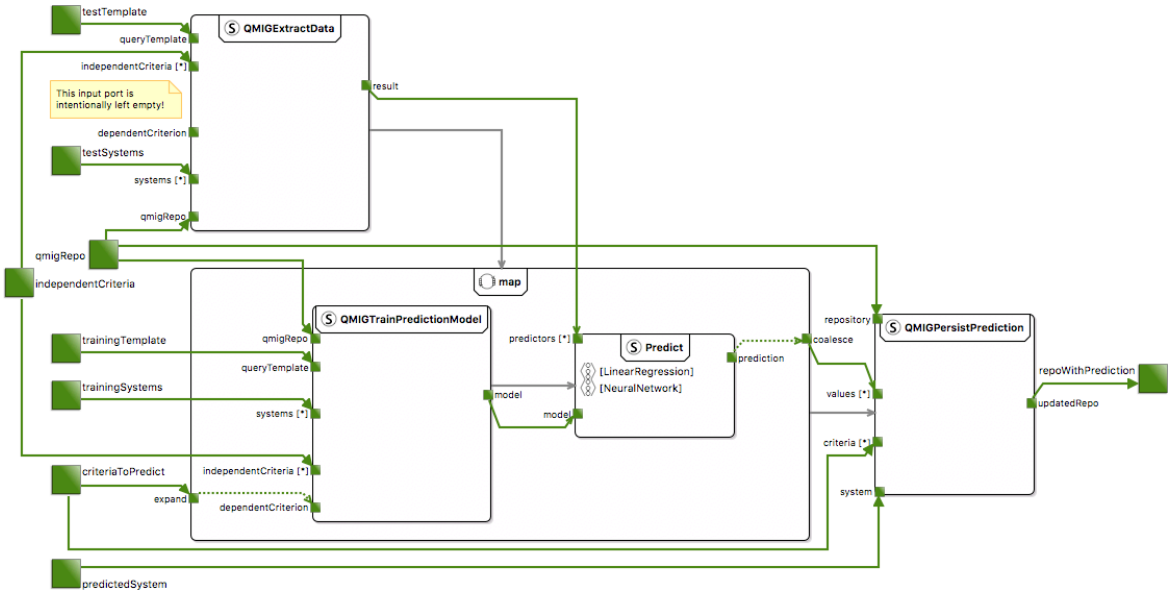



Figure A.11: Q-MIG orchestration to train and predict.

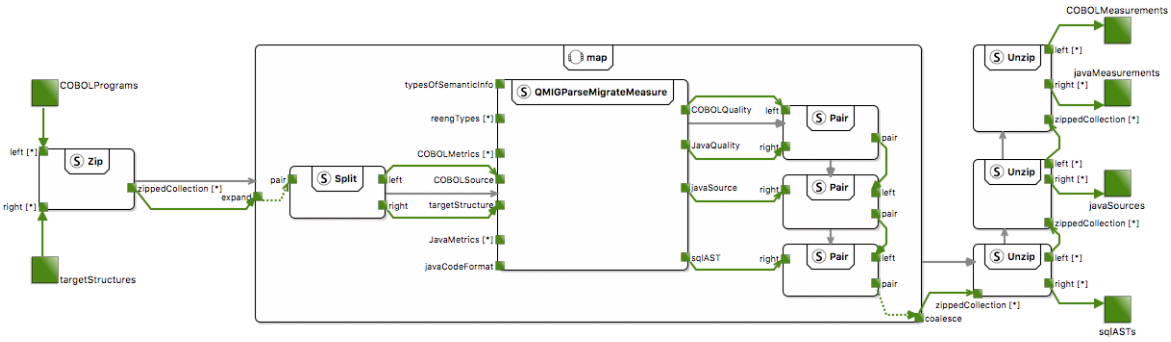



Figure A.12: Q-MIG orchestration to parse, migrate, and measure multiple software (sub-)systems.

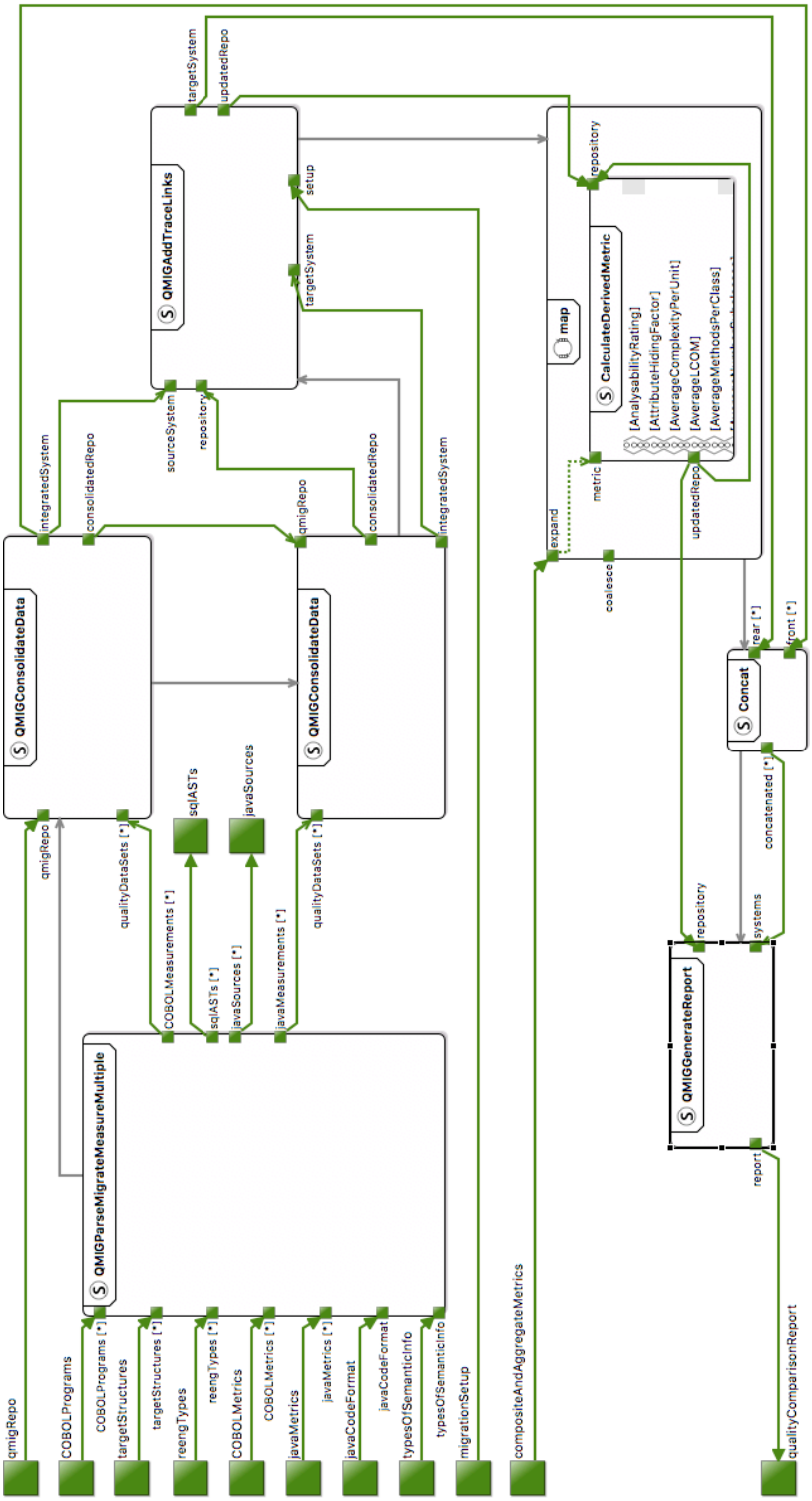



Figure A.13: Orchestration to generate-quality-comparison-report.

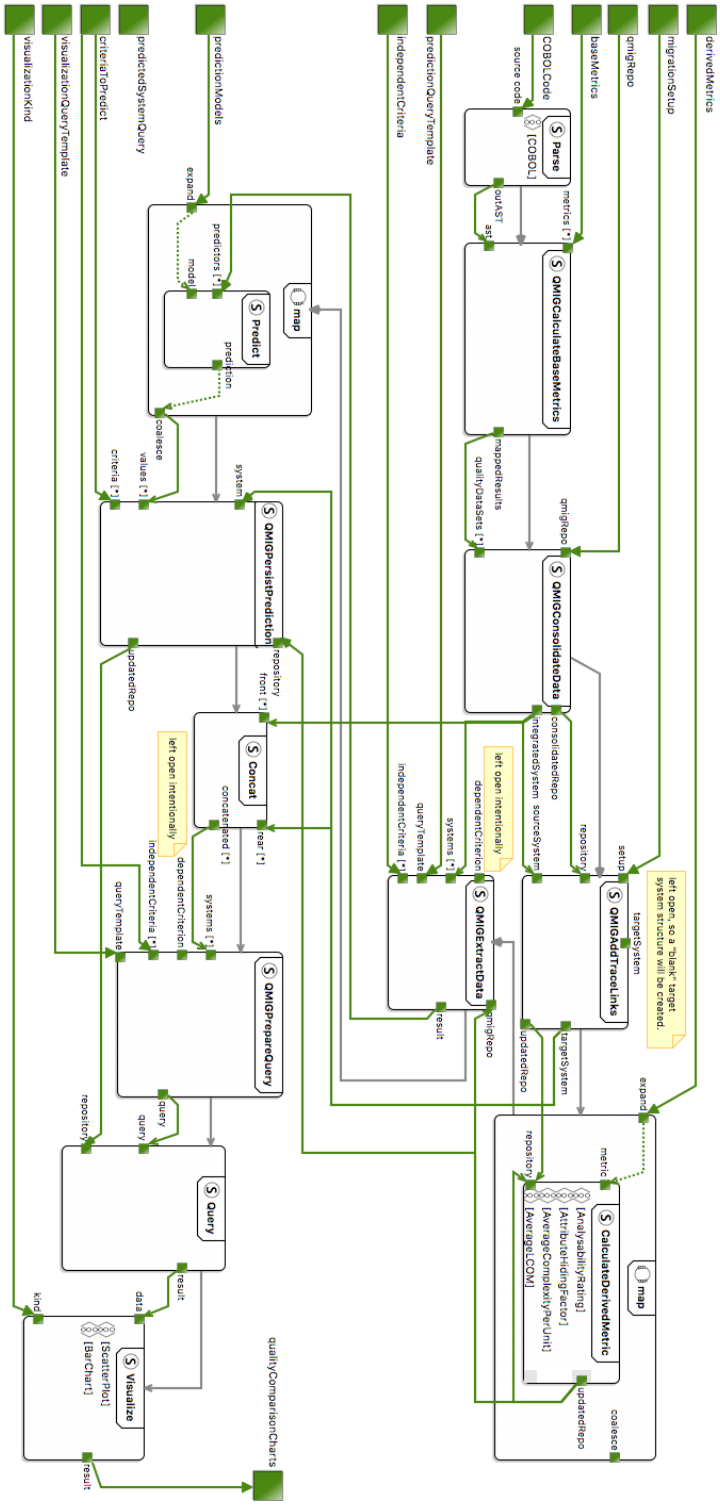



Figure A.14: Orchestration to predict-before-migration.

### A.1.3 The Q-MIG Component Registry

This table covers the components used by the toolchains presented in Section 15.7.

| Component | Service | Capability Tuple |
|---|---|---|
| COBOLParser | Parse | [ COBOL ] |
| DataImprover | QMIGAddAverageValues | [ Default ] |
| | QMIGAddCommonRoot | [ Default ] |
| | QMIGAddDefaultNames | [ Default ] |
| | QMIGAverageOutDuplicateRatings | [ Default ] |
| | QMIGImportData | [ Default ] |
| | QMIGInferMissingRatings | [ Default ] |
| | QMIGObfuscateIdentifiers | [ Default ] |
| | QMIGRemoveDuplicateMeasurements | [ Default ] |
| | QMIGRemoveDuplicateSoftwareSystems | [ Default ] |
| | QMIGResolveAmbiguousIDs | [ Default ] |
| DuDe | CalculateMetric | [ CloneLines, COBOL, Directory ]<br>[ CloneLines, Java, Directory ] |
| | transformDataDefinition | [ QMIGSoftwareSystem2DuDeEntity ]<br>[ DuDeCloneLines2QMIGMetricResult ] |
| IdentityTransformComponent | Transform | [ Identity ] |
| | transformDataDefinition | [ Identity ] |
| IntegrateDataComposer | QMIGIntegrateData | [ Default ] |
| JavaParser | Parse | [ Java ] |
| MetricCalculator | ExtractStructure | [ NoChoice ] |
| | CalculateMetric | [ SLOC, Java, File ]<br>[ SLOC, Java, JavaClass ]<br>[ SLOC, Java, JavaMethod ]<br>[ SLOC, Java, JavaPackage ] |
| | MapResultsToStructure | [ NoChoice ] |
| | transformDataDefinition | [ DataModelTGraph2QMIGExchange ]<br>[ QMIGExchange2DataModelTGraph ]<br>[ SoamigXml2TGraph ] |
| SanitizeMeasurementsComposer | QMIGSanitizeMeasurements | [ Default ] |
| SanitizeRatingsComposer | QMIGSanitizeRatings | [ Default ] |

## A.2 The NEMo SENSEI Model

Section A.2.1 lists the contents of the NEMo service catalog: Figure A.15 depicts all data structures as class diagram. In the following, all the services of the catalog are listed in alphabetical order. Section A.2.2 lists the orchestrations modeled for NEMo. Section A.2.3 lists the contents of the component registry.

### A.2.1 The NEMo Service Catalog

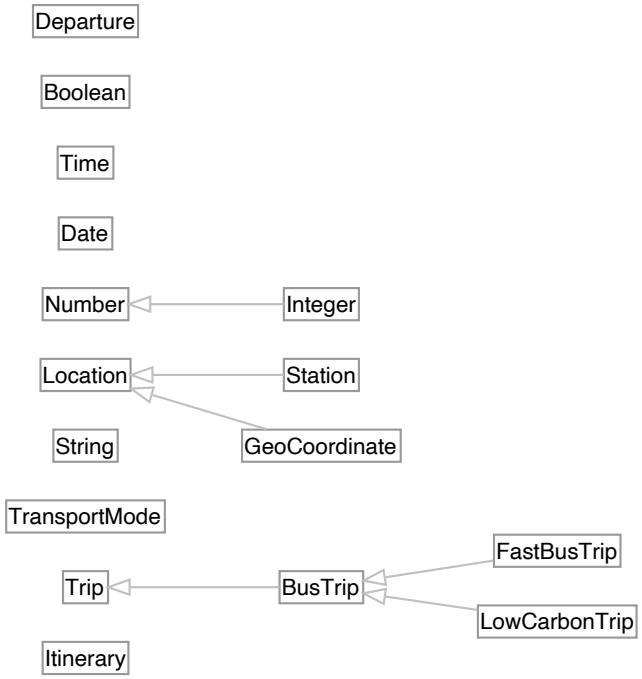



Figure A.15: NEMo data structures.

| | |
|---|---|
| Name | CombineRoutes |
| Description | |
| Input | travelSections [*] : Itinerary |
| Output | interModalTravelInfo : Itinerary |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | FindRoute |
| Description | |
| Input | request : Trip |
| | mode : TransportMode |
| Output | travelInfo : Itinerary |
| Capability | Transport Mode = {car, train, bus, carpool, bike, walking} |
| Classes | Optimization Goal = {fastest, shortest, cheapest, lowestCO2} |

| | |
|---|---|
| Name | FindStops |
| Description | |
| Input | request : Trip |
| Output | sections [*] : Trip |
| Capability | Default = {Default} |
| Classes | |

| | |
|---|---|
| Name | NearestStationFinder |
| Description | Finds the station with the minimal distance to a location |
| Input | location : Location |
| Output | station : Station |
| Capability | TransportMode = {BUS} |
| Classes | |

| | |
|---|---|
| Name | NearestStationsFinder |
| Description | Finds the nearest stations for a transportation mode as specified by capabilities |
| Input | location : Location |
| Output | stations [*] : Station |
| Capability | TransportationMode = {BUS} |
| Classes | |

| | |
|---|---|
| Name | RouteConcatenator |
| Description | Concatenates 3 routes to one route |
| Input | route1 : Itinerary<br>route2 : Itinerary<br>route3 : Itinerary |
| Output | concatenatedRoute : Itinerary |
| Capability | Default = {DEFAULT} |
| Classes | |

| | |
|---|---|
| Name | RouteFinder |
| Description | Finds a route between two locations using a specific transport mode |
| Input | origin : Location<br>destination : Location |
| Output | route : Itinerary |
| Capability | TransportMode = {BUS, WALK} |
| Classes | |

| | |
|---|---|
| Name | RouteSelector |
| Description | |
| Input | routes [*] : Itinerary |
| Output | bestRoute : Itinerary |
| Capability | QualityCriteria = {TIME} |
| Classes | |

### A.2.2 NEMo Orchestrations

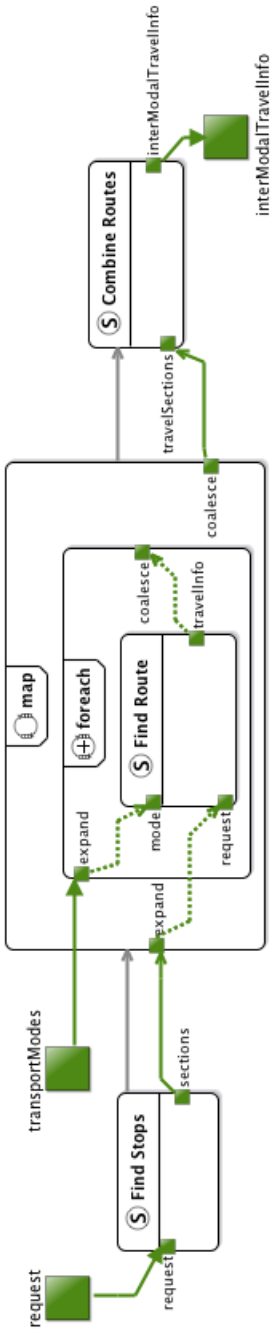



Figure A.16: Inter-modal routing orchestration.

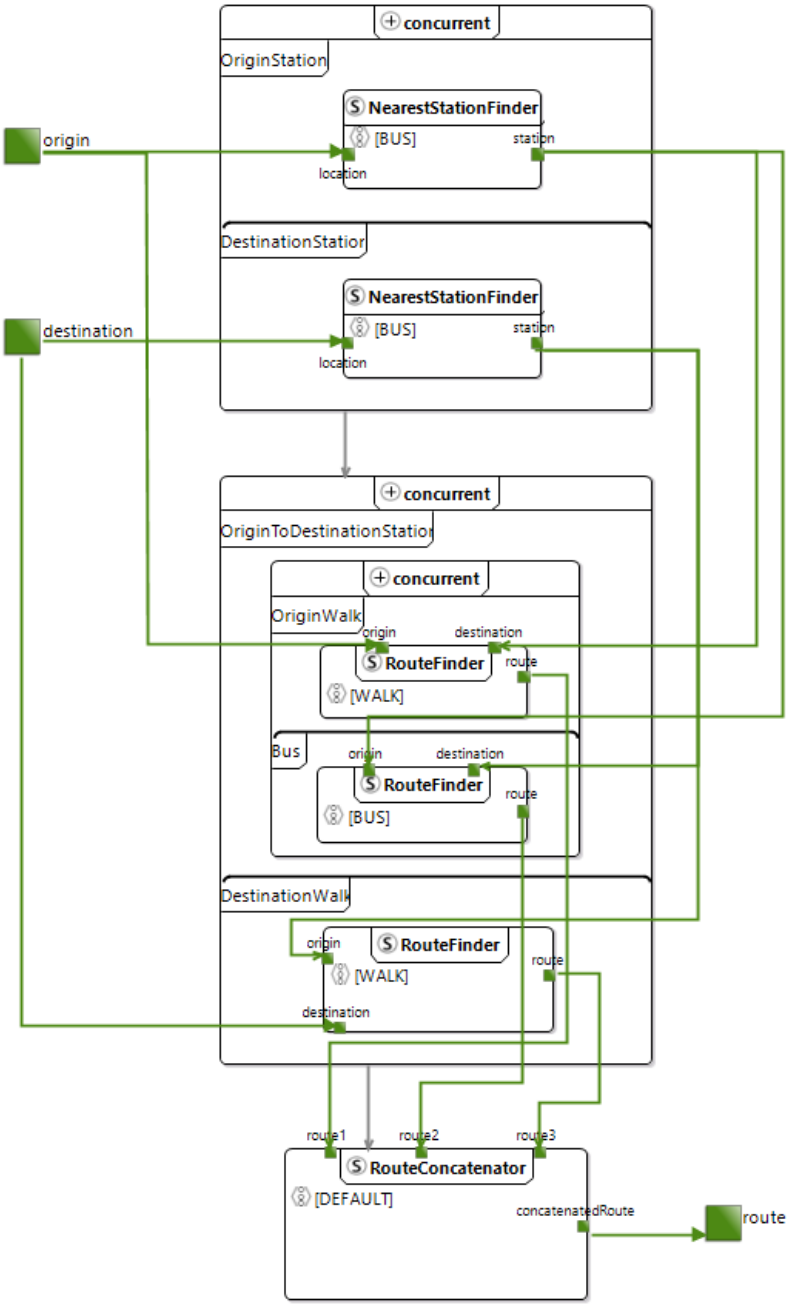



Figure A.17: Bus planning orchestration, taken from Küpker [2015, p. 72].

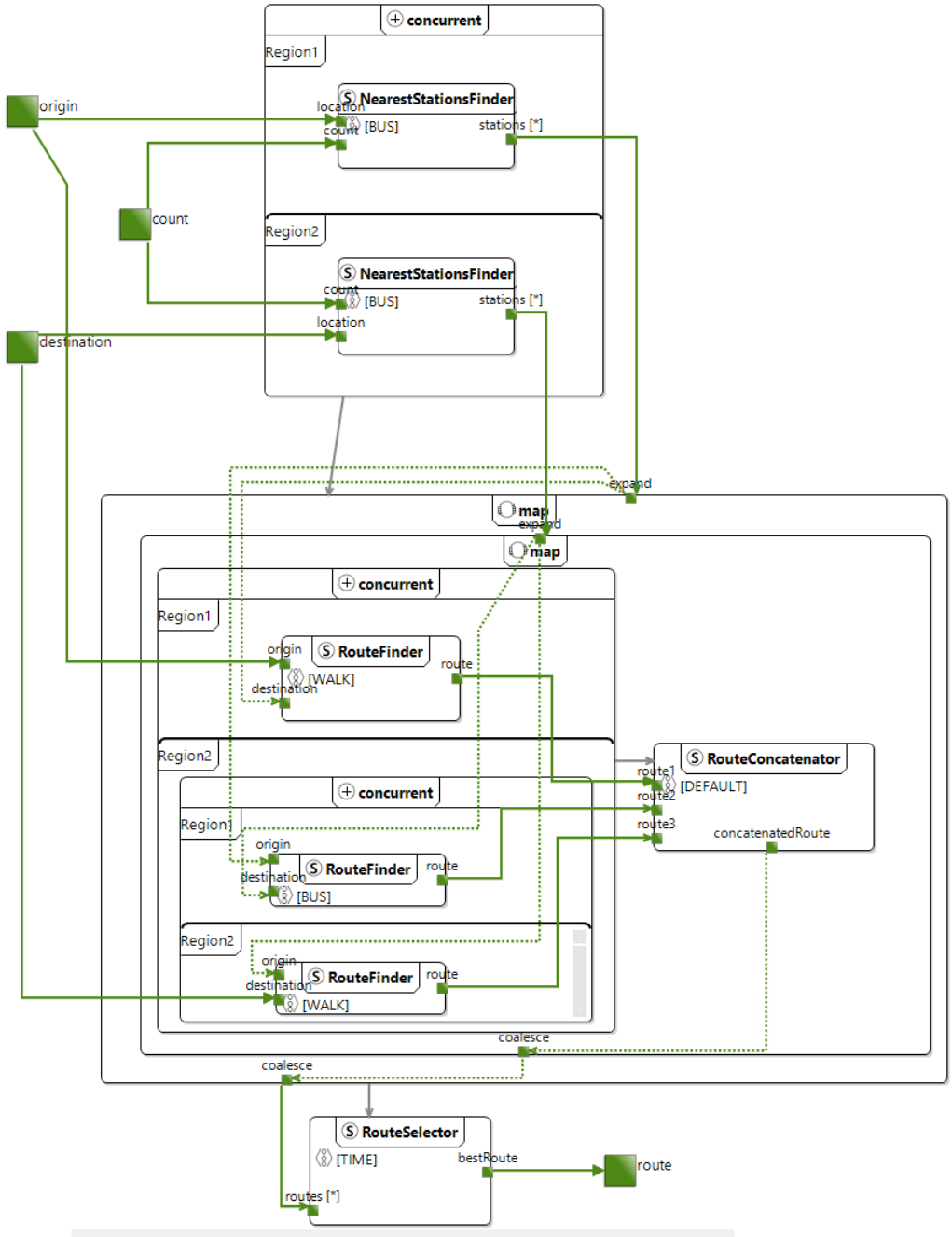



Figure A.18: Optimized bus planning orchestration, taken from Küpker [2015, p. 73].

### A.2.3 The NEMo Component Registry

This table covers the components used by the toolchains presented in Section 16.3. Information on the components mapped to services used for the bus planning case study are documented by Küpker [2015].

| Component | Service | Capability Tuple |
|---|---|---|
| InterModalRouter | CombineRoutes | [ Default ] |
| | FindRoute | [ bus, fastest ] |
| | | [ bus, lowestCO2 ] |
| | | [ carpool, shortest ] |
| | | [ walking, shortest ] |
| | FindStops | [ Default ] |
| IKTSPublicRouter | FindRoute | [ bus, fastest ] |
| | | [ train, fastest ] |
| RouteCalculator | FindRoute | [ car, fastest ] |

APPENDIX B

# SCAffolder Model-to-Model Transformation Reference

Section 14.3 has presented the SENSEI toolchain generator SCAffolder, including a brief overview of the model-to-model transformation stage. This appendix provides a more comprehensive documentation of the implementation. The transformations are grouped as follows:

Section B.1 defines basic utility functions used by other transformation rules. Section B.2 introduces a legacy feature of SCAffolder, a minimal implementation of service-component-matching, which has been superseded by the separate Composition Finder, but remains available, and is included for the sake of completeness. Section B.3 subsumes transformations mainly required for tool stub generation. Section B.4 specifies how SENSEI services are mapped onto SCA services, and corresponding Java interfaces and method signatures. Section B.5 describes how Java types are generated based on data definitions. Section B.6 presents transformations generating the framework for composers. Section B.7 complements this with how orchestration control and data flows are mapped onto Java statements to implement them.

## B.1 Utility Functions

To simplify the transformation process, SCAffolder defines several helper and utility functions. The *Helpers* class contains thirteen variables and twenty-five helper functions, expressed using standard GReTL means. There are some limitations to these means, though: when evaluating the semantic expression of a GReTL operation, the variables it uses a determined by performing a simple text search for each of the previously reg-

```
1  Def.registerQuery(context,"providedCapabilityVector",
2          new String[]
3          {
4              "implementationsMap", "cvEquals"
5          },
6          "using requiredCapabilityVector:\n "
7          + "  (\n "
8          + "    implementationsMap[requiredCapabilityVector] \n "
9          + "    --> &{AbstractServiceInstance @ thisVertex "
10         + "    --> &{Service} <--&{AbstractServiceInstance}"
11         + "    --> requiredCapabilityVector } "
12         + "    --> &{CapabilityVector \n "
13         + "      @
   ↪  cvEquals[tup(requiredCapabilityVector,thisVertex)]}\n "
14         + "  )[0]");
```

Figure B.1: A GReQL function that, given a required capability tuple of a service instance, returns the corresponding provided capability tuple of the component chosen to implement the service instance. It is registered using a custom mechanism to circumvent GReTL's limitations.

istered variables. The GReQL query is then prefixed with a *using* statement, so that those variables can be passed into it.

However, the GReTL implementation does not account for variables used in helper functions, meaning variables cannot be used within helper functions. One solution would be to only use helper functions instead of variables, as the former get registered in the global GReQL function library. This would be *extremely* detrimental to performance, though, as functions get re-evaluated every time they are invoked – GReTL variables are used in SCAffolder to store the results of some expensive calculations.

Therefore, an extension has been implemented in class *Def*, which allows to register functions that use GReTL variables. An example of its use is provided in Figure B.1[1] (a total of six functions are defined using this mechanism in SCAffolder). The extra variables have to be explicitly named in a separate parameter. Def then uses the *GreqlQueryExtraVars* class, which extends JGraLab's *GreqlQuery* class, to represent the query. The specialized class first "tricks" the GReQL parser by appending a dummy definition for the seemingly missing variables. When the query is evaluated, the original query is prefixed with a *using* statement so that they will be passed in. The actual variables are copied from the GReTL context to the query's environment, which the GReQL evaluator will use to assign the correct values.

Two further extensions have been created to complement GReTL's and GReQL's functionality, respectively: a modified transformation operation to allow the creation

[1] In the code, capability tuples are sometimes referred to as capability *vectors*. For all intents and purposes, these two notions refer to the same concept.

```
1  new CreateOrderedEdges(context, ContainsParameter.EC,
2            "from   o : topLevelContainer, "
3          + "       cvCombination : cvCombinations[o], "
4          + "       ip: o (-->{Orchestrates,ContainsAlternative})+
   ↪  &{AbstractServiceInstance} --> "
5          + "           &{InputPort @ isEmpty(thisVertex <--
   ↪  &{Flow,ExpansionPort})} "
6          + "report
   ↪  tup(o,ip,cvCombination,'IComposerContainsParameter'), "
7          + "       tup(o, cvCombination, 'ComposerSignature'), "
8          + "       tup(o, cvCombination, ip, 'IComposerInputParameter')
   ↪  "
9          + "end").execute();
```

Figure B.2: A GReTL operation for creating edges in a predictable order.

of *ordered* edges, and a Java-implemented GReQL function that implements the *mixed-radix* algorithm as defined by Knuth [2011, pp. 281f].

TGraphs are ordered, however, GReTL does not provide a way to generate vertices and edges in a predefined order. SCAffolder's target metamodel relies on the ordering of edge adjacencies in certain cases, though, for example to represent the Java statements inside a method body. In the source metamodel, a sequence of service instances in an orchestration is similarly modeled by the order of containment relationships. GReTL's standard *CreateVertex* and *CreateEdges* operations require the semantic expression to return its results as sets, only. Therefore, SCAffolder defines and implements an additional operation, *CreateOrderedEdges*, which modifies the regular *CreateEdges* operation to accept ordered sets as archetypes, and preserve the ordering when generating their images in the target model. Figure B.2 shows an example of using the operation to create edges that link Java method signature vertices to the parameters they take as arguments. Without a predictable order, it would be impossible to correctly match interfaces to each other, to their implementations, or to invocation statements.

The *mixed-radix generation* algorithm [Knuth, 2011, pp. 281f] creates all tuples of a given length $n$, where each element in the tuple has a potentially different range. More importantly, the cardinality of their ranges $c_i$ may differ, as well. If the cardinalities were all the same ($c$), the problem can be mapped to counting to the highest number representable with $n$ digits in a numeral system to the base of $c$. E.g., there are 1000 different triples in $A^3$, where $A = [0..9] \subset \mathbb{N}_0$, and they can easily be enumerated starting with $(0, 0, 0)$ and counting up to $(9, 9, 9)$. In numeral systems, the base is also called radix. Generalizing this concept to let every digit position have a different radix leads to mixed-radix systems.

As it turns out, this problem occurs in SENSEI in the form of *orchestration trails* (see Section 12.3). They are defined in terms of ordered service instance sets: a single trail is represented as a tuple of length $n$, where $n$ is the number of service instances. The

```
1 setGReQLVariable("cvCombinations", "from o : topLevelContainer "
2         + "reportMap o ->
  ↪  mixedRadix{CapabilityVector}(atomicServices[o]) "
3         + "end");
```

Figure B.3: Defining a map from orchestrations to their orchestration trails, using the custom *mixedRadix* GReQL function.

tuple chooses one capability tuple for each service instance, and since every service instance defines a varying number of required capability tuples, an orchestration trail is essentially a mixed-radix number, where the service instances a the digit positions, their capability tuples are the digits for that position.

SCAffolder needs to distinguish these orchestration trails, to generate code for each one. The generated composer contains a method for each trail, for example[2]. Accordingly, Java statements contained in those methods have to be created once for each trail.

The mixed-radix generation algorithm is implemented in class *MixedRadix*, extending the *Function* class of JGraLab's GReQL function library. It takes two arguments: the vertex type of elements to be chosen, and an (ordered) set of vertices, which have adjacencies to vertices of that type. For each vertex in the set, one "child" element of the provided type is chosen. The result returned from this method is a set of all possible combinations of choices.

Directly, the function is invoked only once within SCAffolder, namely to set the *cvCombinations* map variable, as shown in Figure B.3. The variable acts as a cache, to avoid having to redo the computation over and over again (it occurs more than sixty times in SCAffolder's transformation operations).

## B.2 Transformation-Embedded Composition Finding

A Prolog-based composition finder has been implemented by Meier [2014a] (see Section 14.3.1), but SCAffolder also implements are basic and less feature-complete version of this functionality, realized with GReTL transformations. This is one of the most complex parts of SCAffolder, as the paradigm of model-to-model transformations in general, and GReTL in particular, is not well-suited for solving constraint satisfaction problems, and is now considered deprecated.

[2]These methods assume the choices for all capability tuples are made a priori. There is also a single *execute* method being generated, in which the choice of capability tuples – and thereby implementing components to be invoked – is made dynamically. The transformation operations responsible for its generation make less use of the mixed-radix function, and could probably be refactored to do without it, completely.

```
1  from      cv1 : V{CapabilityVector}, cv2 : V{CapabilityVector}
2  with      cv1 <> cv2
3  reportMap tup(cv1,cv2) -> forall cap1 : cv1 --> &{Capability} @
4                            exists cap2 : cv2 --> &{Capability} @
5                                   cap1.name = cap2.name
6  end
```

(a) The helper variable **cvEquals** is filled with a lookup table to check whether two capability tuples are equal.

```
1  using     sInstance:
2  from      c : theElement(sInstance --> &{Service})
3               <--{LinksToService} <-- &{Component}
4  with      (c --> sInstance) or
5            (forall reqCV: (sInstance --> &{CapabilityVector}) @
6             exists provCV : (c --> &{AbstractServiceInstance}
7                                --> &{CapabilityVector}) @
8                    cvEquals[tup(reqCV,provCV)])
9  reportSet c
10 end
```

(b) Helper function **findImplementations** finds those components which are either connected by an edge to the provided service instance, or which provide all the required capabilities.

Figure B.4: The "simple" case of composition finding, with a one-to-one matching of service instances to components.

Nevertheless, this section will briefly sketch the implementation of SCAffolder, as it remains one of its features. It also goes to show that it should be easy to use an external composition finder in place of the embedded one, as the result of both implementations is basically a mapping of orchestrated service instances and required capability tuples, to component-implemented service instances and provided capability tuples. This mapping can simply be swapped out, and the embedded implementation can be deactivated.

The composition finding mechanism is implemented in a series of GReTL helper variables and functions that build on each other. The implementation language is therefore essentially GReQL. For easier readability, the code snippets shown in the figures in this section have been stripped of the surrounding GReTL Java API boilerplate.

To find appropriate components for each service instance in an orchestration, SCAffolder distinguishes two cases: either there is a single component that covers all required capability tuples, or multiple components must be combined to satisfy all of them.

Finding components of the former case is implemented in the helper function **findImplementations**, shown in Figure B.4(b). Given a service instance, its query ranges over all components that implement the service it conforms to (Lines 2 and 3). It then

```
1  using     sInstance:
2  let       implComponents := theElement(sInstance --> &{Service})
3                                <--{LinksToService} <-- &{Component},
4            reqCVectors := sInstance --> &{CapabilityVector} in
5  from      reqCVector : reqCVectors
6  reportSet (
7            from c : implComponents
8            with exists provCVector: (c --> &{AbstractServiceInstance}
9                                        --> &{CapabilityVector}) @
10                cvEquals[tup(reqCVector,provCVector)]
11           report c
12           end
13           )[0]
14 end
```

Figure B.5: GReQL expression implementing the **findComposedImplementations** helper function. It finds a set of components which combined provide all the capability tuples of the given service instance.

filters them by requiring one of two conditions: in the trivial case, there is an edge between the component and the provided service instance (Line 4). Generally, this will not be the case for service instances from an orchestration, but SCAffolder nevertheless supports this.

The second condition checks whether a component provides a capability tuple for each required one, that contains all the capabilities needed (Lines 5 through 8). It uses the helper variable **cvEquals** for this (Figure B.4(a)). This lookup table contains a boolean for each pair of capability tuples of the model. The name may be somewhat misleading, because it does not *exactly* check for what could be considered a canonical definition for capability tuple equality (both tuples have *exactly* the same elements). For example, it allows that the second capability tuple contains additional capabilities. Also, it does not check whether the capabilities are defined by the same service and capability class, but relies on their names being equal. However, due to the validity constraints that SENSEI models must satisfy, these cases do not occur. That means that for valid, well-formed SENSEI models, cvEquals *essentially* implements an equality test, as long as the first capability tuple is a required one, and the second is a provided one.

Finding compositions for the more general case of multiple components providing the required capabilities of a service instance together is implemented by **findComposedImplementations** depicted in Figure B.5. It takes a service instance as parameter (Line 1), and defines two additional variables in a *let* statement (Lines 2 through 4): *implComponents* is a set of all components implementing the same SENSEI service, without considering capabilities. The *reqCVectors* variable contains all required capability tuples of the given service instance. This is also the range of the query (Line 5): for each required capability tuple, the nested FWR expression (Lines 7 through 14) is

```
1 from   o : E{Orchestrates}
2 with   hasType{AbstractServiceInstance}(endVertex(o))
3 report o
4 end
```

(a) The helper function **orchestratingAtomicServiceInstances**.

```
1 from      si : orchestratedAtomicServiceInstances()
2 reportMap si -> findImplementations(si)
3 end
```

(b) The temporary helper variable **implTmp**.

```
1 from      si : orchestratedAtomicServiceInstances()
2 reportMap si -> ( isEmpty(implTmp[si]) ?
3                   findComposedImplementation(si) :
4                   implTmp[si] )
5 end
```

(c) The helper variable **implementations**.

Figure B.6: Helper variables and functions tying the basic composition finding together.

evaluated. It ranges over the implementing components, and filters out those that do not provide a capability tuple that equals the required one currently inspected. From the resulting set of candidate components, the first one is chosen (Line 15). In the end, this yields a set of components guaranteed to contain at least one component for each required capability tuple providing it.

Since for each required capability tuple, the first component satisfying it is chosen, the overall selection of components will be arbitrary, i.e. there are no other concerns being optimized. For example, it might make sense to minimize the number of total components, which is possible if some components provide more than one of the required capability tuples. Another important aspect is the compatibility with components matched to other service instances, if they need to exchange data. SCAffolder's basic implementation does not take such advanced considerations into account. The *CompositionFinder* by Meier [2014a] does (see Section 14.3.1), as its constraint logic programming paradigm is much better suited to express such optimization goals.

The composition finding functionality is tied together by the **implementations** variable (Figure B.6(c)), whose defining GReQL query relies on two further helpers, the function **orchestratingAtomicServiceInstances**, and the temporary variable **implTmp** (Figure B.6(a) and Figure B.6(b)). The variable **implementations** is filled with a map from orchestrated service instances to the components implementing them. It defaults to the simple case of one-to-one matching implemented in *findImplementations* (via *implTmp*). If this fails, *findComposedImplementations* is invoked instead.

There is another version of the *implementations* helper variable called **implementa-**

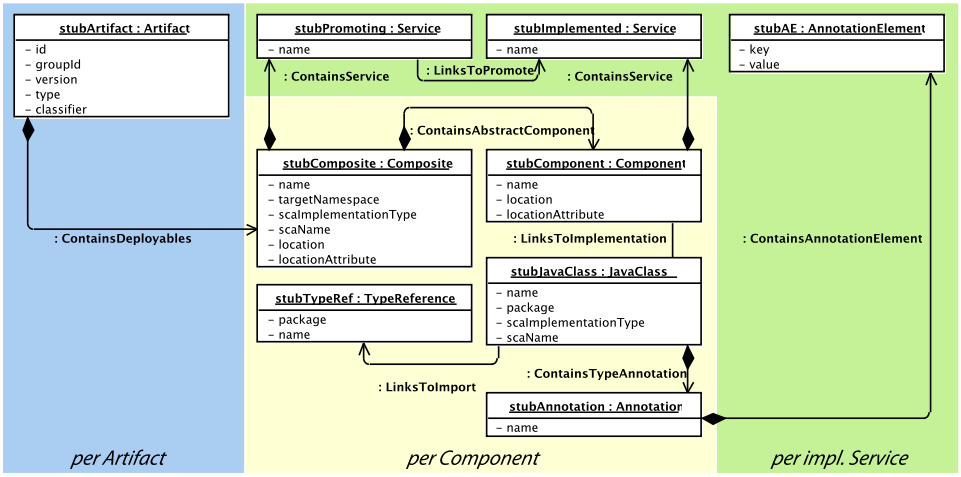



Figure B.7: Model elements created for each *Component* instance in the SENSEI model provided as SCAffolder's input.

**tionsMap**. It is essentially identical, except that its lookup table is organized by required capability tuples, instead of just service instances, which in many cases is more practical. It is used by the *providedCapabilityVector* function which was depicted earlier in Figure B.1. It is an essential element of many transformation operations in SCAffolder, allowing to navigate from service instances and their required capability tuples to the corresponding capability tuples provided by implementing components. As a kind of facade to the embedded composition finding functionality, its implementation could be changed to rely on the results of an external composition finder, instead. Another option would be to manifest the mapping to implementing components by adding edges into the source model. This could be realized as an additional preprocessing step, and because of how *findImplementations* is implemented (Figure B.4), no further changes to the transformation stage would be necessary.

This implementation of composition finding is, of course, fairly limited, especially when compared to the CompositionFinder presented briefly in Section 14.3.1, which offers a much more complete implementation. A major feature that is not realized within SCAffolder itself is the automatic inclusion of appropriate transformer components into data flows.

## B.3 Tool Stubs

Technically, all the transformation operations are always executed completely, independent of whether SCAffolder was invoked to create tool stubs, or to generate toolchains.

The source model is simply filtered in a way that causes operations only relevant for one of the two use cases to range over an empty set, and so essentially do nothing.

The first set of transformation operations is mainly aimed at the creation of tool adapter stubs, although some elements are also required during toolchain creation, e.g. referenced components. Figure B.7 depicts an object diagram visualizing the model elements created in the target model. In GReTL, a target model element is created per *archetype*, i.e. per element an operation's semantic expression returns.

Often, several target model elements are created based on a single source model type and its instances. Therefore, the objects shown in Figure B.7 are clustered into three groups: the five objects and three links to the right of the figure (on yellow background) are created based on the existance of *Component* instances. For example, for each instance of a *Component* in the SENSEI model, an instance of metaclass *Composite* in the target model is created.

The information represented by the target model elements being created will be used to generated a SCDL file and a Java class file stub. The *Composite* and contained *Component* instances, as well as each of their associated SCA *Service* instances, will mainly go into the former, while the *JavaClass*, *TypeReference*, *Annotation*, and *AnnotationElement* instances fill a skeleton Java class file. The *TypeReference* instance represents an import of the SCA service annotation from the standard's namespace. The *Annotation* instance represents one such annotation (*@Service*).

Since a component can potentially implement more than one service, the annotation may contain multiple *AnnotationElements*. Instances of this class are created once for each implemented service and each component (green background). Naturally, the same holds for containment relations linking annotations to their annotation elements.

The *Artifact* instances are actually only used during toolchain generation, to add corresponding dependency declarations into the toolchain's Maven build file. There is actually an identical class *Artifact* in the source model, and there can potentially be multiple artifacts required for a particular composite (blue background). This information is simply copied over from source into target metamodel.

## B.4 SCA Service Interfaces

One of the most important aspects of tool stubs are still missing, namely the (SCA) service interfaces. However, toolchains require the generation of these interfaces, as well, as types of their *referenced* SCA services. For tool stubs, interfaces need to be generated for each implemented SENSEI service instance, and each of its provided capability tuples. For toolchains, the necessary interfaces are determined by the service instances used in orchestrations, and their required capability tuples. Therefore, the operations to create SCA service instances all distinguish these two cases, explicitly. If the input model contains one or more orchestrations, interfaces needed for the corresponding toolchains are generated. SENSEI models declaring components and their implemented

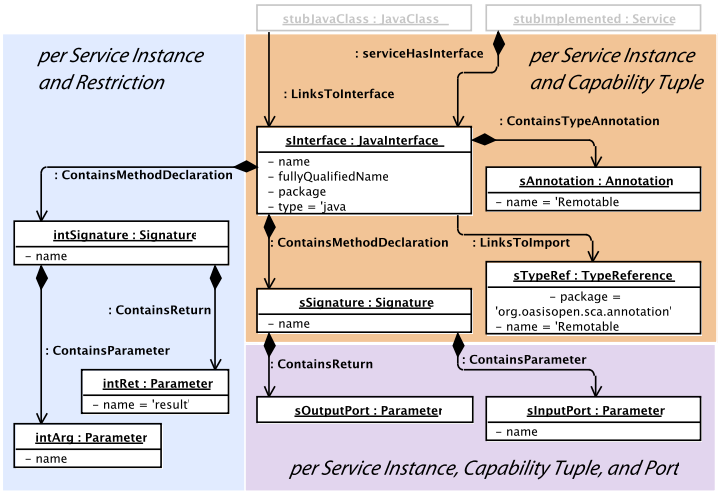



Figure B.8: Target model elements created for each *ServiceInstance* and *CapabilityTuple* instance in the source model.

services, which are used for tool stub generation, are not expected to contain orchestrations, and if they do, SCAffolder will filter them out during preprocessing.

Figure B.8 depicts the objects created in the target model to represent interfaces. For each service instance, and each capability tuple it defines (orange background), a *JavaInterface* is instantiated. There will also be a single method *Signature* in each such interface, which toolchains will call to invoke the corresponding service, and components implement. Having a separate interface for each service instance and each capability tuple allows for greatest flexibility when implementing tool adapters, as the implementation of different capabilities can be spread over different Java classes, or grouped together, however tool developers see fit. If all capability tuples of a service instance where mapped to a single Java interface with multiple methods, a single class would have to implement all of them. This is what SCAffolder will initially produce, i.e. all interfaces of the services implemented by a SENSEI component will be linked to the previously created *JavaClass* instance (depicted in gray) via a *LinksToInterface* edge. This can be freely changed, however – as opposed to the actual Java interface files, the class file will not be overridden by SCAffolder if it already exists.

By default, SCAffolder also creates *Remotable* annotations and corresponding import statements for each Java interface, meaning the associated SCA services can be invoked across Java virtual machine boundaries, in a potentially distributed SCA domain. Since this may not be appropriate for all use cases, it can be turned off.

Each method signature declared in a Java interface takes arguments, which are represented by *Parameter* instances, and created based on the source model's *InputPort*

```
new CreateVertices(context, TypeReference.VC,
    "from d : union(from d1: V{DataDefinition} reportSet d1.format
    ↪ end, "
    + "             set('byte', 'char', 'short', 'int', 'long',
    ↪ 'float',"
    + "                 'double', 'boolean', 'java.lang.String')) "
    + "reportSet d, 'UniqueTypes' "
    + "end").execute();
```

Figure B.9: Example of a transformation operation used in creating unique types.

instances referenced from the corresponding service instance (purple background). Accordingly, *OutputPort* instances are used to create *Parameter* instances for returned results. In Java, methods only return a single value or reference. SCAffolder generates a simple compound data type (i.e. a *struct*-like Java class) in case there are multiple output ports, and uses it as return type in the signature of corresponding methods.

Another set of method signatures is created and associated with each interface, one for each *ServiceInstance*, *CapabilityTuple*, and *Restriction* instance, defined on each of the tuples *Capability* instances. The objects created in the target model are shown at the left side of Figure B.8 (blue background). The corresponding transformation operations are implemented in Java class *Introspection.java* of SCAffolder.

These method signatures take exactly one argument, and return a boolean. They provide tool developers with the ability to implement code that checks input data *before* the actual service functionality is invoked, and report whether the component will be able to handle this kind of data. Toolchains generated by SCAffolder use this to dynamically decide for one of multiple components that implement (different capability tuples of) a particular service instance.

## B.5 Types

SCAffolder next establishes Java *types* to associate the previously created method parameters with, but also to use in variable declarations for the composer implementation (Section B.7). In SENSEI, the concrete implementation type is not chosen on the service or service orchestration level. Rather, the abstract data structures of SENSEI services are bound by tool developers to concrete types using *data definitions*. It is possible to assign different sets of types for each provided capability tuple of implemented services.

Since each type exists only once, it does not make sense to simply create a *TypeReference* instance[3] for each *DataDefinition*. For commonly used types, such as *String*, for

[3] *TypeReference* is the metaclass that represents Java types in the target metamodel. In the metamodel, it is derived from the abstract metaclass *Type*, just like *JavaInterface* and *JavaClass* (see Figure 14.6). While the latter represent the actual type definition, *TypeReference* is used to point to an imported type defined elsewhere. This distinction is important, because code files are generated for classes and interfaces, but

```
1 using provCV,port:
2   theElement(
3     intersection( provCV <-- &{DataDefinition},
4       intersection(
5         theElement( provCV <--^2 &{Component} ) --> &{DataDefinition},
6         port --> <-- &{DataDefinition} )))
```

Figure B.10: A GReQL-based helper function to retrieve *DataDefinitions*.

example, this would create multiple, identical instances. While this is not necessarily a problem, it would make matching and comparing types of parameters unnecessary complicated. Therefore, all types referenced by all the *DataDefinition* instances in the source model are compiled into a single set. SCAffolder currently only uses the *format* attribute, expecting this to be the fully qualified name of a Java type. The resulting set of type names is combined (by union) with a set containing the fully qualified names of all of Java's primitive types, and *String*, ensuring that these types can always be used in transformation operations, independent of the input model. An example transformation operation that creates the *TypeReference* vertices is shown in Figure B.9.

In addition, parameters of SENSEI services have a boolean *isCollection* attribute to indicate they represent a container (set, stream, array, etc.) whose elements are all of the kind modeled by the associated *DataStructure* instance. SCAffolder uses Java's *Collection* interface to implement this. For each type created, a *TypeReference* instance for a corresponding collection is created, which links to the "non-collection" type as its *type parameter* (see Figure 14.6). This association represents Java's *Generics*.

All parameters get associated to a type created like this. The creation of the corresponding *LinksToType* edges has been omitted from all the object diagrams for greater clarity. Basically, for each *Parameter* instance a *LinksToType* edge is also created.

Because of the decoupled nature of SENSEI's services and components, it is not that trivial to determine the correct types. Therefore, there are several variants of a helper function that retrieves them. Figure B.10 depicts the GReQL implementation of one of them, returning the type given a provided capability tuple and a port.

The query consists of several nested GReQL functions. Beginning from the inside (Line 5), all *DataDefinition* instances of the component that defines the given capability tuple are retrieved. The query navigates backwards from the capability tuple, following two edges to a single *Component* instance. The GReQL function *theElement* is used to retrieve that single element from the set that results from the path expression. Then, all edges leading to *DataDefinition* instances are followed.

In Line 6, all *DataDefinition* instances associated to the service parameter corresponding to the given port are retrieved (the *Parameter* instance is not explicitly mentioned in the path expression, but this is the only possibility to arrive at a *DataDefini-*

not for type references.

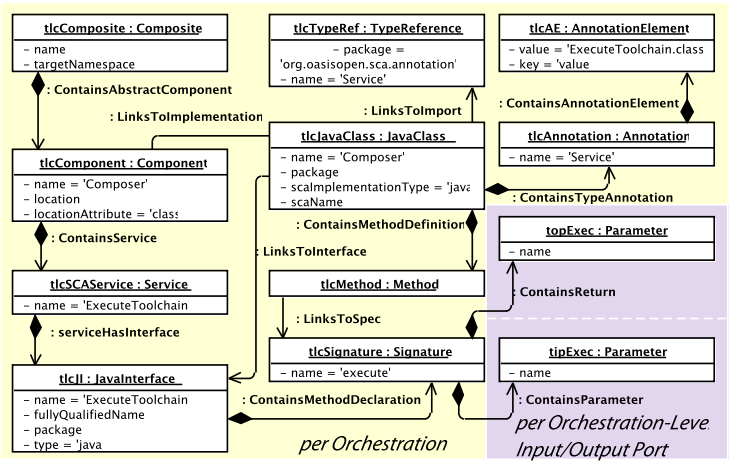



Figure B.11: Model elements created for each orchestration, representing the corresponding composer's structure.

*tion*). Because of the indirection via a service parameter, this will potentially include *DataDefinition* instances not defined by the same component. Therefore, the *intersection* of both sets is constructed (Line 4), leaving only those elements that are associated with the right component *and* the given port.

This set still includes *DataDefinition* instances for different capability tuples. To retrieve only the one element associated with the given capability tuple, the set is intersected again with the set of all *DataDefinition* instances of that tuple (Line 3), leaving only the element that refers to the right port *and* the right capability tuple. Another application of *theElement* drops the set, and returns its only contained element.

## B.6 Composer Structure

For Tool stub generation, the target model is complete after these steps. The remaining transformation operations are dedicated to creating *Composers*, i.e. the SCA components, with all the required software artifacts, that coordinate the processes defined by SENSEI orchestrations. The model elements to be created for the basic structure of composers are mostly the same as for tool stubs, only these are created for each orchestration, as opposed to components.

An overview is given in Figure B.11. Created are a *Composite*, a *Component*, a *JavaClass*, an *Annotation*, and a *TypeReference* instance, just like for tool stubs. The creation of an SCA *Service*, an *AnnotationElement*, and a *JavaInterface* is different, because only one per orchestration is created, just as with the other elements, whereas

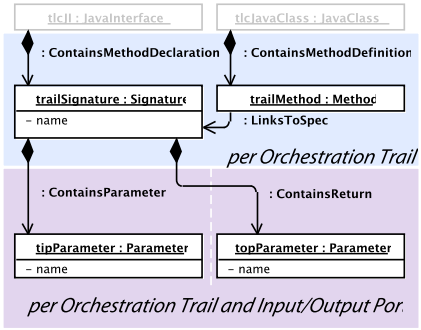



Figure B.12: Model elements created for each orchestration trail, representing Java methods to invoke a specific trail.

for tool stubs, these instances are created once for every implemented service of the corresponding component.

The composer exposes a single SCA service, called *ExecuteToolchain*. The associated Java interface declares a method (*Signature*) called *execute*, which the Java class implements. Input and output parameters for this method are created based on the ports defined on the orchestration itself. This method will dynamically choose appropriate components for service invocations, based on the kind of data provided as input at runtime. Because of this generic nature, it cannot make any assumptions regarding the types of its input and output parameters, which is why all parameters are of type *Object*, the super-type of all other types in Java.

A separate set of methods is created per orchestration trail. In each of these methods, a particular orchestration trail is hard-wired. The corresponding model elements that are created for this are shown in Figure B.12. The methods all have the same number of input and output parameters, but their types may differ, due to different data definitions bound to the provided capabilities.

Composers make use of some SCA concepts that tool stubs do not need. The corresponding target model elements that are created for this are depicted in Figure B.13. For each orchestrated service instance and their required capability tuples, an SCA *Reference* is created, contained in the composer's SCA component. Inside the composite, each of these references is connected to an appropriate SCA service using a *Wire*. The target model elements representing the SCA services have previously been created by transformation operations that are active for both tool stub and toolchain generation (see Figure B.7).

In the Java implementation class, a field for each of the references is needed. They get represented in the target model by *Parameter* instances, and carry the *@Reference Annotation*. This annotation is defined in the OASIS SCA namespace; the required import is represented by a *TypeReference* (which is only required once, of course). The

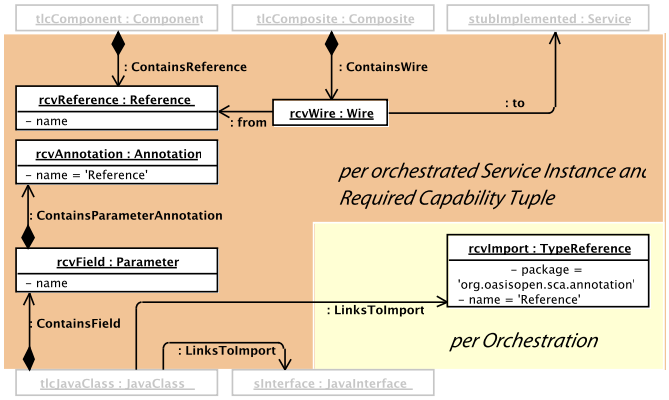



Figure B.13: Model elements created for each orchestration, representing the corresponding composer's structure.

types of the fields that reference SCA services are defined by the service interfaces created earlier (Section B.4). For each of them, an import is created, as well, represented by *LinksToImport* edges between the composer's Java implementation class and the Java service interfaces (recall from Figure 14.6 that *JavaInterface* extends *Type* in the target metamodel).

## B.7 Composer Implementation

The last step in the model-to-model transformation stage is to fill the method bodies of composers with the coordination logic that realizes the processes defined by orchestrations in the input SENSEI model. Basically, there are two sets of transformation operations: One creates the statements for all methods that implement specific orchestration trails. The other creates the statements for the more dynamic implementation in the *execute* method. Since the latter can be thought of as a more generic extension of the former, only the creation of statements for the *execute* method will be discussed in this section.

The first transformation operations deal with inputs and outputs of the service orchestrations themselves. At the beginning of execution, every corresponding composer will have to accept the inputs, and at the end, the results have to be returned. Therefore, *CopyStatement* instances are created for all *data delegations* in an orchestration, which connect orchestration input ports with input ports of nested service instances, or the output ports of nested service instances with output ports of the surrounding orchestration. Accordingly, the *CopyStatement* metaclass represents the value of a source parameter being assigned to a target parameter. The *CopyStatement* vertices, and the

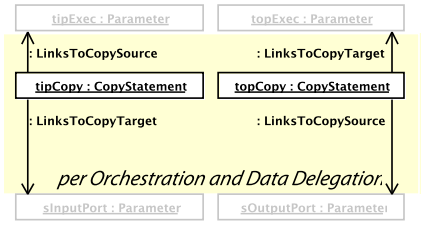



Figure B.14: Model elements created for each orchestration, representing the corresponding composer's structure.

edges created to interconnect parameters, which have been created by previous transformation instances, are depicted in Figure B.14

Next, there are transformation operations associated with each type of service instance and control flow construct in the orchestration layer of the SENSEI metamodel:

- *map* operators (*ForEachServiceInstance*) are transformed into *ForEachStatement* instances in the target metamodel, which represent Java for loops.
- *concurrencies* (*ConcurrentServiceInstance*) are transformed into *ConcurrentStatement* instances in the target metamodel, to represent threads in Java.
- *orchestrations* (*ServiceOrchestration*) are transformed into *BlockStatement* instances in the target metamodel, representing Java blocks, e.g. the bodies of for loops, and separate threads.
- *service instances* (*ServiceInstance*) are transformed into *InvokeAndReturnStatement* instances, representing the corresponding method invocation on the appropriate SCA reference field, and the subsequent assignment of the returned result to a variable in Java. For the *execute* method, all statements created for to the same service instance (but representing different capability tuples) are wrapped in a *SwitchInvokeAndReturn* statement, which will be turned into if-then-else statements that select the right SCA service to invoke dynamically, based on input data introspection.

SCAffolder does not currently support alternatives (conditional branches). Thanks to SENSEI's capability mechanism, high-level decisions can often be modeled declaratively, as was the case for all the applications SCAffolder has been used in so far (see Part V). The simplicity and conciseness of this approach is illustrated by the service instance shown in Figure 11.3 (page 183), when compared with the semantically equivalent service orchestration using an alternative, depicted in Figure 11.8 (page 190).

After having taken care of control flow, SCAffolder creates *CopyStatement* instances for all the data flows present in an orchestration. During code generation, this will be turned into Java code that stores results returned from SCA service invocations in a central map, using the names of input ports targeted by data flows as keys in the map.

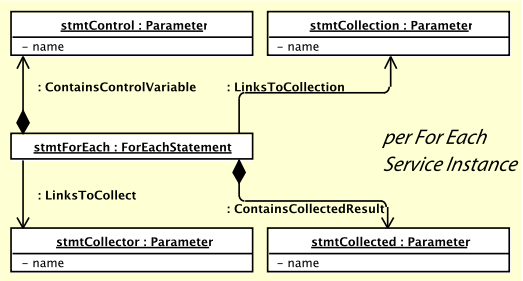



Figure B.15: Model elements created for each *map* (for-each-loop) control flow construct.

Finally, all model elements representing statements are linked to the composer methods they are contained in, ensuring the right order using the custom *CreateOrderedEdges* transformation operation.

The creation of statements for the different control flow constructs is mostly straightforward. The most complex transformations are required for the map operator, because of its additional ports. Therefore, the transformation of this control flow construct serves as an example.

The elements created in the target model are depicted in Figure B.15. A *ForEachStatement* has two pairs of parameters. On the input side, the *collection* to iterate over, and the *control variable* to represent the current element. On the output side, the *collector*, representing a variable to accept the result of a single iteration, and the *collected result*, a container to which all individual results are added. Both the input collection and the collector would not need a new parameter – instead, the transformation operations could link to existing parameters corresponding to an output port feeding the for loop, and an output port of a service instance nested in the loop, respectively. However, creating separate parameters here makes it easier later to handle all ports uniformly when transforming data flows.

As evident from the simplicity of this example, the creation of model elements to represent statements implementing composer methods leaves more details to be filled during code generation. This was a conscious decision to keep the Java-part of the target metamodel simple. This means that the target metamodel cannot represent arbitrary Java programs, and also that the Velocity templates used for code generation are very specific to SCAffolder. The template-based approach to code generation makes it easy to include a lot of boilerplate code that never changes, and only mix in those parts dynamically that depend on the target metamodel, instead of writing out the whole code “by hand”.

# References

For the reader's convenience, and to address different preferences and reading habits, the references referred to in this thesis are listed twice in the following: first by order of first appearance in the text, and second (starting on page 431) sorted by author names, publication year, and title of the work.

## References by Order of First Appearance


Wasserman, Anthony I. [1990]. "Tool Integration in Software Engineering Environments". In: *Software Engineering Environments. Proceedings of the International Workshop on Environments*. Lecture Notes in Computer Science 467. Ed. by Fred Long, pp. 137–149 [cit. on pp. x, 32, 39, 41, 44, 45, 48–51].

Brown, Alan W. and John A. McDermid [1992]. "Learning from IPSE's mistakes". In: *IEEE Software* 9.2, pp. 23–28 [cit. on pp. x, 41, 45, 48, 51–53, 64–66, 71].

Karsai, Gabor, Andras Lang, and Sandeep Neema [2005]. "Design patterns for open tool integration". In: *Software & System Modeling* 4.2, pp. 157–170 [cit. on pp. x, 48, 53–55, 76, 167].

Yang, Yun and Jun Han [1996]. "Classification of and Experimentation on Tool Interfacing in Software Development Environments". In: *Proceedings of the 3rd Asia-Pacific Software Engineering Conference*. IEEE, Los Alamitos, pp. 56–65 [cit. on pp. x, 46, 48, 55–58].

Fuggetta, Alfonso [1993]. "A Classification of CASE Technology". In: *Computer* 26.12, pp. 25–38 [cit. on pp. x, 48, 58–61, 68–70, 72, 90].

Naur, Peter and Brian Randell [1968]. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. Conference Report. NATO Scientific Affairs Division, Brussels [cit. on pp. 3, 39].

McIlroy, Malcolm Douglas [1968]. "Mass-Produced Software Components". In: *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. Ed. by Peter Naur and Brian Randell, pp. 138–155 [cit. on pp. 3, 99].

Dahl, Ole-Johan [2004]. "The Birth of Object Orientation: the Simula Languages". In: *From Object-Orientation to Formal Methods*. Ed. by Olaf Owe, Stein Krogdahl, and Tom Lyche. Lecture Notes in Computer Science 2635. Springer, Berlin, Heidelberg, pp. 15–25 [cit. on pp. 3, 99].

Kay, Alan C. [1993]. "The early history of Smalltalk". In: *ACM SIGPLAN Notices* 28.3, pp. 69–95 [cit. on pp. 3, 99].

Szyperski, Clemens [1997]. *Component software: beyond object-oriented programming*. ACM Press Books. Addison Wesley, Boston [cit. on pp. 3, 99, 100, 102, 103].

Erl, Thomas [2005]. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall, Upper Saddle River [cit. on pp. 3, 112, 115, 116, 118, 122–125].

Manes, Anne Thomas. [2003]. *Web Services: A Manager's Guide*. Addison-Wesley, Boston [cit. on p. 3].

Gorton, Ian, David Thurman, and Judi Thomson [2003]. "Next generation application integration: challenges and new approaches". In: *Proceedings of the 27th Annual International Computer Software and Applications Conference*. IEEE, Los Alamitos, pp. 576–581 [cit. on p. 3].

Lehman, Meir M. [1980]. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9, pp. 1060–1076 [cit. on pp. 3, 333].

Lehman, Meir M. [1996]. "Laws of software evolution revisited". In: *Proceedings of the 5th European Workshop on Software Process Technology*. Lecture Notes in Computer Science 1149. Springer, Berlin, Heidelberg, pp. 108–124 [cit. on pp. 3, 308].

Brodie, Michael L. and Michael Stonebraker [1995]. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco [cit. on p. 3].

Lientz, Bennet P. and E. Burton Swanson [1980]. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, Boston [cit. on p. 3].

Seacord, Robert C., Daniel Plakosh, and Grace A. Lewis [2003]. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, Boston [cit. on p. 3].

Broy, Manfred [2018]. "Yesterday, Today, and Tomorrow: 50 Years of Software Engineering". In: *IEEE Software* 35.5, pp. 38–43 [cit. on p. 3].

Sanner, Michael F. [1999]. "Python: a programming language for software integration and development". In: *Journal of molecular graphics & modelling* 17.1, pp. 57–61 [cit. on p. 4].

Land, Rikard and Ivica Crnkovic [2004]. "Existing Approaches to Software Integration - and a Challenge for the Future". In: *Proceedings of the 4th Conference on Software Engineering Research and Practice in Sweden*. Mälardalen University, Västerås. URL: `http://www.es.mdh.se/publications/642-Existing_`

Approaches_to_Software_Integration___and_a_Challenge_for_the_Future [cit. on p. 4].

Beck, Kent. [2004]. *Extreme Programming Explained: Embrace Change*. 2nd ed. Addison-Wesley, Boston [cit. on p. 4].

Schwaber, Ken and Mike Beedle [2002]. *Agile software development with Scrum*. Prentice Hall, Upper Saddle River [cit. on p. 4].

Boehm, Barry [2002]. "Get ready for agile methods, with care". In: *Computer* 35.1, pp. 64–69 [cit. on p. 4].

Kruchten, Philippe [2010]. "Software Architecture and Agile Software Development—A Clash of Two Cultures?" In: *Proceedings of the 32nd International Conference on Software Engineering, Volume 2*. ACM, New York, p. 497 [cit. on p. 4].

Knodel, Jens and Matthias Naab [2014]. "Mitigating the Risk of Software Change in Practice". In: *Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Ed. by Serge Demeyer, David Binkley, and Filippo Ricca. IEEE, Antwerp, pp. 2–17 [cit. on pp. 4, 19].

Müller, Hausi A., Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong [2000]. "Reverse engineering: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM, New York, pp. 47–60 [cit. on pp. 4, 5, 8].

Sim, Susan Elliott [2000]. "Next Generation Data Interchange: Tool-to-Tool Application Program Interface". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 278–280 [cit. on pp. 4, 5, 7, 52, 93].

Jin, Dean and James R. Cordy [2005a]. "Ontology-based software analysis and reengineering tool integration: the OASIS service-sharing methodology". In: *Proceedings of the 21st International Conference on Software Maintenance*. IEEE, Los Alamitos, pp. 613–616 [cit. on pp. 4, 5, 8, 65, 75].

Ghezzi, Giacomo and Harald C. Gall [2013]. "A framework for semi-automated software evolution analysis composition". In: *Automated Software Engineering* 20.3, pp. 463–496 [cit. on pp. 4, 6, 8, 78].

Borchers, Jens [1996]. "Reengineering-Factory — Erfolgsmechanismen großer Reengineering-Maßnahmen". In: *Softwarewartung und Reengineering*. Ed. by Franz Lehner. Information Engineering und IV-Controlling. Deutscher Universitätsverlag, Wiesbaden, pp. 19–29 [cit. on pp. 4, 5, 8].

Bergey, John K., Scott R. Tilley, Steven Woods, Dennis B. Smith, and Nelson W. Weiderman [1999]. *Why reengineering projects fail*. Technical Report. Software Engineering Institute, Carnegie Mellon University, Pittsburgh [cit. on p. 5].

Pike, Rob and Brian W. Kernighan [1984]. "The UNIX System: Program Design in the UNIX Environment". In: *AT&T Bell Laboratories Technical Journal* 63.8, pp. 1595–1605 [cit. on pp. 5, 70].

Holt, Richard C., Andreas Winter, and Andy Schürr [2000]. "GXL: toward a standard exchange format". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 162–171 [cit. on pp. 5, 201].

Mens, Tom, Michel Wermelinger, Serge Demeyer, Robert Hirschfeld, Stéphane Ducasse, and M Jazayeri [2005]. "Challenges in software evolution". In: *Proceedings of the 8th International Workshop on Principles of Software Evolution*. Ed. by Motoshi Saeki, Gerardo Canfora, and Shuichiro Yamamoto. IEEE, Los Alamitos, pp. 13–22 [cit. on pp. 5, 8].

Sneed, Harry M., Ellen Wolf, and Heidi Heilmann [2010]. *Softwaremigration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. Dpunkt, Heidelberg [cit. on pp. 5, 7].

Rajlich, Václav [2014]. "Software evolution and maintenance". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM, New York, pp. 133–144 [cit. on p. 6].

Newman, Sam [2015]. *Building Microservices*. O'Reilly, Sebastopol [cit. on pp. 6, 114, 118, 122, 125, 126].

Jamshidi, Pooyan, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov [2018]. "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3, pp. 24–35 [cit. on p. 6].

Cerny, Tomas, Michael Jeffry Donahoo, and Michal Trnka [2018]. "Contextual Understanding of Microservice Architecture: Current and Future Directions". In: *ACM SIGAPP Applied Computing Review* 17.4, pp. 29–45 [cit. on p. 6].

Atzori, Luigi, Antonio Iera, and Giacomo Morabito [2010]. "The Internet of Things: A survey". In: *Computer Networks* 54.15, pp. 2787–2805 [cit. on p. 6].

Lee, Jay, Behrad Bagheri, and Hung-An Kao [2015]. "A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems". In: *Manufacturing Letters* 3, pp. 18–23 [cit. on p. 6].

Ebert, Christof, Gerd Hoefner, and V. S. Mani [2015]. "What Next? Advances in Software- Driven Industries". In: *IEEE Software* 32.1, pp. 22–28 [cit. on p. 6].

Zimmermann, Olaf [2017]. "Microservices Tenets: Agile Approach to Service Development and Deployment". In: *Proceedings of the 10th Advanced Summer School on Service-Oriented Computing*. Computer Science - Research and Development 32.3-4. Springer, Berlin, Heidelberg, pp. 301–310 [cit. on p. 6].

Fuhr, Andreas, Andreas Winter, Uwe Erdmenger, Tassilo Horn, Uwe Kaiser, Volker Riediger, and Werner Teppe [2012]. "Model-Driven Software Migration - Process Model, Tool Support and Application". In: *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. Ed. by Anca Daniela Ionita, Martin Litoiu, and Grace Lewis. IGI Global, Hershey, pp. 153–184 [cit. on pp. 7, 15, 299].

Grieger, Marvin and Masud Fazal-Baqaie [2015]. "Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods". In: *Softwaretechnik-Trends* 35.2, pp. 41–42 [cit. on p. 7].

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas [2001]. *Manifesto for Agile Software Development*. URL: `http://agilemanifesto.org/` [visited on 01/15/2020] [cit. on p. 7].

Eden, Amnon H. and Tom Mens [2006]. "Measuring software flexibility". In: *IEE Proceedings - Software* 153.3, pp. 113–125 [cit. on pp. 7, 330].

ISO/IEC/IEEE 24765 [2017]. *Systems and software engineering – Vocabulary*. International Standard. International Organization for Standardization, Geneva [cit. on pp. 8, 44, 47].

ISO/IEC 9126-1 [2001]. *Software Engineering – Product Quality – Part 1: Quality Model*. International Standard. International Organisation for Standardization, Geneva [cit. on p. 8].

ISO/IEC 25010 [2011]. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. International Standard. International Organization for Standardization, Geneva [cit. on pp. 8, 12].

Jelschen, Jan and Andreas Winter [2011]. "Towards a Catalogue of Software Evolution Services". In: *Softwaretechnik-Trends* 31.2, pp. 36–37 [cit. on pp. 9, 339].

Jelschen, Jan and Andreas Winter [2012]. "A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems". In: *Softwaretechnik-Trends* 32.2, pp. 67–68 [cit. on pp. 9, 339].

Jelschen, Jan [2013]. "Discovery and Description of Software Evolution Services". In: *Softwaretechnik-Trends* 33.2, pp. 59–60 [cit. on pp. 9, 81, 117, 157, 164, 339].

Jelschen, Jan, Johannes Meier, Marie-Christin Ostendorp, and Andreas Winter [2013]. "A Description Model for Software Evolution Services". In: *1er Congreso Nacional de Ingeniería Informática / Sistemas de Información*. RIISIC, Cordoba. URL: `http://www.conaiisi.unsl.edu.ar/ingles/papers.php` [cit. on pp. 9, 26, 177, 339].

Jelschen, Jan and Andreas Winter [2014]. "Modeling Service Capabilities for Software Evolution Tool Integration". In: *Softwaretechnik-Trends* 34.2, pp. 91–92 [cit. on pp. 9, 154, 339].

Jelschen, Jan [2014a]. "SENSEI: Software Evolution Service Integration". In: *Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Ed. by Serge Demeyer, David Binkley, and Filippo Ricca. IEEE, Antwerp, pp. 469–472 [cit. on pp. 9, 339].

Jelschen, Jan [2015]. "Service-Oriented Toolchains for Software Evolution". In: *Proceedings of the 9th International Symposium on the Maintenance and Evolution of*

*Service-Oriented and Cloud-Based Environments*. Ed. by Andreas Winter, Mike Smit, and Muhammad Ali Barbar. IEEE, Los Alamitos, pp. 51–58 [cit. on pp. 9, 339].

Jelschen, Jan, Johannes Meier, and Andreas Winter [2015]. "SENSEI Applied: An Auto-Generated Toolchain for Q-MIG". In: *Softwaretechnik-Trends* 35.2, pp. 39–40 [cit. on pp. 9, 339].

Jelschen, Jan, Christoph Alexander Küpker, Andreas Winter, Alexander Sandau, Benjamin Wagner vom Berg, and Jorge Marx Gómez [2016]. "Towards a Sustainable Software Architecture for the NEMo Mobility Platform". In: *Proceedings of the 30th International Conference on Environmental Informatics – Stability, Continuity, Innovation: Current trends and future perspectives based on 30 years of history*. Ed. by Volker Wohlgemuth, Frank Fuchs-Kittowski, and Jochen Wittmann. Berichte aus der Umweltinformatik. Shaker, Herzogenrath, pp. 41–48 [cit. on pp. 9, 318, 339, 341].

Meier, Johannes, Dilshodbek Kuryazov, Jan Jelschen, and Andreas Winter [2015]. "A Quality Control Center for Software Migration". In: *Softwaretechnik-Trends* 35.2, pp. 19–20 [cit. on p. 12].

Jelschen, Jan [2014b]. *The Q-MIG Data Exchange Format*. Project Report. Carl von Ossietzky University, Oldenburg [cit. on p. 15].

Ebert, Jürgen, Volker Riediger, and Andreas Winter [2008]. "Graph Technology in Reverse Engineering: The TGraph Approach." In: *Proceedings of the 10th Workshop Software Reengineering*. Ed. by Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter. Lecture Notes in Informatics P-126. Gesellschaft für Informatik, Bonn, pp. 67–81 [cit. on pp. 17, 193, 236, 293].

Boehm, Barry [2006]. "A view of 20th and 21st century software engineering". In: *Proceedings of the 28th International Conference on Software Engineering*. ACM, New York [cit. on pp. 18, 39–41].

Pandey, Gaurav [2014]. *Short Report on Clone Detection Tools*. Tech. rep. Carl von Ossietzky University, Oldenburg [cit. on pp. 18, 32].

Juergens, Elmar, Florian Deißenböck, and Benjamin Hummel [2009]. "CloneDetective - A workbench for clone detection research". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 603–606 [cit. on p. 18].

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue [2002]. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7, pp. 654–670 [cit. on p. 18].

Wettel, Richard and Radu Marinescu [2005]. "Archeology of code duplication: Recovering duplication chains from small duplication fragments". In: *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE. Los Alamitos, pp. 63–70 [cit. on pp. 18, 284, 285].

Schmidt, Douglas C. [1999]. "Why Software Reuse has Failed and How to Make it Work for You". In: *C++ Report* 11.1 [cit. on p. 20].

Jelschen, Jan, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter [2012]. "Towards applying reengineering services to energy-efficient applications". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 353–358 [cit. on p. 25].

*Q-MIG* [2015]. Software Engineering Group of Carl von Ossietzky University and pro et con Innovative Informatikanwendungen GmbH. URL: `http://se.uni-oldenburg.de/Q-MIG` [visited on 01/15/2020] [cit. on p. 25].

Bourque, Pierre and Richard E. Fairley, eds. [2014]. *Guide to the Software Engineering Body of Knowledge (Swebok(r)): Version 3.0*. IEEE, Los Alamitos [cit. on pp. 31, 101, 103].

Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts [1999]. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston [cit. on p. 31].

Roy, Chanchal K., James R. Cordy, and Rainer Koschke [2009]. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Science of Computer Programming* 74.7, pp. 470–495 [cit. on p. 31].

*CORBA* [2020]. Object Management Group. URL: `http://www.corba.org/` [visited on 01/15/2020] [cit. on pp. 34, 99, 118].

Booth, David, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard [2004]. *Web Services Architecture*. URL: `https://www.w3.org/TR/ws-arch/` [visited on 01/15/2020] [cit. on pp. 34, 115, 119].

Fielding, Roy T. and Richard N. Taylor [2002]. "Principled design of the modern Web architecture". In: *ACM Transactions on Internet Technology* 2.2, pp. 115–150 [cit. on pp. 34, 68, 90].

Parnas, David Lorge [1972]. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12, pp. 1053–1058 [cit. on p. 36].

Asplund, Fredrik and Martin Törngren [2015]. "The discourse on tool integration beyond technology, a literature survey". In: *Journal of Systems and Software* 106, pp. 117–131 [cit. on pp. 39, 49, 52, 64].

Buxton, John N. and Vic Stenning [1980]. *Requirements for Ada Programming Support Environments: "Stoneman"*. Tech. rep. ADA100404. US Department of Defense [cit. on pp. 39, 40].

Dijkstra, Edsger W. [1968]. "Go To Statement Considered Harmful". In: *Communications of the ACM* 11.3, pp. 147–148 [cit. on p. 39].

Dahl, Ole-Johan, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare [1972]. *Structured programming*. Academic Press, London [cit. on pp. 39, 99, 187].

Royce, Winston W. [1970]. "Managing the development of large software systems". In: *WESCON technical papers. Papers presented at the Western Electronic Show and Convention*. Los Angeles. IEEE, Los Alamitos, pp. 328–338 [cit. on p. 39].

Wirth, Niklaus [2008]. "A Brief History of Software Engineering". In: *IEEE Annals of the History of Computing* 30.3, pp. 32–39 [cit. on p. 39].

Endres, Albert [1996]. "A Synopsis of Software Engineering History: The Industrial Perspective". In: *Proceedings of the Dagstuhl Seminar 9635 on History of Software Engineering*. Ed. by Andreas Brennecke and Reinhard Keil-Slawik. Dagstuhl Seminar Reports 153. Leibniz-Zentrum für Informatik, Wadern, pp. 20–24 [cit. on pp. 40, 42].

Brown, Alan W. [1988]. "Integrated project support environments". In: *Information & Management* 15.3, pp. 125–134 [cit. on p. 40].

*Eclipse Modeling Project* [2020]. Eclipse Foundation. URL: `https://projects.eclipse.org/projects/modeling` [visited on 01/15/2020] [cit. on p. 40].

Bott, Frank, ed. [1989]. *Eclipse, an integrated project support environment*. Vol. 14. IEE Computing Series. Peter Peregrinus, Hitchin [cit. on p. 40].

Dowson, Mark [1987]. "ISTAR—an integrated project support environment". In: *ACM SIGPLAN Notices* 22.1, pp. 27–33 [cit. on p. 40].

Long, Fred and Edwin J. Morris [1993]. *An Overview of PCTE: A Basis for a Portable Common Tool Environment*. Tech. rep. CMU/SEI-93-TR-001. Software Engineering Institute, Pittsburgh [cit. on p. 40].

Parker, Burt [1992]. "Introducing EIA-CDIF: the CASE Data Interchange Format Standard". In: *Proceedings of the 2nd Symposium on Assessment of Quality Software Development Tools*. IEEE, Los Alamitos, pp. 74–82 [cit. on pp. 40, 65].

*XMI (XML Metadata Interchange)* [2015]. Object Management Group. URL: `http://www.omg.org/spec/XMI/` [visited on 01/15/2020] [cit. on pp. 40, 65].

Martin, Roger J. [1993]. *Reference Model for Frameworks of Software Engineering Environments*. Technical Report / Special Publication ECMA TR/55, NIST SP 500-211, European Computer Manufacturers Association / National Institute of Standards and Technology [cit. on pp. 40, 42, 89, 90].

Brown, Alan, David Carney, Patricia Oberndorf, and Marvin Zelkowitz [1993]. *Reference Model for Project Support Environments (Version 2.0)*. Tech. rep. CMU/SEI-93-TR-23, NIST SP 500-213. Software Engineering Institute, National Institute of Standards and Technology [cit. on pp. 40, 41, 43, 58, 91].

Chikofsky, Elliot J. [1988]. "Guest Editor's Introduction: Software Technology People Can Really Use". In: *IEEE Software* 5.2, pp. 8–10 [cit. on p. 41].

Wicks, Michael N. and Richard G. Dewar [2007]. "A new research agenda for tool integration". In: *Journal of Systems and Software* 80.9, pp. 1569–1585 [cit. on pp. 41, 46].

Sharon, David and Rodney Bell [1995]. "Tools that bind: Creating Integrated Environments". In: *IEEE Software* 12.2, pp. 76–85 [cit. on p. 41].

Schmidt, Douglas C. [2006]. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2, pp. 25–31 [cit. on pp. 42, 309].

ElShazly, Hassan and Varun Grover [1993]. "A Study on the Evaluation of CASE Technology". In: *Journal of Information Technology Management* IV.1, pp. 15–24 [cit. on p. 42].

Lending, Diane and Norman L. Chervany [1998]. "The use of CASE tools". In: *Proceedings of the ACM SIGCPR Conference on Computer Personnel Research*. Ed. by Fred Niederman and Ritu Agarwal. ACM, New York, pp. 49–58 [cit. on p. 42].

Ocampo, Camilo, Begoña Albizuri, and Pere Botella [1998]. "Is CASE Technology Still Alive?" In: *Actas de las III Jornadas de Ingeniería del Software*. Ed. by José Ambrosio Toval Álvarez and Joaquín Nicolás Ros. Diego Marín, Murcia, pp. 127–139 [cit. on p. 42].

Kemerer, Chris F. [1992]. "How the Learning Curve Affects CASE Tool Adoption". In: *IEEE Software* 9.3, pp. 23–28 [cit. on p. 42].

Chau, Patrick Y. K. [1996]. "An empirical investigation on factors affecting the acceptance of CASE by systems developers". In: *Information and Management* 30.6, pp. 269–280 [cit. on p. 42].

Kelly, Steven and Juha-Pekka Tolvanen [2008]. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Chichester [cit. on pp. 42, 134, 138, 140].

Stahl, Thomas, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen [2006]. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester [cit. on pp. 42, 135–138].

Brown, Alan W., Peter H. Feiler, and Kurt C. Wallnau [1992]. "Past and Future Models of CASE Integration". In: *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*. IEEE, Los Alamitos, pp. 36–45 [cit. on pp. 42, 43].

Booch, Grady, James Rumbaugh, and Ivar Jacobson [1999]. *The Unified Modeling Language User Guide*. 5th ed. Object Technology Series. Addison-Wesley, Boston [cit. on pp. 43, 102, 103].

Kleppe, Anneke, Jos Warmer, and Wim Bast [2003]. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston [cit. on pp. 43, 129, 133, 134, 137–139].

Gašević, Dragan, Dragan Djurić, and Vladan Devedžić [2006]. *Model Driven Architecture and Ontology Development*. Springer, Berlin, Heidelberg [cit. on p. 43].

Terry, B. and D. Logee [1990]. "Terminology for Software Engineering Environment (SEE) and Computer-Aided Software Engineering (CASE)". In: *ACM SIGSOFT Software Engineering Notes* 15.2, pp. 83–94 [cit. on p. 45].

Erdmenger, Uwe and Denis Uhlig [2011]. "Ein Translator für die COBOL-Java-Migration". In: *Softwaretechnik-Trends* 31.2 [cit. on pp. 45, 68].

Biehl, Matthias [2013]. "A Modeling Language for the Description and Development of Tool Chains for Embedded Systems". Doctoral Thesis. Royal Institute of Technology, Stockholm [cit. on pp. 45, 64, 78, 82, 84].

Thomas, Ian and Brian A. Nejmeh [1992]. "Definitions of tool integration for environments". In: *IEEE Software* 9.2, pp. 29–35 [cit. on pp. 45–48, 50, 52].

Brooks, Frederick P. [1987]. "No Silver Bullet: Essence and Accidents of Software Engineering". In: *Computer* 20.4, pp. 10–19 [cit. on pp. 46, 56].

Wegner, Peter [1996]. "Interoperability". In: *ACM Computing Surveys* 28.1, pp. 285–287 [cit. on pp. 46, 47].

Morris, Edwin, Linda Levine, Patrick R. Place, Daniel Plakosh, and B. Craig Meyers [2004]. *Systems of Systems Interoperability*. Tech. rep. CMU/SEI-2004-TR-004. Software Engineering Institute, Pittsburgh [cit. on p. 47].

Tolk, Andreas and James Muguira [2003]. "The Levels of Conceptual Interoperability Model". In: *Proceedings of the Fall Simulation Interoperability Workshop*. Curran Associates, Red Hook, pp. 53–62 [cit. on p. 47].

Gürdür, Didem, Fredrik Asplund, and Jad El-Khoury [2016]. "Measuring Tool Chain Interoperability in Cyber-physical Systems". In: *Proceedings of the 11th System of Systems Engineering Conference*. IEEE, Los Alamitos [cit. on p. 47].

Laws, Simon, Mark Combellack, Raymond Feng, Haleh Mahb, and Simon Nash [2011]. *Tuscany SCA in Action*. Manning, Shelter Island [cit. on pp. 48, 186, 240, 286, 294, 299].

*WSO2* [2020]. URL: `http://wso2.com` [visited on 01/15/2020] [cit. on pp. 48, 105, 256, 310].

Wasserman, Anthony I. [1996]. "Toward a discipline of software engineering". In: *IEEE Software* 13.6, pp. 23–31 [cit. on p. 49].

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal [1996]. *Pattern-Oriented Software Architecture: A System of Patterns*. Vol. 1. Wiley, Chichester [cit. on pp. 53, 88].

Meier, Johannes and Andreas Winter [2016]. "Towards Metamodel Integration Using Reference Metamodels". In: *Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. Ed. by Colin Atkinson, Erik Burger, Thomas Goldschmidt, and Ralf Reussner. Karlsruhe Reports in Informatics 2016.7. Karlsruher Institut für Technologie, pp. 19–22 [cit. on p. 54].

Burger, Erik, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe [2016]. "View-based model-driven software development with ModelJoin". In: *Software and Systems Modeling* 15.2, pp. 473–496 [cit. on p. 54].

Tunjic, Christian and Colin Atkinson [2015]. "Synchronization of Projective Views on a Single-Underlying-Model". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. Ed. by Uwe Aßmann, Colin Atkinson, Erik Burger, Thomas Goldschmidt, and Ralf Reussner. ACM, New York, pp. 55–58 [cit. on p. 54].

Josuttis, Nicolai M. [2007]. *SOA in Practice: The Art of Distributed System Design*. O'Reilly, Sebastopol [cit. on pp. 54, 109, 112–119, 124–126, 201].

Ghezzi, Giacomo [2012]. "SOFAS, Software Analysis as a Service. Improving and Rethinking Software Evolution Analysis". Dissertation. University of Zurich [cit. on pp. 64, 78].

Holt, Richard C., Andy Schürr, Susan Elliott Sim, and Andreas Winter [2006]. "GXL: A graph-based standard exchange format for reengineering". In: *Science of Computer Programming* 60.2, pp. 149–170 [cit. on p. 65].

Ducasse, Stéphane, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Gîrba [2011]. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Tech. rep. hal-00646884. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on pp. 65, 67].

Nierstrasz, Oscar [2012]. "Agile software assessment with Moose". In: *ACM SIGSOFT Software Engineering Notes* 37.3, pp. 1–5 [cit. on pp. 65, 67].

W3C OWL Working Group [2012]. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. URL: https://www.w3.org/TR/owl2-overview/ [visited on 01/15/2020] [cit. on p. 65].

Cyganiak, Richard, David Wood, and Markus Lanthaler [2014]. *RDF 1.1 Concepts and Abstract Syntax*. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ [visited on 01/15/2020] [cit. on p. 65].

*Open Services for Lifecycle Collaboration* [2020]. OSLC Community. URL: http://open-services.net/ [visited on 01/15/2020] [cit. on pp. 65, 85].

Lethbridge, Timothy C., Sander Tichelaar, and Erhard Ploedereder [2004]. "The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering". In: *Electronic Notes in Theoretical Computer Science* 94, pp. 7–18 [cit. on pp. 65, 66, 68, 70].

Würsch, Michael, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall [2012]. "SEON: a pyramid of ontologies for software evolution and its applications". In: *Computing* 94.11, pp. 857–885 [cit. on pp. 65, 67, 201].

Hesse, Wolfgang and Heinrich C. Mayr [2008]. "Modellierung in der Softwaretechnik: eine Bestandsaufnahme". In: *Informatik-Spektrum* 31.5, pp. 377–393 [cit. on pp. 65, 132, 133].

Tichelaar, Sander, Stéphane Ducasse, and Serge Demeyer [2000]. "FAMIX and XMI". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 296–299 [cit. on p. 67].

*SEON - Software Evolution ONtologies* [2016]. Software Evolution and Architecture Lab. URL: http://se-on.org [visited on 01/15/2020] [cit. on p. 67].

*Knowledge Discovery Metamodel* [2016]. Object Management Group. URL: http://www.omg.org/spec/KDM/ [visited on 01/15/2020] [cit. on p. 67].

*Meta Object Facility* [2013]. Object Management Group. URL: http://www.omg.org/spec/MOF/2.4.1/ [visited on 01/15/2020] [cit. on pp. 67, 135].

Pérez-Castillo, Ricardo, Ignacio García Rodríguez De Guzmán, and Mario Piattini [2011]. "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems". In: *Computer Standards & Interfaces* 33.6, pp. 519–532 [cit. on p. 67].

Electronic Industries Association [1994]. *CDIF Integrated Meta-model Foundation Subject Area*. Interim Standard EIA/IS-111. Electronic Industries Association, Arlington [cit. on p. 67].

Leitner, Andrea, Beate Herbst, and Roland Mathijssen [2016]. "Lessons Learned from Tool Integration with OSLC". In: *Proceedings of the 22nd International Conference Information and Software Technologies*. Communications in Computer and Information Science 639. Springer, Cham, pp. 242–254 [cit. on p. 68].

Kraft, Nicholas A. [2007]. "An Infrastructure to Support Interoperability in Reverse Engineering". PhD thesis. Clemson University [cit. on pp. 68, 71, 90].

Kienle, Holger M. and Hausi A. Müller [2008]. "The Rigi reverse engineering environment". In: *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques*. University of Bern. URL: `http://scg.unibe.ch/download/wasdett/wasdett2008-paper06.pdf` [cit. on p. 69].

Raza, Aoun, Gunther Vogel, and Erhard Plödereder [2006]. "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering". In: *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies*. Ed. by Luís Miguel Pinho and Michael González Harbour. Lecture Notes in Computer Science 4006. Springer, Berlin, Heidelberg, pp. 71–82 [cit. on p. 69].

Koschke, Rainer [2000]. "Atomic Architectural Component Recovery for Program Understanding and Evolution: Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method". Dissertation. University of Stuttgart [cit. on p. 69].

Czeranski, Jörg, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner [2000]. "Data exchange in Bauhaus". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 293–295 [cit. on p. 70].

Kazman, Rick and S. Jeromy Carrière [1999]. "Playing detective: Reconstructing software architecture from available evidence". In: *Automated Software Engineering* 6.2, pp. 107–138 [cit. on p. 70].

Chen, Yih-Farn, Michael Y. Nishimoto, and Chittoor V. Ramamoorthy [1990]. "The C information abstraction system". In: *IEEE Transactions on Software Engineering* 16.3, pp. 325–334 [cit. on p. 70].

Holt, Richard C., Michael W. Godfrey, and Andrew J. Malton [2003]. "The Build / Comprehend Pipelines". In: *Proceedings of the Second ASERC Workshop on Software Architecture*. Alberta Software Engineering Research Consortium. URL: `https://plg.uwaterloo.ca/%7B~%7Dmigod/papers/2003/aserc03.pdf` [cit. on p. 70].

*SWAG Tools* [2020]. SWAG. URL: `http://www.swag.uwaterloo.ca/tools.html` [visited on 01/15/2020] [cit. on p. 70].

Godfrey, Michael W. and Lijie Zou [2005]. "Using origin analysis to detect merging and splitting of source code entities". In: *IEEE Transactions on Software Engineering* 31.2, pp. 166–181 [cit. on p. 70].

Bevan, Jennifer, E. James Whitehead, Sunghun Kim, and Michael W. Godfrey [2005]. "Facilitating software evolution research with kenyon". In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, New York, pp. 177–186 [cit. on p. 70].

Storey, Margaret-Anne, Casey Best, and Jeff Michand [2001]. "Shrimp views: An interactive environment for exploring java programs". In: *Proceedings of the 9th International Workshop on Program Comprehension*. IEEE, Los Alamitos [cit. on p. 71].

Ebert, Jürgen, Bernt Kullbach, Volker Riediger, and Andreas Winter [2002]. "GUPRO - Generic understanding of programs: An overview". In: *Proceedings of the 1st International Conference on Graph Transformation*. Electronic Notes in Theoretical Computer Science 72.2. Elsevier, Amsterdam, pp. 59–68 [cit. on p. 71].

Kullbach, Bernt and Andreas Winter [1999]. "Querying as an enabling technology in software reengineering". In: *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 42–50 [cit. on p. 71].

Ferenc, Rudolf, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy [2002]. "Columbus – Reverse Engineering Tool and Schema for C++". In: *Proceedings of the 18th International Conference on Software Maintenance*. IEEE, Los Alamitos, pp. 172–181 [cit. on p. 71].

Deißenböck, Florian, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas Parareda, and Markus Pizka [2008]. "Tool Support for Continuous Quality Control". In: *IEEE Software* 25.5, pp. 60–67 [cit. on p. 72].

Deißenböck, Florian, Lars Heinemann, Benjamin Hummel, and Elmar Juergens [2010]. "Flexible architecture conformance assessment with ConQAT". In: *Proceedings of the 32nd International Conference on Software Engineering*. ACM, New York, pp. 247–250 [cit. on p. 72].

Heinemann, Lars, Benjamin Hummel, and Daniela Steidl [2014]. "Teamscale: software quality control in real-time". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, pp. 592–595 [cit. on p. 72].

Ducasse, Stéphane, Tudor Gîrba, and Oscar Nierstrasz [2005]. "Moose: an Agile Reengineering Environment". In: *ACM SIGSOFT Software Engineering Notes* 30.5, pp. 99–102 [cit. on p. 73].

Ducasse, Stéphane, Michele Lanza, and Sander Tichelaar [2000]. "MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems". In: *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*. Ed. by Ian Ferguson, Jonathan Gray, and Louise Scott, pp. 24–30 [cit. on p. 73].

*Moose* [2020]. Moose Community. URL: http://www.moosetechnology.org/ [visited on 01/15/2020] [cit. on p. 73].

Black, Andrew P., Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker [2018]. *Pharo by Example 5*. Square Bracket Associates [cit. on p. 73].

Lungu, Mircea, Michele Lanza, and Oscar Nierstrasz [2014]. "Evolutionary and collaborative software architecture recovery with Softwarenaut". In: *Science of Computer Programming* 79, pp. 204–223 [cit. on p. 73].

Alvaro, Alexandre, Daniel Lucrédio, Vinicius Cardoso Garcia, Antonio Francisco do Prado, Luis Carlos Trevelin, and Eduardo Santana de Almeida [2003]. "Orion-RE: a component-based software reengineering environment". In: *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 248–257 [cit. on p. 73].

Bruneliere, Hugo, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot [2010a]. "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, pp. 173–174 [cit. on p. 73].

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks [2008]. *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. The Eclipse Series. Addison-Wesley, Boston [cit. on pp. 73, 143, 211, 213].

Bruneliere, Hugo, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot [2014]. "Modisco: A model driven reverse engineering framework". In: *Information and Software Technology* 56.8, pp. 1012–1032 [cit. on p. 73].

Baxter, Ira D., Christopher Pidgeon, and Michael Mehlich [2004]. "DMS: Program transformations for practical scalable software evolution". In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 625–634 [cit. on pp. 73, 144].

*DMS Software Reengineering Toolkit* [2020]. Semantic Designs. URL: http://www.semdesigns.com/Products/DMS/DMSToolkit.html [visited on 01/15/2020] [cit. on p. 73].

Bergstra, Jan A. and Paul Klint [1998]. "The discrete time ToolBus — A software coordination architecture". In: *Science of Computer Programming* 31.2-3, pp. 205–229 [cit. on p. 75].

Jong, Hayco de and Paul Klint [2003]. "ToolBus: The Next Generation". In: *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Lecture Notes in Computer Science 2852. Springer, Berlin, Heidelberg, pp. 220–241 [cit. on p. 75].

Brand, Mark G. J. van den, Arie van Deursen, Jan Heering, Heyco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser [2001]. "The ASF+SDF

Meta-environment: A Component-Based Language Development Environment". In: *Proceedings of the 10th International Conference on Compiler Construction*. Ed. by Reinhard Wilhelm. Lecture Notes in Computer Science 2027. Springer, Berlin, Heidelberg, pp. 365–370 [cit. on pp. 75, 143].

Jin, Dean, James R. Cordy, and Thomas R. Dean [2003]. "Transparent reverse engineering tool integration using a conceptual transaction adapter". In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. March. IEEE, Los Alamitos, pp. 399–408 [cit. on p. 75].

Jin, Dean and James R. Cordy [2003]. "A Service Sharing Approach to Integrating Program Comprehension Tools". In: *Proceedings of the Workshop on Tool Integration in System Development*. Ed. by Heiko Dörr and Andy Schürr. Darmstadt University of Technology, pp. 73–78. URL: https://web.archive.org/web/20070629182619/https://www.es.tu-darmstadt.de/english/events/tis/ [cit. on p. 75].

Jin, Dean and James R. Cordy [2005b]. "Factbase Filtering Issues in an Ontology-Based Reverse Engineering Tool Integration System". In: *Electronic Notes in Theoretical Computer Science* 137.3, pp. 65–75 [cit. on p. 75].

Winter, Andreas and Jürgen Ebert [2005a]. "Metamodel-driven Service Interoperability". In: *Pre-Proceedings of 13th International Workshop on Software Technology and Engineering Practice*. Ed. by Ying Zou and Massimiliano Di Penta. Queen's University, Kingston, pp. 167–176. URL: http://post.queensu.ca/%7B~%7Dzouy/files/preproc-step-2005.pdf%7B%5C#%7Dpage=178 [cit. on p. 75].

Winter, Andreas and Jürgen Ebert [2005b]. "Using metamodels in service interoperability". In: *Proceedings of 13th International Workshop on Software Technology and Engineering Practice*. Ed. by Kostas Kontogiannis, Ying Zou, and Massimiliano Di Penta. IEEE, Los Alamitos, pp. 147–156 [cit. on pp. 75, 76].

Bézivin, Jean, Hugo Bruneliere, Frédéric Jouault, and Ivan Kurtev [2005]. "Model Engineering Support for Tool Interoperability". In: *Proceedings of the 4th UML / MoDELS Workshop in Software Model Engineering*. URL: https://web.archive.org/web/20070812181354fw%7B%5C_%7D/http://www.planetmde.org/wisme-2005/ModelEngineeringSupportForToolInteroperability.pdf [cit. on p. 76].

Bruneliere, Hugo, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin [2010b]. "Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools". In: *Proceedings of the 6th European Conference on Modelling Foundations and Applications*. Ed. by Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier. Lecture Notes in Computer Science 6138. Springer, Berlin, Heidelberg, pp. 32–47 [cit. on p. 76].

Wirsing, Martin and Matthias Hölzl, eds. [2011]. *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineer-*

*ing for Service-Oriented Computing*. Vol. 6582. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg [cit. on p. 76].

Gönczy, László, Ábel Hegedüs, and Dániel Varró [2011]. "Methodologies for model-driven development and deployment: an overview". In: *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*. Ed. by Martin Wirsing and Matthias Hölzl. Lecture Notes in Computer Science 6582. Springer, Berlin, Heidelberg, pp. 541–560 [cit. on p. 76].

Amelunxen, Carsten, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr [2008]. "Metamodel-based tool integration with MOFLON". In: *Proceedings of the 30th International Conference on Software Engineering*. ACM Press, New York, pp. 807–810 [cit. on p. 76].

Hein, Christian, Tom Ritter, and Michael Wagner [2009]. "Model-Driven Tool Integration with ModelBus". In: *Proceedings of the 1st International Workshop on Future Trends of Model-Driven Development*. Ed. by Slimane Hammoudi and Luís Ferreira Pires. INSTICC, Setubal, pp. 35–39 [cit. on p. 77].

*ModelBus* [2017]. Fraunhofer Institute for Open Communication Systems. URL: `http://www.modelbus.org/` [visited on 01/15/2020] [cit. on p. 77].

Baumgart, Andreas [2010]. "A common meta-model for the interoperation of tools with heterogeneous data models". In: *Proceedings of the 3rd Workshop on Model-Driven Tool & Process Integration*. Ed. by Christian Hein, Michael Wagner, Roland Mader, Andreas Keis, and Eric Armengaud. Fraunhofer, Stuttgart, pp. 31–40 [cit. on p. 77].

Armengaud, Eric, Markus Zoier, Andreas Baumgart, Matthias Biehl, Dejiu Chen, Gerhard Griessnig, Christian Hein, Tom Ritter, and Ramin Tavakoli Kolagari [2011]. "Model-based toolchain for the efficient development of safety-relevant automotive embedded systems". In: *Proceedings of the SAE World Congress and Exhibition*. SAE International, Warrendale. URL: `https://saemobilus.sae.org/content/2011-01-0056` [cit. on p. 77].

Baumgart, Andreas, Christian Ellen, Stefan Farfeleder, Rainer Koopmann, Markus Oertel, and Philip Rehkop [2012]. "A reference technology platform with common interfaces for distributed heterogeneous data". In: *Proceedings of the Embedded World 2012 Exhibition and Conference*. WEKA-Fachmedien, Haar [cit. on p. 77].

Küpker, Christoph Alexander [2015]. "Applying the SENSEI Service Orchestration Approach to WSO2". Master's Thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 79, 85, 105, 230, 233, 256–258, 277, 312–314, 316, 376–378].

*Service Component Architecture (SCA)* [2015]. OASIS Open CSA. URL: `http://oasis-opencsa.org/sca` [visited on 01/15/2020] [cit. on pp. 79, 105, 231].

Hadley, Marc [2009]. *Web Application Description Language*. Standard. W3C, Cambridge [cit. on p. 80].

Biehl, Matthias, Wenqing Gu, and Frédéric Loiret [2012]. "Model-based service discovery and orchestration for OSLC services in tool chains". In: *Proceedings of the 12th International Conference on Web Engineering*. Ed. by Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf. Lecture Notes in Computer Science 7387. Springer, Berlin, Heidelberg, pp. 283–290 [cit. on pp. 83–85].

Biehl, Matthias, Jiarui Hong, and Frederic Loiret [2012]. "Automated Construction of Data Integration Solutions for Tool Chains". In: *Proceedings of the 7th International Conference on Software Engineering Advances*. IARIA, Wilmington, pp. 102–111 [cit. on pp. 84, 85].

Biehl, Matthias [2012]. *Semantic Anchoring of TIL*. Technical Report. Royal Institute of Technology, Stockholm. URL: `https://sites.google.com/site/mattbiehl/research/publications/semantics.pdf` [cit. on p. 84].

Nassi, Isaac and Ben Shneiderman [1973]. "Flowchart techniques for structured programming". In: *SIGPLAN Notices* 8.8, pp. 12–26 [cit. on p. 85].

Hull, Duncan, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn [2006]. "Taverna: A tool for building and running workflows of services". In: *Nucleic Acids Research* 34, W729–W732 [cit. on p. 87].

Wolstencroft, Katherine, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P Balcazar Vargas, Shoaib Sufi, and Carole Goble [2013]. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." In: *Nucleic acids research* 41, W557–W561 [cit. on p. 87].

Bhagat, Jiten, Franck Tanoh, Eric Nzuobontane, Thomas Laurent, Jerzy Orlowski, Marco Roos, Katy Wolstencroft, Sergejs Aleksejevs, Robert Stevens, Steve Pettifer, Rodrigo Lopez, and Carole A. Goble [2010]. "BioCatalogue: A universal catalogue of web services for the life sciences". In: *Nucleic Acids Research* 38, W689–W694 [cit. on p. 88].

Keenan, Ed, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn [2012]. "TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 1375–1378 [cit. on p. 88].

Frijters, Jeroen [2014]. *IKVM.NET Home Page*. URL: `http://www.ikvm.net/` [visited on 01/15/2020] [cit. on p. 88].

Margaria, Tiziana, Christian Kubczak, and Bernhard Steffen [2008]. "Bio-jETI: a service integration, design, and provisioning platform for orchestrated bioinformatics processes". In: *A Semantic Web for Bioinformatics: Goals, Tools, Systems, Applications*.

*Proceedings of the 7th International Workshop on Network Tools and Applications in Biology*. Ed. by Paolo Romano, Michael Schroeder, Nicola Cannata, and Roberto Marangoni. BMC Bioinformatics 9(Suppl 4).S12. BioMed Central, London [cit. on p. 88].

Margaria, Tiziana, Christian Kubczak, Mark Njoku, and Bernhard Steffen [2006]. "Model-based design of distributed collaborative bioinformatics processes in the jABC". In: *Proceedings of the 11th International Conference on Engineering of Complex Computer Systems*. IEEE, Los Alamitos, pp. 169–176 [cit. on p. 88].

Lamprecht, Anna-Lena, Tiziana Margaria, and Bernhard Steffen [2014]. "Modeling and Execution of Scientific Workflows with the jABC Framework". In: *Process Design for Natural Scientists: An Agile Model-Driven Approach*. Ed. by Anna-Lena Lamprecht and Tiziana Margaria. Communications in Computer and Information Science 500. Springer, Berlin, Heidelberg, pp. 14–29 [cit. on p. 88].

Lamprecht, Anna-Lena, Bernhard Steffen, and Tiziana Margaria [2016]. "Scientific workflows with the jABC framework: A review after a decade in the field". In: *International Journal on Software Tools for Technology Transfer* 18.6, pp. 629–651 [cit. on pp. 88, 89].

Margaria, Tiziana, Ralf Nagel, and Bernhard Steffen [2005]. "jETI : A Tool for Remote Tool Integration". In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Lecture Notes in Computer Science 3440. Springer, Berlin, Heidelberg, pp. 557–562 [cit. on p. 88].

Lamprecht, Anna-Lena [2013]. *User-Level Workflow Design: A Bioinformatics Perspective*. Vol. 8311. LNCS Programming and Software Engineering. Springer, Berlin, Heidelberg [cit. on pp. 88, 89].

Finnigan, Patrick J., Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong [1997]. "The software bookshelf". In: *IBM Systems Journal* 36.4, pp. 564–593 [cit. on pp. 89, 90].

Kienle, Holger M. [2006]. "Building Reverse Engineering Tools with Software Components". PhD thesis. University of Victoria [cit. on pp. 89, 91–93].

Kienle, Holger M. [2007]. "Building Reverse Engineering Tools with Software Components: Ten Lessons Learned". In: *Proceedings of the 14th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 289–292 [cit. on p. 91].

Ledbetter, Lamar and Brad Cox [1985]. "Software-ICs: A plan for building reusable software components". In: *BYTE* 10.6, pp. 307–316 [cit. on p. 99].

Conner, Mike, Nurcan Coskun, Scott Danforth, Larry Loucks, Andy Martin, Larry Raper, and Roger Sessions [1992]. "Developing language neutral class libraries with the System Object Model (SOM)". In: *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*. ACM, New York, pp. 191–193 [cit. on p. 99].

Kindel, Charlie [1997]. "COM: What Makes it Work — black-box encapsulation through multiple, immutable interfaces". In: *Proceedings of the 1st International Enterprise Distributed Object Computing Workshop*. IEEE, Los Alamitos, pp. 68–77 [cit. on p. 99].

Hamilton, Graham [1997]. *JavaBeans 1.01*. Specification. Sun Microsystems, Mountain View. URL: http://www.oracle.com/technetwork/articles/javaee/spec-136004.html [cit. on p. 100].

*Enterprise JavaBeans* [2019]. Oracle Corporation. URL: http://www.oracle.com/technetwork/java/javaee/ejb/ [visited on 01/15/2020] [cit. on pp. 100, 105, 237].

*OSGi* [2020]. OSGi Alliance. URL: https://www.osgi.org [cit. on pp. 100, 102, 105, 237].

Edwards, Mike and Martin Chapman [2016]. *Service Component Architecture Assembly Technical Committee*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sca-assembly [visited on 01/15/2020] [cit. on p. 100].

Heineman, George T. and William T. Councill, eds. [2001]. *Component-based software engineering: putting the pieces together*. Addison-Wesley, Boston [cit. on pp. 100, 102, 106].

Crnkovic, Ivica and Magnus Peter Henrik Larsson, eds. [2002]. *Building Reliable Component-based Software Systems*. Artech House, Boston [cit. on p. 100].

Crnkovic, Ivica, Severine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron [2011]. "A Classification Framework for Software Component Models". In: *IEEE Transactions on Software Engineering* 37.5, pp. 593–615 [cit. on pp. 102, 104, 106].

Sommerville, Ian [2011]. *Software Engineering*. 9th ed. Addison-Wesley, Boston [cit. on pp. 103, 110, 112, 118, 120, 122].

Weinreich, Rainer and Johannes Sametinger [2001]. "Component models and component services: concepts and principles". In: *Component-based software engineering: putting the pieces together*. Ed. by George T. Heineman and William T. Councill. Addison-Wesley, Boston, pp. 33–48 [cit. on p. 104].

*COM: Component Object Model Technologies* [2018]. Microsoft Corporation. URL: https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal [visited on 01/15/2020] [cit. on pp. 105, 237].

*CORBA Component Model (CCM) 4.0* [2006]. Object Management Group. URL: http://www.omg.org/spec/CCM [visited on 01/15/2020] [cit. on pp. 105, 237].

Ringe, Mathias [2013]. "Vergleich komponentenbasierter Frameworks zur Werkzeugintegration". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 105, 106, 237–239].

*Mailing List Archives of the SCA-Bindings Technical Committee* [2013]. OASIS. URL: https://lists.oasis-open.org/archives/sca-bindings/201306/maillist.html [visited on 01/15/2020] [cit. on pp. 106, 238].

*GlassFish Server* [2020]. Oracle. URL: http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html [visited on 01/15/2020] [cit. on p. 106].

*JBoss Developer* [2020]. Red Hat. URL: https://developer.jboss.org [visited on 01/15/2020] [cit. on p. 106].

*Equinox* [2020]. Eclipse Foundation. URL: http://www.eclipse.org/equinox/ [visited on 01/15/2020] [cit. on p. 106].

*Apache Felix* [2020]. Apache Software Foundation. URL: http://felix.apache.org/ [visited on 01/15/2020] [cit. on p. 106].

*Apache Tuscany* [2016]. Apache Software Foundation. URL: http://tuscany.apache.org/ [visited on 01/15/2020] [cit. on pp. 106, 237].

*Fabric3* [2016]. Metaform Systems. URL: https://web.archive.org/web/20180428063719/http://www.fabric3.org:80/ [visited on 01/15/2020] [cit. on pp. 106, 239].

*SwitchYard* [2020]. Red Hat. URL: http://switchyard.jboss.org/ [visited on 2020] [cit. on pp. 106, 239].

Mell, Peter and Timothy Grance [2011]. *The NIST definition of cloud computing*. Special Publication NIST SP 800-145. National Institute of Standards and Technology, Gaithersburg, MD [cit. on pp. 107, 199].

Schulte, W. Roy and Yefim Natis [1996]. *"Service Oriented" Architectures, Part 1*. Research Note SPA-401-068, G0029201. Gartner, Stamford [cit. on p. 109].

*Gartner Hype Cycle* [2015]. Gartner. URL: http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp [visited on 01/15/2020] [cit. on p. 109].

Vaughan-Nichols, Steven J. [2002]. "Web services: beyond the hype". In: *Computer* 35.2, pp. 18–21 [cit. on p. 109].

Manes, Anne Thomas [2009]. *SOA Is Dead; Long Live Services*. URL: https://web.archive.org/web/20160506053536/http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services/comments/page/1/ [visited on 01/15/2020] [cit. on pp. 109, 111, 118, 122].

Richardson, Leonard and Sam Ruby [2008]. *RESTful Web Services*. O'Reilly, Sebastopol [cit. on pp. 109, 119, 120, 122, 125].

Lewis, James and Martin Fowler [2014]. *Microservices*. URL: http://martinfowler.com/articles/microservices.html [visited on 01/15/2020] [cit. on pp. 109, 114, 122].

Breivold, Hongyu Pei and Magnus Larsson [2007]. "Component-based and service-oriented software engineering: Key concepts and principles". In: *Proceedings of*

*the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, Los Alamitos, pp. 13–20 [cit. on p. 112].

MacKenzie, C. Matthew, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz [2006]. *Reference Model for Service Oriented Architecture 1.0*. Standard. OASIS, Burlington [cit. on pp. 112, 118].

Vogel, Oliver, Ingo Arnold, Arif Chughtai, and Timo Kehrer [2011]. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. Springer, Berlin, Heidelberg [cit. on pp. 112, 113].

Melzer, Ingo and Sebastian Eberhard [2010]. *Service-orientierte Architekturen mit Web Services*. 4. Edition. Spektrum, Heidelberg [cit. on pp. 112, 113, 119, 120].

Erl, Thomas [2007]. *SOA Principles of Service Design*. Prentice Hall, Upper Saddle River [cit. on pp. 112, 114, 115, 120, 121].

Marino, Jim and Michael Rowley [2009]. *Understanding SCA (Service Component Architecture)*. Addison Wesley, Boston [cit. on pp. 113, 238, 240, 299].

Schmidt, Alexander, Boris Otto, and Hubert Österle [2010]. "Integrating information systems: Case studies on current challenges". In: *Electronic Markets* 20.2, pp. 161–174 [cit. on p. 116].

Krafzig, Dirk, Karl Banke, and Dirk Slama [2005]. *Enterprise SOA: Service-oriented Architecture Best Practices*. Prentice Hall, Upper Saddle River [cit. on p. 116].

Hentrich, Carsten and Uwe Zdun [2009]. "A pattern language for process execution and integration design in service-oriented architectures". In: *Transactions on Pattern Languages of Programming I*. Ed. by James Noble and Ralph Johnson. Lecture Notes in Computer Science 5770. Springer, Berlin, Heidelberg, pp. 136–191 [cit. on p. 116].

Manolescu, Dragos [2000]. "Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development". PhD thesis. University of Illinois at Urbana-Champaign [cit. on p. 116].

Cohen, Shy [2007]. "Ontology and Taxonomy of Services in a Service-Oriented Architecture". In: *The Architecture Journal (Microsoft Online Publication)* 11. URL: `https://web.archive.org/web/20110127030807/http://msdn.microsoft.com/en-us/library/bb491121.aspx` [cit. on p. 117].

Siedersleben, Johannes [2004]. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. Dpunkt, Heidelberg [cit. on pp. 117, 118].

Fowler, Martin [2005a]. *ServiceOrientedAmbiguity*. URL: `http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html` [visited on 01/15/2020] [cit. on pp. 118, 122].

Winter, Andreas [2000]. *Referenz-Metaschema für visuelle Modellierungssprachen*. Deutscher Universitätsverlag, Wiesbaden [cit. on p. 118].

Laskey, Ken, Peter Brown, Jeff A. Estefan, Francis G. McCabe, and Danny Thornton [2012]. *Reference Architecture for Service Oriented Architecture Version 1.0*. Standard. OASIS, Burlington [cit. on p. 118].

ISO/IEC 18384 [2016]. *Information technology – Reference Architecture for Service Oriented Architecture (SOA RA)*. International Standard. International Organization for Standardization, Geneva [cit. on p. 118].

Henning, Michi [2006]. "The Rise and Fall of CORBA". In: *Queue* 4.5, pp. 28–34 [cit. on p. 118].

Zimmermann, Olaf, Mark Tomlinson, and Stefan Peuser [2005]. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. 2nd correc. Springer, Berlin, Heidelberg [cit. on pp. 119, 125].

Menascé, Daniel A. [2005]. "MOM vs. RPC: Communication Models for Distributed Applications". In: *IEEE Internet Computing* 9.2, pp. 90–93 [cit. on p. 119].

Marx Gómez, Jorge [2019]. "Serviceorientierte Architektur". In: *Enzyklopädie der Wirtschaftsinformatik – Online-Lexikon*. Ed. by Norbert Gronau, Jörg Becker, Natalia Kliewer, Jan Marco Leimeister, and Sven Overhage. GITO, Berlin [cit. on pp. 119–121].

Ran, Shuping [2003]. "A model for web services discovery with QoS". In: *ACM SIGecom Exchanges* 4.1, pp. 1–10 [cit. on p. 120].

Zhu, Haibin [2005]. "Challenges to Reusable Services". In: *Proceedings of the International Conference on Services Computing, Volume II*. IEEE, Los Alamitos, pp. 243–244 [cit. on p. 120].

Michlmayr, Anton, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar [2007]. "Towards recovering the broken SOA triangle". In: *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering*. ACM, New York, pp. 22–28 [cit. on pp. 120, 121].

Gray, Jim [2006]. "A conversation with Werner Vogels". In: *Queue* 4.4. Ed. by Charlene O'Hanlon, pp. 14–22 [cit. on p. 122].

Kokko, Timo, Jari Antikainen, and Tarja Systä [2009]. "Adopting SOA - Experiences from nine finnish organizations". In: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 129–138 [cit. on pp. 122, 164].

Nurkiewicz, Tomasz [2015]. *RESTful Considered Harmful*. URL: `https://dzone.com/articles/restful-considered-harmful` [visited on 01/15/2020] [cit. on p. 122].

Martin, Robert C. [2003]. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River [cit. on pp. 122, 292].

Hofstede, Arthur H. M. ter, Wil M. P. van der Aalst, Michael Adams, and Nick Russell, eds. [2010]. *Modern Business Process Automation: YAWL and Its Support Environment*. Springer, Berlin, Heidelberg [cit. on pp. 123, 124, 193].

Hollingsworth, David [1995]. *The Workflow Reference Model*. Standard TC00-1003. Workflow Management Coalition, Winchester [cit. on p. 123].

Draheim, Dirk [2010]. *Business Process Technology: A Unified View on Business Processes, Workflows and Enterprise Applications*. Springer, Berlin, Heidelberg [cit. on p. 123].

Hohpe, Gregor and Bobby Woolf [2004]. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston [cit. on pp. 124, 193].

Jordan, Diane, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu [2007]. *Web Services Business Process Execution Language Version 2.0*. Standard. OASIS, Burlington [cit. on p. 124].

*Business Process Model and Notation (BPMN) Version 2.0* [2011]. Object Management Group. URL: `http://www.omg.org/spec/BPMN/2.0/PDF/` [visited on 01/15/2020] [cit. on p. 124].

Russell, Nick, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed [2006]. "On the suitability of UML 2.0 activity diagrams for business process modelling." In: *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modeling*. Ed. by Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki. Conferences in Research and Practice in Information Technology 53, Australian Computer Science Communications 28.6. Australian Computer Society, Darlinghurst, pp. 95–104 [cit. on p. 124].

Wohed, Petia, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell [2006]. "On the Suitability of BPMN for Business Process Modelling". In: *Proceedings of the 4th International Conference on Business Process Management*. Ed. by Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth. Lecture Notes in Computer Science 4102. Springer, Berlin, Heidelberg, pp. 161–176 [cit. on p. 124].

Aalst, Wil M. P. van der, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros [2003]. "Workflow Patterns". In: *Distributed and Parallel Databases* 14.1, pp. 5–51 [cit. on p. 124].

Bieberstein, Norbert, Robert G. Laird, Keith Jones, and Tilak Mitra [2008]. *Executing SOA: A Practical Guide for the Service-Oriented Architect*. IBM Press, Indianapolis [cit. on p. 125].

Decker, Gero, Oliver Kopp, Frank Leymann, and Mathias Weske [2007]. "BPEL4Chor: Extending BPEL for Modeling Choreographies". In: *Proceedings of the 5th International Conference on Web Services*. IEEE, Los Alamitos, pp. 296–303 [cit. on p. 125].

Hollingsworth, David [2004]. "The Workflow Reference Model: 10 Years On". In: *Workflow Handbook 2004*. Ed. by Layna Fischer. Future Strategies, Lighthouse Point, pp. 295–312 [cit. on p. 125].

Kopp, Oliver and Frank Leymann [2008]. "Choreography Design Using WS-BPEL". In: *Bulletin of the Technical Committee on Data Engineering* 31.3, pp. 31–34 [cit. on p. 125].

Kavantzas, Nickolas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto [2005]. *Web Services Choreography Description Language Version 1.0*. Standard. W3C, Cambridge [cit. on p. 125].

Decker, Gero, Oliver Kopp, and Alistair Barros [2008]. "An Introduction to Service Choreographies (Servicechoreographien – eine Einführung)". In: *it - Information Technology* 50.2, pp. 122–127 [cit. on p. 125].

Michelson, Branda M. [2006]. *Event-Driven Architecture Overview: Event-Driven SOA Is Just Part of the EDA Story*. White paper. Patricia Seybold Group, Boston [cit. on p. 125].

Levina, Olga and Vladimir Stantchev [2009]. "Realizing Event-Driven SOA". In: *Proceedings of the 4th International Conference on Internet and Web Applications and Services*. IEEE, Los Alamitos, pp. 37–42 [cit. on p. 125].

Atkinson, Colin and Thomas Kühne [2003]. "Model-driven development: a metamodeling foundation". In: *IEEE Software* 20.5, pp. 36–41 [cit. on pp. 129, 136].

Stachowiak, Herbert [1973]. *Allgemeine Modelltheorie*. Springer, Wien [cit. on p. 132].

Skyttner, Lars [2005]. *General Systems Theory: Problems, Perspectives, Practice*. 2nd ed. World Scientific, Singapore [cit. on p. 132].

Seidewitz, Ed [2003]. "What models mean". In: *IEEE Software* 20.5, pp. 26–32 [cit. on p. 133].

Aßmann, Uwe, Steffen Zschaler, and Gerd Wagner [2006]. "Ontologies, Meta-models, and the Model-Driven Paradigm". In: *Ontologies for Software Engineering and Software Technology*. Ed. by Coral Calero, Francisco Ruiz, and Mario Piattini. Springer, Berlin, Heidelberg, pp. 249–273 [cit. on p. 133].

Peirce, Charles Santiago Sanders [1906]. "Prolegomena to an Apology for Pragmaticism". In: *Monist* 16.4, pp. 492–546 [cit. on p. 133].

Kühne, Thomas [2006]. "Matters of (Meta-) Modeling". In: *Software & Systems Modeling* 5.4, pp. 369–385 [cit. on p. 133].

Kleppe, Anneke [2009]. "The Field of Software Language Engineering". In: *Proceedings of the First International Conference on Software Language Engineering*. Ed. by Dragan Gašević, Ralf Lämmel, and Eric Van Wyk. Lecture Notes in Computer Science 5452. Springer, Berlin, Heidelberg, pp. 1–7 [cit. on p. 134].

Atkinson, Colin and Thomas Kühne [2001]. "The essence of multilevel metamodeling". In: *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gogolla and Cris Kobryn. Lecture Notes in Computer Science 2185. Springer, Berlin, Heidelberg, pp. 19–33 [cit. on p. 136].

Goldstein, Robert C. and Veda C. Storey [1994]. "Materialization". In: *IEEE Transactions on Knowledge and Data Engineering* 6.5, pp. 835–842 [cit. on p. 136].

Gonzalez-Perez, Cesar and Brian Henderson-Sellers [2008]. *Metamodelling for Software Engineering*. Wiley, Chichester [cit. on p. 136].

Neumayr, Bernd, Katharina Grün, and Michael Schrefl [2009]. "Multi-level domain modeling with m-objects and m-relationships". In: *Proceedings of the 6th Asia-Pacific Conference on Conceptual Modeling*. Ed. by Markus Kirchberg and Sebastian Link. Conferences in Research and Practice in Information Technology 96, Australian Computer Science Communications 31.6. Australian Computer Society, Darlinghurst, pp. 107–116 [cit. on p. 136].

Laarman, Alfons and Ivan Kurtev [2010]. "Ontological metamodeling with explicit instantiation". In: *Proceedings of the 2nd International Conference on Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Lecture Notes in Computer Science 5969. Springer, Berlin, Heidelberg, pp. 174–183 [cit. on p. 136].

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer [2012]. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, Williston [cit. on p. 136].

Wagner, Christian [2014]. *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Fachmedien, Wiesbaden [cit. on p. 136].

*Model Driven Architecture* [2020]. Object Management Group. URL: `http://www.omg.org/mda/` [visited on 01/15/2020] [cit. on p. 136].

Pastor, Óscar and Juan Carlos Molina [2007]. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer, Berlin, Heidelberg [cit. on pp. 136, 138].

Karsai, Gabor, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty [2003]. "Model-integrated development of embedded software". In: *Proceedings of the IEEE* 91.1, pp. 145–164 [cit. on p. 137].

Zeppenfeld, Klaus and Regine Wolters [2005]. *Generative Software-Entwicklung mit der MDA*. Spektrum, Heidelberg [cit. on pp. 137, 138].

Siegel, Jon M. [2014]. *MDA Guide Rev. 2.0*. White paper ORMSC/2014-06-01. Object Management Group, Needham [cit. on pp. 138, 140].

Guttman, Michael and John Parodi [2006]. *Real-Life MDA: Solving Business Problems with Model Driven Architecture*. Morgan Kaufmann, San Francisco [cit. on p. 138].

Mens, Tom and Pieter Van Gorp [2006]. "A Taxonomy of Model Transformation". In: *Proceedings of the International Workshop on Graph and Model Transformation*. Ed. by Gabor Karsai and Gabriele Taentzer. Electronic Notes in Theoretical Computer Science 152. Elsevier, Amsterdam, pp. 125–142 [cit. on pp. 139, 140].

Czarnecki, Krzysztof and Simon Helsen [2003]. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. URL: `https://s23m.com/oopsla2003/czarnecki.pdf` [cit. on pp. 139, 140].

Czarnecki, Krzysztof and Simon Helsen [2006]. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3, pp. 621–645 [cit. on pp. 139, 140].

Jakumeit, Edgar, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek [2014]. "A survey and comparison of transformation tools based on the transformation tool contest". In: *Science of Computer Programming* 85, Part A, pp. 41–99 [cit. on pp. 140, 144].

Jouault, Frédéric, Jean Bézivin, and Mikaël Barbero [2009]. "Towards an advanced model-driven engineering toolbox". In: *Innovations in Systems and Software Engineering* 5.1, pp. 5–12 [cit. on p. 140].

Kurtev, Ivan, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez [2006]. "Model-based DSL frameworks". In: *Proceedings of the 21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 602–616 [cit. on p. 140].

Cook, Steve [2004]. "Domain-Specific Modeling and Model Driven Architecture". In: *The MDA Journal: Model Driven Architecture Straight from the Masters*. Meghan Kiffer, Tampa. Chap. 5 [cit. on p. 140].

Bentley, Jon [1986]. "Programming pearls". In: *Communications of the ACM* 29.8, pp. 711–721 [cit. on p. 140].

Deursen, Arie van, Paul Klint, and Joost Visser [2000]. "Domain-Specific Languages: An Annotated Bibliography". In: *ACM SIGPLAN Notices* 35.6, pp. 26–36 [cit. on p. 140].

Czarnecki, Krzysztof [2004]. "Overview of generative software development". In: *International Workshop on Unconventional Programming Paradigms, Revised Selected and Invited Papers*. Ed. by Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel. Lecture Notes in Computer Science 3566. Springer, Berlin, Heidelberg, pp. 326–341 [cit. on pp. 140, 141, 144].

Ghosh, Debasish [2010]. *DSLs in Action*. Manning, Shelter Island [cit. on p. 141].

Sprinkle, Jonathan, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis [2009]. "What Kinds of Nails Need a Domain-Specific Hammer". In: *IEEE Software* 26.4, pp. 15–18 [cit. on p. 141].

Whittle, Jon, John Hutchinson, and Mark Rouncefield [2014]. "The State of Practice in Model-Driven Engineering". In: *IEEE Software* 31.3, pp. 79–85 [cit. on p. 141].

Abouzahra, Anas, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault [2005]. "A practical approach to bridging domain specific languages with UML profiles". In: *Proceedings of the 4th OOPSLA Workshop on Best Practices for Model Driven Software Development*. URL: `https://www.s23m.com/oopsla2005/bezivin1.pdf` [cit. on p. 141].

Fowler, Martin [2005b]. *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: http://www.martinfowler.com/articles/languageWorkbench.html [visited on 01/15/2020] [cit. on pp. 141, 212].

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning [2013]. "The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge". In: *Proceedings of the 6th International Conference on Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Lecture Notes in Computer Science 8225. Springer, Cham, pp. 197–217 [cit. on pp. 141, 144].

Fowler, Martin [2010]. *Domain-Specific Languages*. Addison-Wesley, Boston [cit. on p. 141].

Clements, Paul and Linda Northrop [2002]. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston [cit. on p. 141].

Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson [1990]. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Pittsburgh [cit. on p. 142].

Kang, Kyo C., Jaejoon Lee, and Patrick Donohoe [2002]. "Feature-oriented product line engineering". In: *IEEE Software* 19.4, pp. 58–65 [cit. on p. 142].

Kurtev, Ivan, Jean Bézivin, and Mehmet Aksit [2002]. *Technological Spaces: An Initial Appraisal*. Paper presented at the 4th International Symposium on Distributed Objects and Applications. University of Twente. URL: https://research.utwente.nl/en/publications/technological-spaces-an-initial-appraisal [cit. on p. 142].

Bézivin, Jean and Ivan Kurtev [2006]. *Model-based technology integration with the technical space concept*. Tech. rep. hal-00483587. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on pp. 142, 143].

Bézivin, Jean [2006]. "Model driven engineering: an emerging technical space". In: *Generative and Transformational Techniques in Software Engineering*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Lecture Notes in Computer Science 4143. Springer, Berlin, Heidelberg, pp. 36–64 [cit. on p. 142].

Klint, Paul, Ralf Lämmel, and Chris Verhoef [2005]. "Toward an engineering discipline for grammarware". In: *ACM Transactions on Software Engineering and Methodology* 14.3, pp. 331–380 [cit. on p. 142].

Thompson, Henry S., David Beech, Murray Maloney, and Noah Mendelsohn [2004]. *XML Schema Part 1: Structures Second Edition*. URL: https://www.w3.org/TR/xmlschema-1/#normative-schemaSchema [visited on 01/15/2020] [cit. on p. 142].

Microsoft [2016]. *Overview of Domain-Specific Language Tools*. URL: https://msdn.microsoft.com/en-us/library/bb126327.aspx [visited on 01/15/2020] [cit. on pp. 143, 213].

Greenfield, Jack and Keith Short [2004]. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Chichester [cit. on p. 143].

Ebert, Jürgen [1987]. "A versatile data structure for edge-oriented graph algorithms". In: *Communications of the ACM* 30.6, pp. 513–519 [cit. on p. 143].

Ebert, Jürgen [2008]. "Metamodels Taken Seriously: The TGraph Approach". In: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*. Ed. by Kostas Kontogiannis, Christos Tjortjis, and Andreas Winter. IEEE, Los Alamitos, p. 2 [cit. on p. 143].

*Graphical Editing Framework* [2020]. Eclipse Foundation. URL: https://eclipse.org/gef/ [visited on 01/15/2020] [cit. on p. 143].

*Graphical Modeling Framework* [2020]. Eclipse Foundation. URL: http://www.eclipse.org/modeling/gmp/ [visited on 01/15/2020] [cit. on pp. 143, 213].

*Graphiti* [2020]. Eclipse Foundation. URL: https://eclipse.org/graphiti/ [visited on 01/15/2020] [cit. on pp. 143, 213].

Kolovos, Dimitrios S., Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck [2010]. "Taming EMF and GMF Using Model Transformation". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, Part I*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Lecture Notes in Computer Science 6394. Springer, Berlin, Heidelberg, pp. 211–225 [cit. on pp. 143, 213].

*Sirius* [2020]. Eclipse Foundation. URL: https://eclipse.org/sirius/index.html [visited on 01/15/2020] [cit. on pp. 143, 211, 213].

Eysholdt, Moritz and Heiko Behrens [2010]. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, New York, pp. 307–309 [cit. on p. 143].

Jouault, Frédéric, F Allilaire, Jean Bézivin, I Kurtev, and P Valduriez [2006]. "ATL: a QVT-like transformation language". In: *Proceedings of the 21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 719–720 [cit. on p. 143].

Hildebrandt, Stephan, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr [2013]. "A survey of triple graph grammar tools". In: *Proceedings of the 2nd Workshop on Bidirectional Transformations*. Ed. by Perdita Stevens and James F. Terwilliger. Electronic Communications of the EASST 57. European Association of Software Science and Technology [cit. on p. 143].

Kelly, Steven, Kalle Lyytinen, and Matti Rossi [1996]. "Metaedit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment". In: *Proceedings of the 8th International Conference on Advanced Information Systems Engineering*. Ed. by Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou. Lecture Notes in Computer Science 1080. Springer, Berlin, Heidelberg, pp. 1–21 [cit. on pp. 143, 213].

Ledeczi, Akos, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi [2001]. "The Generic Modeling Environment". In: *Proceedings of the 2nd Workshop on Intelligent Signal Processing*. IEEE, Los Alamitos. URL: `https://www.isis.vanderbilt.edu/sites/default/files/GME2000Overview.pdf` [cit. on p. 143].

Pech, Vaclav, Alex Shatalin, and Markus Völter [2013]. "JetBrains MPS as a tool for extending Java". In: *Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools*. ACM, New York, pp. 165–168 [cit. on pp. 143, 213].

Kats, Lennart C. L. and Eelco Visser [2010]. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 444–463 [cit. on p. 143].

Klint, Paul, Tijs van der Storm, and Jurgen Vinju [2011]. "{EASY} Meta-programming with Rascal". In: *Generative and Transformational Techniques in Software Engineering III - International Summer School*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Lecture Notes in Computer Science 6491. Springer, Berlin, Heidelberg, pp. 222–289 [cit. on p. 143].

Visser, Eelco [2004]. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Lecture Notes in Computer Science 3016. Springer, Berlin, Heidelberg, pp. 216–238 [cit. on p. 144].

Cordy, James R. [2004]. "TXL - A Language for Programming Language Tools and Applications". In: *Electronic Notes in Theoretical Computer Science* 110, pp. 3–31 [cit. on p. 144].

Amyot, Daniel, Hanna Farah, and Jean-François Roy [2006]. "Evaluation of development tools for domain-specific modeling languages". In: *Proceedings of the 5th International Workshop on System Analysis and Modeling: Language Profiles*. Ed. by Reinhard Gotzhein and Rick Reed. Lecture Notes in Computer Science 4320. Springer, Berlin, Heidelberg, pp. 183–197 [cit. on p. 144].

de Sousa Saraiva, João and Alberto Rodrigues da Silva [2008]. "Evaluation of MDE Tools from a Metamodeling Perspective". In: *Journal of Database Management* 19.4, pp. 50–75 [cit. on p. 144].

Meier, Johannes [2014a]. "Editoren für Service-Orchestrierungen". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 157, 182, 211–213, 215, 216].

Wirth, Niklaus [1986]. *Algorithms and Data Structures*. Prentice Hall, Upper Saddle River [cit. on p. 166].

Dale, Nell and Henry M. Walker [1996]. *Abstract Data Types: Specifications, Implementations, and Applications*. D. C. Heath and Company, Lexington [cit. on p. 166].

*Object Constraint Language* [2014]. Object Management Group. URL: http://www.omg.org/spec/OCL/2.4 [visited on 01/15/2020] [cit. on p. 175].

Kuryazov, Dilshodbek [2014]. "Delta Operations Language for Model Difference Representation". In: *Informatik 2014: Big Data, Komplexität meistern. Beitragsband der 44. Jahrestagung der Gesellschaft für Informatik*. Ed. by Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull. Lecture Notes in Informatics 232. Gesellschaft für Informatik, Bonn, pp. 2221–2232 [cit. on p. 193].

Kuryazov, Dilshodbek and Andreas Winter [2014]. "Representing Model Differences by Delta Operations". In: *Proceedings of the 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*. IEEE, Los Alamitos, pp. 211–220 [cit. on p. 193].

Winter, Andreas, Bernt Kullbach, and Volker Riediger [2002]. "An Overview of the GXL Graph Exchange Language". In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer, Berlin, Heidelberg, pp. 324–336 [cit. on p. 193].

*Apache Camel* [2020]. Apache Software Foundation. URL: http://camel.apache.org/ [visited on 01/15/2020] [cit. on p. 193].

*Apache Maven* [2020]. Apache Software Foundation. URL: https://maven.apache.org/ [visited on 01/15/2020] [cit. on pp. 195, 199, 235].

Brodie, Michael L. [1984]. "On the Development of Data Models". In: *On conceptual modelling*. Ed. by Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt. Topics in Information Systems. Springer, New York. Chap. 2, pp. 19–47 [cit. on p. 200].

Red Hat [2015]. *JBoss Fuse Service Works 6.0 Development Guide Volume 1: SwitchYard*. Development Guide. Red Hat, Raleigh. URL: https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse_service_works/6.0/pdf/development_guide_volume_1_switchyard/Red_Hat_JBoss_Fuse_Service_Works-6.0-Development_Guide_Volume_1_SwitchYard-en-US.pdf [cit. on p. 200].

Resende, Luciano and Raymond Feng [2007]. "Handling Heterogeneous Data Sources in a SOA Environment with Service Data Objects (SDO)". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Volume 1*. Ed. by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, New York, pp. 895–897 [cit. on p. 201].

Meier, Almuth [2014b]. "Ein Composition-Finder für Service-Orchestrierungen". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 202, 204, 207, 208, 243, 256, 382, 385].

Ebert, Jürgen and Angelika Franzke [1995]. "A declarative approach to graph based modeling". In: *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Lecture Notes in Computer Science 903. Springer, Berlin, Heidelberg, pp. 38–50 [cit. on pp. 211, 236].

Heckelmann, Kristina [2010]. "Abbildung von Ecore nach grUML". Bachelor's Thesis. University of Koblenz [cit. on p. 211].

*Sirius Specifier Manual* [2020]. Eclipse Foundation. URL: `https://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html` [visited on 01/15/2020] [cit. on pp. 211, 220].

*MOFM2T 1.0* [2008]. Object Management Group. URL: `http://www.omg.org/spec/MOFM2T/1.0/` [visited on 01/15/2020] [cit. on p. 212].

*Acceleo* [2020]. Eclipse Foundation. URL: `http://www.eclipse.org/acceleo/` [visited on 01/15/2020] [cit. on p. 212].

*Xtext - Language Engineering for Everyone!* [2020]. Eclipse Foundation. URL: `http://www.eclipse.org/Xtext/` [visited on 01/15/2020] [cit. on p. 212].

Kühne, Stefan and Christian Wetzel [2006]. "Metamodellierung am Beispiel der E-Government-Domäne Meldewesen und Eclipse GMF". In: *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*. Ed. by Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck, and Julia Wagner. Leipziger Beiträge zur Informatik IV. Leipziger Informatik-Verbund, Leipzig, pp. 59–72 [cit. on p. 212].

MetaCase [2020]. *MetaCase - Domain-Specific Modeling with MetaEdit+*. URL: `http://www.metacase.com/` [visited on 01/15/2020] [cit. on p. 213].

Kolovos, Dimitrios S., Louis M. Rose, Richard F. Paige, and Fiona A. C. Polack [2009]. "Raising the level of abstraction in the development of GMF-based graphical model editors". In: *Proceedings of the 1st Workshop on Modeling in Software Engineering*. IEEE, Los Alamitos, pp. 13–19 [cit. on p. 213].

Wienands, Christoph and Michael Golm [2009]. "Anatomy of a Visual Domain-Specific Language Project in an Industrial Context". In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Lecture Notes in Computer Science 5795. Springer, Berlin, Heidelberg, pp. 453–467 [cit. on p. 213].

Kouhen, Amine El, Cedric Dumoulin, Sébastien Gerard, and Pierre Boulet [2012]. *Evaluation of Modeling Tools Adaptation*. Tech. rep. hal-00706701. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on p. 213].

Strittmatter, Misha, Michael Junker, Kiana Rostami, Sebastian Lehrig, Amine Kechaou, Bo Liu, and Robert Heinrich [2016]. "Extensible Graphical Editors for Palladio". In: *Softwaretechnik Trends* 36.4, pp. 49–51 [cit. on p. 214].

Vujović, Vladimir, Mirjana Maksimović, and Branko Perišić [2014]. "Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs . Sirius". In: *Proceedings of the 23rd International Electrotechnical and Computer Science Conference*. Ed. by Baldomir Zajc and Andrej Trost. University of Ljubljana, Ljubljana, pp. 7–10. URL: https://erk.fe.uni-lj.si/2014/index.html [cit. on p. 214].

Kamp, Manfred [1998]. "Managing a multi-file, multi-language software repository for-program comprehension tools: a generic approach". In: *Proceedings of the 6th International Workshop on Program Comprehension*. IEEE, Los Alamitos [cit. on p. 236].

Marchewka, Katrin [2006]. "GReQL 2". Diploma Thesis. University of Koblenz-Landau [cit. on p. 236].

Ebert, Jürgen and Tassilo Horn [2012]. "GReTL: an extensible, operational, graph-based transformation language". In: *Software & Systems Modeling* 13.1. Ed. by Jordi Cabot and Eelco Visser, pp. 301–321 [cit. on p. 236].

*Apache Velocity* [2020]. Apache Software Foundation. URL: http://velocity.apache.org/ [visited on 01/15/2020] [cit. on p. 236].

Chapman, Martin, Mike Edwards, Michael Beisiegel, Anish Karmarkar, Sanjay Patil, and Michael Rowley [2011]. *Service Component Architecture Assembly Model Specification Version 1.1*. Standard. OASIS, Burlington. URL: http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec-v1.1-csprd03.html [cit. on p. 237].

Meier, Johannes [2012]. "Eine Fallstudie zur Interoperabilität von Software-Evolutions-Werkzeugen in SCA". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Tihonov, Sergej [2013]. "Servicebasierte Refactorings". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Crone, Thomas [2013]. "Software-Evolutions-Services zur Berechnung und Visualisierung von Metriken". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Yandell, Henri [2016]. *Apache Tuscany retired*. URL: http://mail-archives.apache.org/mod_mbox/www-announce/201608.mbox/browser [visited on 01/15/2020] [cit. on p. 238].

IBM [2016]. *Statements of deprecation and general direction: IBM Rational Application Developer for WebSphere Software and the IBM Rational Software Architect Designer program family*. URL: http://www-01.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/5/897/ENUS216-245/index.html&lang=en&request_locale=en [visited on 01/19/2017] [cit. on p. 239].

Tsang, Edward. [1993]. *Foundations of constraint satisfaction*. Academic Press, London [cit. on p. 243].

Wielemaker, Jan [2020]. *SWI-Prolog*. URL: http://www.swi-prolog.org/ [visited on 01/15/2020] [cit. on p. 243].

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides [1995]. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston [cit. on p. 256].

Lin, Yuan, Richard C. Holt, and Andrew J. Malton [2003]. "Completeness of a fact extractor". In: *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 196–205 [cit. on p. 268].

Ferenc, Rudolf, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy [2001]. "Towards a standard schema for C/C++". In: *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 49–58 [cit. on p. 268].

Becker, Christian and Uwe Kaiser [2014]. "Applikationswissen in der Sprachkonvertierung am Beispiel des COBOL-Java-Converters CoJaC." In: *Softwaretechnik-Trends* 34.2 [cit. on p. 268].

Becker, Christian and Uwe Kaiser [2016]. "Toolbasierte Software-Migration nach Plan". In: *Softwaretechnik-Trends* 36.2 [cit. on p. 268].

*Code Conventions for the Java Programming Language* [1999]. Oracle. URL: http://www.oracle.com/technetwork/java/index-135089.html [visited on 03/17/2017] [cit. on p. 268].

*JGroups* [2020]. Red Hat. URL: http://www.jgroups.org/ [visited on 01/15/2020] [cit. on p. 299].

Statistisches Bundesamt [2015]. *Statistisches Jahrbuch Deutschland 2015*. Statistisches Bundesamt, Wiesbaden [cit. on p. 308].

Combemale, Benoit, Betty H.C. Cheng, Ana Moreira, Jean-Michel Bruel, and Jeff Gray [2016]. "Modeling for Sustainability". In: *Proceedings of the 8th International Workshop on Modeling in Software Engineering*. ACM, New York, pp. 62–66 [cit. on p. 309].

Rajlich, Vaclav and Keith Bennett [2000]. "A Staged Model for the Software Life Cycle". In: *Computer* 33.7, pp. 66–71 [cit. on p. 309].

*ICT Services* [2016]. URL: http://www.ikts-niedersachsen.de/en [visited on 01/15/2020] [cit. on p. 309].

Wagner vom Berg, Benjamin [2015]. *Konzeption Eines Sustainability Customer Relationship Managements (SusCRM) Für Anbieter Nachhaltiger Mobilität*. Shaker, Herzogenrath [cit. on p. 309].

*Google Maps Directions API* [2020]. Google. URL: https://developers.google.com/maps/documentation/directions/ [visited on 01/15/2020] [cit. on p. 315].

*Google Places API* [2020]. Google. URL: https://developers.google.com/places/ [visited on 01/15/2020] [cit. on p. 315].

*IFML: The Interaction Flow Modeling Language* [2017]. Object Management Group. URL: http://www.ifml.org/ [visited on 07/28/2017] [cit. on pp. 323, 325].

Schlömer, Timo [2017]. "Modellgetriebene GUI-Erstellung für serviceorientierte Anwendungen". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 324–326, 342].

Bischopink, Christopher, Stephan Bogs, Hauke Fischer, Hannah Meyer, Felix Kempa, Nancy Kramer, Thomas Sprock, and Lisa Ripke [2018]. *Projektgruppe DORI*. Project Report. Carl von Ossietzky University, Oldenburg [cit. on pp. 324, 326, 327, 342].

Chignell, Mark H. [1990]. "A taxonomy of user interface terminology". In: *ACM SIGCHI Bulletin* 21.4, p. 27 [cit. on p. 325].

Kuryazov, Dilshodbek, Andreas Winter, and Alexander Sandau [2019]. "Sustainable Software Architecture for NEMo Mobility Platform". In: *Smart Cities/Smart Regions - Technische, wirtschaftliche und gesellschaftliche Innovationen. Konferenzband zu den 10. BUIS-Tagen*. Ed. by Jorge Marx Gómez, Andreas Solsbach, Thomas Klenke, and Volker Wohlgemuth. Springer Fachmedien, Wiesbaden, pp. 229–239 [cit. on pp. 327, 339].

Akyol, Ali, Jantje Halberstadt, Kimberly Hebig, Jan Jelschen, Andreas Winter, Alexander Sandau, and Jorge Marx Gómez [2017]. "Flexible Software Support for Mobility Services". In: *Informatik 2017. Beitragsband der 47. Jahrestagung der Gesellschaft für Informatik*. Ed. by Maximilian Eibl, Martin Gaedke, and Cornelia Winter. Gesellschaft für Informatik, Bonn [cit. on p. 339].

Hebig, Kimberly, Andreas Winter, Dilshodbek Kuryazov, and Alexander Sandau [2018]. "Development of a catalog describing and classifying mobility services in the NEMo project". In: *Environmental Informatics: Techniques and Trends – Adjunct Proceedings of the 32nd EnviroInfo conference*. Ed. by Hans-Joachim Bungartz, Dieter Kranzlmüller, Volker Weinberg, Jens Weismüller, and Volker Wohlgemuth. Berichte aus der Umweltinformatik. Shaker, Herzogenrath, pp. 287–292 [cit. on p. 339].

Harrison, Robert, Daniel Vera, and Bilal Ahmad [2016]. "Engineering Methods and Tools for Cyber-Physical Automation Systems". In: *Proceedings of the IEEE* 104.5, pp. 973–985 [cit. on p. 343].

Artikov, Muzaffar, Dilshodbek Kuryazov, and Andreas Winter [2019]. "Towards Model-driven IoT Maintenance". In: *Softwaretechnik-Trends* 39.2, pp. 49–50 [cit. on p. 343].

*Smart Modeling* [2020]. Carl von Ossietzky University of Oldenburg and Urgench branch of Tashkent University of Information Technologies named after Muhammed al-Khorazmiy. URL: https://smart-modeling.ubtuit.uz/ [visited on 01/15/2020] [cit. on p. 343].

Knuth, Donald E. [2011]. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, Boston [cit. on p. 381].

## References by Author Name, Year, and Title


Aalst, Wil M. P. van der, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros [2003]. "Workflow Patterns". In: *Distributed and Parallel Databases* 14.1, pp. 5–51 [cit. on p. 124].

Abouzahra, Anas, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault [2005]. "A practical approach to bridging domain specific languages with UML profiles". In: *Proceedings of the 4th OOPSLA Workshop on Best Practices for Model Driven Software Development*. URL: https://www.s23m.com/oopsla2005/bezivin1.pdf [cit. on p. 141].

*Acceleo* [2020]. Eclipse Foundation. URL: http://www.eclipse.org/acceleo/ [visited on 01/15/2020] [cit. on p. 212].

Akyol, Ali, Jantje Halberstadt, Kimberly Hebig, Jan Jelschen, Andreas Winter, Alexander Sandau, and Jorge Marx Gómez [2017]. "Flexible Software Support for Mobility Services". In: *Informatik 2017. Beitragsband der 47. Jahrestagung der Gesellschaft für Informatik*. Ed. by Maximilian Eibl, Martin Gaedke, and Cornelia Winter. Gesellschaft für Informatik, Bonn [cit. on p. 339].

Alvaro, Alexandre, Daniel Lucrédio, Vinicius Cardoso Garcia, Antonio Francisco do Prado, Luis Carlos Trevelin, and Eduardo Santana de Almeida [2003]. "Orion-RE: a component-based software reengineering environment". In: *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 248–257 [cit. on p. 73].

Amelunxen, Carsten, Felix Klar, Alexander Königs, Tobias Rötschke, and Andy Schürr [2008]. "Metamodel-based tool integration with MOFLON". In: *Proceedings of the 30th International Conference on Software Engineering*. ACM Press, New York, pp. 807–810 [cit. on p. 76].

Amyot, Daniel, Hanna Farah, and Jean-François Roy [2006]. "Evaluation of development tools for domain-specific modeling languages". In: *Proceedings of the 5th International Workshop on System Analysis and Modeling: Language Profiles*. Ed. by Reinhard Gotzhein and Rick Reed. Lecture Notes in Computer Science 4320. Springer, Berlin, Heidelberg, pp. 183–197 [cit. on p. 144].

*Apache Camel* [2020]. Apache Software Foundation. URL: http://camel.apache.org/ [visited on 01/15/2020] [cit. on p. 193].

*Apache Felix* [2020]. Apache Software Foundation. URL: http://felix.apache.org/ [visited on 01/15/2020] [cit. on p. 106].

*Apache Maven* [2020]. Apache Software Foundation. URL: https://maven.apache.org/ [visited on 01/15/2020] [cit. on pp. 195, 199, 235].

*Apache Tuscany* [2016]. Apache Software Foundation. URL: http://tuscany.apache.org/ [visited on 01/15/2020] [cit. on pp. 106, 237].

*Apache Velocity* [2020]. Apache Software Foundation. URL: http://velocity.apache.org/ [visited on 01/15/2020] [cit. on p. 236].

Armengaud, Eric, Markus Zoier, Andreas Baumgart, Matthias Biehl, Dejiu Chen, Gerhard Griessnig, Christian Hein, Tom Ritter, and Ramin Tavakoli Kolagari [2011]. "Model-based toolchain for the efficient development of safety-relevant automotive embedded systems". In: *Proceedings of the SAE World Congress and Exhibition*. SAE International, Warrendale. URL: `https://saemobilus.sae.org/content/2011-01-0056` [cit. on p. 77].

Artikov, Muzaffar, Dilshodbek Kuryazov, and Andreas Winter [2019]. "Towards Model-driven IoT Maintenance". In: *Softwaretechnik-Trends* 39.2, pp. 49–50 [cit. on p. 343].

Asplund, Fredrik and Martin Törngren [2015]. "The discourse on tool integration beyond technology, a literature survey". In: *Journal of Systems and Software* 106, pp. 117–131 [cit. on pp. 39, 49, 52, 64].

Aßmann, Uwe, Steffen Zschaler, and Gerd Wagner [2006]. "Ontologies, Meta-models, and the Model-Driven Paradigm". In: *Ontologies for Software Engineering and Software Technology*. Ed. by Coral Calero, Francisco Ruiz, and Mario Piattini. Springer, Berlin, Heidelberg, pp. 249–273 [cit. on p. 133].

Atkinson, Colin and Thomas Kühne [2001]. "The essence of multilevel metamodeling". In: *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by Martin Gogolla and Cris Kobryn. Lecture Notes in Computer Science 2185. Springer, Berlin, Heidelberg, pp. 19–33 [cit. on p. 136].

Atkinson, Colin and Thomas Kühne [2003]. "Model-driven development: a metamodeling foundation". In: *IEEE Software* 20.5, pp. 36–41 [cit. on pp. 129, 136].

Atzori, Luigi, Antonio Iera, and Giacomo Morabito [2010]. "The Internet of Things: A survey". In: *Computer Networks* 54.15, pp. 2787–2805 [cit. on p. 6].

Baumgart, Andreas [2010]. "A common meta-model for the interoperation of tools with heterogeneous data models". In: *Proceedings of the 3rd Workshop on Model-Driven Tool & Process Integration*. Ed. by Christian Hein, Michael Wagner, Roland Mader, Andreas Keis, and Eric Armengaud. Fraunhofer, Stuttgart, pp. 31–40 [cit. on p. 77].

Baumgart, Andreas, Christian Ellen, Stefan Farfeleder, Rainer Koopmann, Markus Oertel, and Philip Rehkop [2012]. "A reference technology platform with common interfaces for distributed heterogeneous data". In: *Proceedings of the Embedded World 2012 Exhibition and Conference*. WEKA-Fachmedien, Haar [cit. on p. 77].

Baxter, Ira D., Christopher Pidgeon, and Michael Mehlich [2004]. "DMS: Program transformations for practical scalable software evolution". In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 625–634 [cit. on pp. 73, 144].

Beck, Kent, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland,

and Dave Thomas [2001]. *Manifesto for Agile Software Development*. URL: http://agilemanifesto.org/ [visited on 01/15/2020] [cit. on p. 7].

Beck, Kent. [2004]. *Extreme Programming Explained: Embrace Change*. 2nd ed. Addison-Wesley, Boston [cit. on p. 4].

Becker, Christian and Uwe Kaiser [2014]. "Applikationswissen in der Sprachkonvertierung am Beispiel des COBOL-Java-Converters CoJaC." In: *Softwaretechnik-Trends* 34.2 [cit. on p. 268].

Becker, Christian and Uwe Kaiser [2016]. "Toolbasierte Software-Migration nach Plan". In: *Softwaretechnik-Trends* 36.2 [cit. on p. 268].

Bentley, Jon [1986]. "Programming pearls". In: *Communications of the ACM* 29.8, pp. 711–721 [cit. on p. 140].

Bergey, John K., Scott R. Tilley, Steven Woods, Dennis B. Smith, and Nelson W. Weiderman [1999]. *Why reengineering projects fail*. Technical Report. Software Engineering Institute, Carnegie Mellon University, Pittsburgh [cit. on p. 5].

Bergstra, Jan A. and Paul Klint [1998]. "The discrete time ToolBus — A software coordination architecture". In: *Science of Computer Programming* 31.2-3, pp. 205–229 [cit. on p. 75].

Bevan, Jennifer, E. James Whitehead, Sunghun Kim, and Michael W. Godfrey [2005]. "Facilitating software evolution research with kenyon". In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, New York, pp. 177–186 [cit. on p. 70].

Bézivin, Jean [2006]. "Model driven engineering: an emerging technical space". In: *Generative and Transformational Techniques in Software Engineering*. Ed. by Ralf Lämmel, João Saraiva, and Joost Visser. Lecture Notes in Computer Science 4143. Springer, Berlin, Heidelberg, pp. 36–64 [cit. on p. 142].

Bézivin, Jean, Hugo Bruneliere, Frédéric Jouault, and Ivan Kurtev [2005]. "Model Engineering Support for Tool Interoperability". In: *Proceedings of the 4th UML / MoDELS Workshop in Software Model Engineering*. URL: https://web.archive.org/web/20070812181354fw%7B%5C_%7D/http://www.planetmde.org/wisme-2005/ModelEngineeringSupportForToolInteroperability.pdf [cit. on p. 76].

Bézivin, Jean and Ivan Kurtev [2006]. *Model-based technology integration with the technical space concept*. Tech. rep. hal-00483587. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on pp. 142, 143].

Bhagat, Jiten, Franck Tanoh, Eric Nzuobontane, Thomas Laurent, Jerzy Orlowski, Marco Roos, Katy Wolstencroft, Sergejs Aleksejevs, Robert Stevens, Steve Pettifer, Rodrigo Lopez, and Carole A. Goble [2010]. "BioCatalogue: A universal catalogue of web services for the life sciences". In: *Nucleic Acids Research* 38, W689–W694 [cit. on p. 88].

Bieberstein, Norbert, Robert G. Laird, Keith Jones, and Tilak Mitra [2008]. *Executing SOA: A Practical Guide for the Service-Oriented Architect*. IBM Press, Indianapolis [cit. on p. 125].

Biehl, Matthias [2012]. *Semantic Anchoring of TIL*. Technical Report. Royal Institute of Technology, Stockholm. URL: `https://sites.google.com/site/mattbiehl/research/publications/semantics.pdf` [cit. on p. 84].

Biehl, Matthias [2013]. "A Modeling Language for the Description and Development of Tool Chains for Embedded Systems". Doctoral Thesis. Royal Institute of Technology, Stockholm [cit. on pp. 45, 64, 78, 82, 84].

Biehl, Matthias, Wenqing Gu, and Frédéric Loiret [2012]. "Model-based service discovery and orchestration for OSLC services in tool chains". In: *Proceedings of the 12th International Conference on Web Engineering*. Ed. by Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf. Lecture Notes in Computer Science 7387. Springer, Berlin, Heidelberg, pp. 283–290 [cit. on pp. 83–85].

Biehl, Matthias, Jiarui Hong, and Frederic Loiret [2012]. "Automated Construction of Data Integration Solutions for Tool Chains". In: *Proceedings of the 7th International Conference on Software Engineering Advances*. IARIA, Wilmington, pp. 102–111 [cit. on pp. 84, 85].

Bischopink, Christopher, Stephan Bogs, Hauke Fischer, Hannah Meyer, Felix Kempa, Nancy Kramer, Thomas Sprock, and Lisa Ripke [2018]. *Projektgruppe DORI*. Project Report. Carl von Ossietzky University, Oldenburg [cit. on pp. 324, 326, 327, 342].

Black, Andrew P., Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker [2018]. *Pharo by Example 5*. Square Bracket Associates [cit. on p. 73].

Boehm, Barry [2002]. "Get ready for agile methods, with care". In: *Computer* 35.1, pp. 64–69 [cit. on p. 4].

Boehm, Barry [2006]. "A view of 20th and 21st century software engineering". In: *Proceedings of the 28th International Conference on Software Engineering*. ACM, New York [cit. on pp. 18, 39–41].

Booch, Grady, James Rumbaugh, and Ivar Jacobson [1999]. *The Unified Modeling Language User Guide*. 5th ed. Object Technology Series. Addison-Wesley, Boston [cit. on pp. 43, 102, 103].

Booth, David, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard [2004]. *Web Services Architecture*. URL: `https://www.w3.org/TR/ws-arch/` [visited on 01/15/2020] [cit. on pp. 34, 115, 119].

Borchers, Jens [1996]. "Reengineering-Factory — Erfolgsmechanismen großer Reengineering-Maßnahmen". In: *Softwarewartung und Reengineering*. Ed. by Franz Lehner. Information Engineering und IV-Controlling. Deutscher Universitätsverlag, Wiesbaden, pp. 19–29 [cit. on pp. 4, 5, 8].

Bott, Frank, ed. [1989]. *Eclipse, an integrated project support environment*. Vol. 14. IEE Computing Series. Peter Peregrinus, Hitchin [cit. on p. 40].

Bourque, Pierre and Richard E. Fairley, eds. [2014]. *Guide to the Software Engineering Body of Knowledge (Swebok(r)): Version 3.0*. IEEE, Los Alamitos [cit. on pp. 31, 101, 103].

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer [2012]. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, Williston [cit. on p. 136].

Brand, Mark G. J. van den, Arie van Deursen, Jan Heering, Heyco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser [2001]. "The ASF+SDF Meta-environment: A Component-Based Language Development Environment". In: *Proceedings of the 10th International Conference on Compiler Construction*. Ed. by Reinhard Wilhelm. Lecture Notes in Computer Science 2027. Springer, Berlin, Heidelberg, pp. 365–370 [cit. on pp. 75, 143].

Breivold, Hongyu Pei and Magnus Larsson [2007]. "Component-based and service-oriented software engineering: Key concepts and principles". In: *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, Los Alamitos, pp. 13–20 [cit. on p. 112].

Brodie, Michael L. [1984]. "On the Development of Data Models". In: *On conceptual modelling*. Ed. by Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt. Topics in Information Systems. Springer, New York. Chap. 2, pp. 19–47 [cit. on p. 200].

Brodie, Michael L. and Michael Stonebraker [1995]. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco [cit. on p. 3].

Brooks, Frederick P. [1987]. "No Silver Bullet: Essence and Accidents of Software Engineering". In: *Computer* 20.4, pp. 10–19 [cit. on pp. 46, 56].

Brown, Alan, David Carney, Patricia Oberndorf, and Marvin Zelkowitz [1993]. *Reference Model for Project Support Environments (Version 2.0)*. Tech. rep. CMU/SEI-93-TR-23, NIST SP 500-213. Software Engineering Institute, National Institute of Standards and Technology [cit. on pp. 40, 41, 43, 58, 91].

Brown, Alan W. [1988]. "Integrated project support environments". In: *Information & Management* 15.3, pp. 125–134 [cit. on p. 40].

Brown, Alan W., Peter H. Feiler, and Kurt C. Wallnau [1992]. "Past and Future Models of CASE Integration". In: *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*. IEEE, Los Alamitos, pp. 36–45 [cit. on pp. 42, 43].

Brown, Alan W. and John A. McDermid [1992]. "Learning from IPSE's mistakes". In: *IEEE Software* 9.2, pp. 23–28 [cit. on pp. x, 41, 45, 48, 51–53, 64–66, 71].

Broy, Manfred [2018]. "Yesterday, Today, and Tomorrow: 50 Years of Software Engineering". In: *IEEE Software* 35.5, pp. 38–43 [cit. on p. 3].

Bruneliere, Hugo, Jordi Cabot, Cauê Clasen, Frédéric Jouault, and Jean Bézivin [2010a]. "Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools". In: *Proceedings of the 6th European Conference on Modelling Foun-*

*dations and Applications*. Ed. by Thomas Kühne, Bran Selic, Marie-Pierre Gervais, and François Terrier. Lecture Notes in Computer Science 6138. Springer, Berlin, Heidelberg, pp. 32–47 [cit. on p. 76].

Bruneliere, Hugo, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot [2014]. "Modisco: A model driven reverse engineering framework". In: *Information and Software Technology* 56.8, pp. 1012–1032 [cit. on p. 73].

Bruneliere, Hugo, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot [2010b]. "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, pp. 173–174 [cit. on p. 73].

Burger, Erik, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe [2016]. "View-based model-driven software development with ModelJoin". In: *Software and Systems Modeling* 15.2, pp. 473–496 [cit. on p. 54].

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal [1996]. *Pattern-Oriented Software Architecture: A System of Patterns*. Vol. 1. Wiley, Chichester [cit. on pp. 53, 88].

*Business Process Model and Notation (BPMN) Version 2.0* [2011]. Object Management Group. URL: `http://www.omg.org/spec/BPMN/2.0/PDF/` [visited on 01/15/2020] [cit. on p. 124].

Buxton, John N. and Vic Stenning [1980]. *Requirements for Ada Programming Support Environments: "Stoneman"*. Tech. rep. ADA100404. US Department of Defense [cit. on pp. 39, 40].

Cerny, Tomas, Michael Jeffry Donahoo, and Michal Trnka [2018]. "Contextual Understanding of Microservice Architecture: Current and Future Directions". In: *ACM SIGAPP Applied Computing Review* 17.4, pp. 29–45 [cit. on p. 6].

Chapman, Martin, Mike Edwards, Michael Beisiegel, Anish Karmarkar, Sanjay Patil, and Michael Rowley [2011]. *Service Component Architecture Assembly Model Specification Version 1.1*. Standard. OASIS, Burlington. URL: `http://docs.oasis-open.org/opencsa/sca-assembly/sca-assembly-spec-v1.1-csprd03.html` [cit. on p. 237].

Chau, Patrick Y. K. [1996]. "An empirical investigation on factors affecting the acceptance of CASE by systems developers". In: *Information and Management* 30.6, pp. 269–280 [cit. on p. 42].

Chen, Yih-Farn, Michael Y. Nishimoto, and Chittoor V. Ramamoorthy [1990]. "The C information abstraction system". In: *IEEE Transactions on Software Engineering* 16.3, pp. 325–334 [cit. on p. 70].

Chignell, Mark H. [1990]. "A taxonomy of user interface terminology". In: *ACM SIGCHI Bulletin* 21.4, p. 27 [cit. on p. 325].

Chikofsky, Elliot J. [1988]. "Guest Editor's Introduction: Software Technology People Can Really Use". In: *IEEE Software* 5.2, pp. 8–10 [cit. on p. 41].

Clements, Paul and Linda Northrop [2002]. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston [cit. on p. 141].

*Code Conventions for the Java Programming Language* [1999]. Oracle. URL: http://www.oracle.com/technetwork/java/index-135089.html [visited on 03/17/2017] [cit. on p. 268].

Cohen, Shy [2007]. "Ontology and Taxonomy of Services in a Service-Oriented Architecture". In: *The Architecture Journal (Microsoft Online Publication)* 11. URL: https://web.archive.org/web/20110127030807/http://msdn.microsoft.com/en-us/library/bb491121.aspx [cit. on p. 117].

*COM: Component Object Model Technologies* [2018]. Microsoft Corporation. URL: https://docs.microsoft.com/en-us/windows/win32/com/component-object-model--com--portal [visited on 01/15/2020] [cit. on pp. 105, 237].

Combemale, Benoit, Betty H.C. Cheng, Ana Moreira, Jean-Michel Bruel, and Jeff Gray [2016]. "Modeling for Sustainability". In: *Proceedings of the 8th International Workshop on Modeling in Software Engineering*. ACM, New York, pp. 62–66 [cit. on p. 309].

Conner, Mike, Nurcan Coskun, Scott Danforth, Larry Loucks, Andy Martin, Larry Raper, and Roger Sessions [1992]. "Developing language neutral class libraries with the System Object Model (SOM)". In: *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*. ACM, New York, pp. 191–193 [cit. on p. 99].

Cook, Steve [2004]. "Domain-Specific Modeling and Model Driven Architecture". In: *The MDA Journal: Model Driven Architecture Straight from the Masters*. Meghan Kiffer, Tampa. Chap. 5 [cit. on p. 140].

*CORBA* [2020]. Object Management Group. URL: http://www.corba.org/ [visited on 01/15/2020] [cit. on pp. 34, 99, 118].

*CORBA Component Model (CCM) 4.0* [2006]. Object Management Group. URL: http://www.omg.org/spec/CCM [visited on 01/15/2020] [cit. on pp. 105, 237].

Cordy, James R. [2004]. "TXL - A Language for Programming Language Tools and Applications". In: *Electronic Notes in Theoretical Computer Science* 110, pp. 3–31 [cit. on p. 144].

Crnkovic, Ivica and Magnus Peter Henrik Larsson, eds. [2002]. *Building Reliable Component-based Software Systems*. Artech House, Boston [cit. on p. 100].

Crnkovic, Ivica, Severine Sentilles, Aneta Vulgarakis, and Michel R. V. Chaudron [2011]. "A Classification Framework for Software Component Models". In: *IEEE Transactions on Software Engineering* 37.5, pp. 593–615 [cit. on pp. 102, 104, 106].

Crone, Thomas [2013]. "Software-Evolutions-Services zur Berechnung und Visualisierung von Metriken". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Cyganiak, Richard, David Wood, and Markus Lanthaler [2014]. *RDF 1.1 Concepts and Abstract Syntax*. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ [visited on 01/15/2020] [cit. on p. 65].

Czarnecki, Krzysztof [2004]. "Overview of generative software development". In: *International Workshop on Unconventional Programming Paradigms, Revised Selected and Invited Papers*. Ed. by Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel. Lecture Notes in Computer Science 3566. Springer, Berlin, Heidelberg, pp. 326–341 [cit. on pp. 140, 141, 144].

Czarnecki, Krzysztof and Simon Helsen [2003]. "Classification of model transformation approaches". In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. URL: https://s23m.com/oopsla2003/czarnecki.pdf [cit. on pp. 139, 140].

Czarnecki, Krzysztof and Simon Helsen [2006]. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3, pp. 621–645 [cit. on pp. 139, 140].

Czeranski, Jörg, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang, Jean-François Girard, and Martin Würthner [2000]. "Data exchange in Bauhaus". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 293–295 [cit. on p. 70].

Dahl, Ole-Johan [2004]. "The Birth of Object Orientation: the Simula Languages". In: *From Object-Orientation to Formal Methods*. Ed. by Olaf Owe, Stein Krogdahl, and Tom Lyche. Lecture Notes in Computer Science 2635. Springer, Berlin, Heidelberg, pp. 15–25 [cit. on pp. 3, 99].

Dahl, Ole-Johan, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare [1972]. *Structured programming*. Academic Press, London [cit. on pp. 39, 99, 187].

Dale, Nell and Henry M. Walker [1996]. *Abstract Data Types: Specifications, Implementations, and Applications*. D. C. Heath and Company, Lexington [cit. on p. 166].

de Sousa Saraiva, João and Alberto Rodrigues da Silva [2008]. "Evaluation of MDE Tools from a Metamodeling Perspective". In: *Journal of Database Management* 19.4, pp. 50–75 [cit. on p. 144].

Decker, Gero, Oliver Kopp, and Alistair Barros [2008]. "An Introduction to Service Choreographies (Servicechoreographien – eine Einführung)". In: *it - Information Technology* 50.2, pp. 122–127 [cit. on p. 125].

Decker, Gero, Oliver Kopp, Frank Leymann, and Mathias Weske [2007]. "BPEL4Chor: Extending BPEL for Modeling Choreographies". In: *Proceedings of the 5th International Conference on Web Services*. IEEE, Los Alamitos, pp. 296–303 [cit. on p. 125].

Deißenböck, Florian, Lars Heinemann, Benjamin Hummel, and Elmar Juergens [2010]. "Flexible architecture conformance assessment with ConQAT". In: *Proceedings*

*of the 32nd International Conference on Software Engineering*. ACM, New York, pp. 247–250 [cit. on p. 72].

Deißenböck, Florian, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas Parareda, and Markus Pizka [2008]. "Tool Support for Continuous Quality Control". In: *IEEE Software* 25.5, pp. 60–67 [cit. on p. 72].

Deursen, Arie van, Paul Klint, and Joost Visser [2000]. "Domain-Specific Languages: An Annotated Bibliography". In: *ACM SIGPLAN Notices* 35.6, pp. 26–36 [cit. on p. 140].

Dijkstra, Edsger W. [1968]. "Go To Statement Considered Harmful". In: *Communications of the ACM* 11.3, pp. 147–148 [cit. on p. 39].

*DMS Software Reengineering Toolkit* [2020]. Semantic Designs. URL: `http://www.semdesigns.com/Products/DMS/DMSToolkit.html` [visited on 01/15/2020] [cit. on p. 73].

Dowson, Mark [1987]. "ISTAR—an integrated project support environment". In: *ACM SIGPLAN Notices* 22.1, pp. 27–33 [cit. on p. 40].

Draheim, Dirk [2010]. *Business Process Technology: A Unified View on Business Processes, Workflows and Enterprise Applications*. Springer, Berlin, Heidelberg [cit. on p. 123].

Ducasse, Stéphane, Nicolas Anquetil, Muhammad Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Gîrba [2011]. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Tech. rep. hal-00646884. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on pp. 65, 67].

Ducasse, Stéphane, Tudor Gîrba, and Oscar Nierstrasz [2005]. "Moose: an Agile Reengineering Environment". In: *ACM SIGSOFT Software Engineering Notes* 30.5, pp. 99–102 [cit. on p. 73].

Ducasse, Stéphane, Michele Lanza, and Sander Tichelaar [2000]. "MOOSE: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems". In: *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*. Ed. by Ian Ferguson, Jonathan Gray, and Louise Scott, pp. 24–30 [cit. on p. 73].

Ebert, Christof, Gerd Hoefner, and V. S. Mani [2015]. "What Next? Advances in Software- Driven Industries". In: *IEEE Software* 32.1, pp. 22–28 [cit. on p. 6].

Ebert, Jürgen [1987]. "A versatile data structure for edge-oriented graph algorithms". In: *Communications of the ACM* 30.6, pp. 513–519 [cit. on p. 143].

Ebert, Jürgen [2008]. "Metamodels Taken Seriously: The TGraph Approach". In: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*. Ed. by Kostas Kontogiannis, Christos Tjortjis, and Andreas Winter. IEEE, Los Alamitos, p. 2 [cit. on p. 143].

Ebert, Jürgen and Angelika Franzke [1995]. "A declarative approach to graph based modeling". In: *Proceedings of the 20th International Workshop on Graph-Theoretic*

*Concepts in Computer Science*. Ed. by Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer. Lecture Notes in Computer Science 903. Springer, Berlin, Heidelberg, pp. 38–50 [cit. on pp. 211, 236].

Ebert, Jürgen and Tassilo Horn [2012]. "GReTL: an extensible, operational, graph-based transformation language". In: *Software & Systems Modeling* 13.1. Ed. by Jordi Cabot and Eelco Visser, pp. 301–321 [cit. on p. 236].

Ebert, Jürgen, Bernt Kullbach, Volker Riediger, and Andreas Winter [2002]. "GUPRO - Generic understanding of programs: An overview". In: *Proceedings of the 1st International Conference on Graph Transformation*. Electronic Notes in Theoretical Computer Science 72.2. Elsevier, Amsterdam, pp. 59–68 [cit. on p. 71].

Ebert, Jürgen, Volker Riediger, and Andreas Winter [2008]. "Graph Technology in Reverse Engineering: The TGraph Approach." In: *Proceedings of the 10th Workshop Software Reengineering*. Ed. by Rainer Gimnich, Uwe Kaiser, Jochen Quante, and Andreas Winter. Lecture Notes in Informatics P-126. Gesellschaft für Informatik, Bonn, pp. 67–81 [cit. on pp. 17, 193, 236, 293].

*Eclipse Modeling Project* [2020]. Eclipse Foundation. URL: `https://projects.eclipse.org/projects/modeling` [visited on 01/15/2020] [cit. on p. 40].

Eden, Amnon H. and Tom Mens [2006]. "Measuring software flexibility". In: *IEE Proceedings - Software* 153.3, pp. 113–125 [cit. on pp. 7, 330].

Edwards, Mike and Martin Chapman [2016]. *Service Component Architecture Assembly Technical Committee*. URL: `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sca-assembly` [visited on 01/15/2020] [cit. on p. 100].

Electronic Industries Association [1994]. *CDIF Integrated Meta-model Foundation Subject Area*. Interim Standard EIA/IS-111. Electronic Industries Association, Arlington [cit. on p. 67].

ElShazly, Hassan and Varun Grover [1993]. "A Study on the Evaluation of CASE Technology". In: *Journal of Information Technology Management* IV.1, pp. 15–24 [cit. on p. 42].

Endres, Albert [1996]. "A Synopsis of Software Engineering History: The Industrial Perspective". In: *Proceedings of the Dagstuhl Seminar 9635 on History of Software Engineering*. Ed. by Andreas Brennecke and Reinhard Keil-Slawik. Dagstuhl Seminar Reports 153. Leibniz-Zentrum für Informatik, Wadern, pp. 20–24 [cit. on pp. 40, 42].

*Enterprise JavaBeans* [2019]. Oracle Corporation. URL: `http://www.oracle.com/technetwork/java/javaee/ejb/` [visited on 01/15/2020] [cit. on pp. 100, 105, 237].

*Equinox* [2020]. Eclipse Foundation. URL: `http://www.eclipse.org/equinox/` [visited on 01/15/2020] [cit. on p. 106].

Erdmenger, Uwe and Denis Uhlig [2011]. "Ein Translator für die COBOL-Java-Migration". In: *Softwaretechnik-Trends* 31.2 [cit. on pp. 45, 68].

Erdweg, Sebastian, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning [2013]. "The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge". In: *Proceedings of the 6th International Conference on Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Lecture Notes in Computer Science 8225. Springer, Cham, pp. 197–217 [cit. on pp. 141, 144].

Erl, Thomas [2005]. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall, Upper Saddle River [cit. on pp. 3, 112, 115, 116, 118, 122–125].

Erl, Thomas [2007]. *SOA Principles of Service Design*. Prentice Hall, Upper Saddle River [cit. on pp. 112, 114, 115, 120, 121].

Eysholdt, Moritz and Heiko Behrens [2010]. "Xtext: implement your language faster than the quick and dirty way". In: *Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, New York, pp. 307–309 [cit. on p. 143].

*Fabric3* [2016]. Metaform Systems. URL: `https://web.archive.org/web/20180428063719/http://www.fabric3.org:80/` [visited on 01/15/2020] [cit. on pp. 106, 239].

Ferenc, Rudolf, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy [2002]. "Columbus – Reverse Engineering Tool and Schema for C++". In: *Proceedings of the 18th International Conference on Software Maintenance*. IEEE, Los Alamitos, pp. 172–181 [cit. on p. 71].

Ferenc, Rudolf, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy [2001]. "Towards a standard schema for C/C++". In: *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 49–58 [cit. on p. 268].

Fielding, Roy T. and Richard N. Taylor [2002]. "Principled design of the modern Web architecture". In: *ACM Transactions on Internet Technology* 2.2, pp. 115–150 [cit. on pp. 34, 68, 90].

Finnigan, Patrick J., Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kenny Wong [1997]. "The software bookshelf". In: *IBM Systems Journal* 36.4, pp. 564–593 [cit. on pp. 89, 90].

Fowler, Martin [2005a]. *Language Workbenches: The Killer-App for Domain Specific Languages?* URL: `http://www.martinfowler.com/articles/languageWorkbench.html` [visited on 01/15/2020] [cit. on pp. 141, 212].

Fowler, Martin [2005b]. *ServiceOrientedAmbiguity*. URL: `http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html` [visited on 01/15/2020] [cit. on pp. 118, 122].

Fowler, Martin [2010]. *Domain-Specific Languages*. Addison-Wesley, Boston [cit. on p. 141].

Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts [1999]. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston [cit. on p. 31].

Frijters, Jeroen [2014]. *IKVM.NET Home Page*. URL: `http://www.ikvm.net/` [visited on 01/15/2020] [cit. on p. 88].

Fuggetta, Alfonso [1993]. "A Classification of CASE Technology". In: *Computer* 26.12, pp. 25–38 [cit. on pp. x, 48, 58–61, 68–70, 72, 90].

Fuhr, Andreas, Andreas Winter, Uwe Erdmenger, Tassilo Horn, Uwe Kaiser, Volker Riediger, and Werner Teppe [2012]. "Model-Driven Software Migration - Process Model, Tool Support and Application". In: *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. Ed. by Anca Daniela Ionita, Martin Litoiu, and Grace Lewis. IGI Global, Hershey, pp. 153–184 [cit. on pp. 7, 15, 299].

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides [1995]. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston [cit. on p. 256].

*Gartner Hype Cycle* [2015]. Gartner. URL: `http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp` [visited on 01/15/2020] [cit. on p. 109].

Gašević, Dragan, Dragan Djurić, and Vladan Devedžić [2006]. *Model Driven Architecture and Ontology Development*. Springer, Berlin, Heidelberg [cit. on p. 43].

Ghezzi, Giacomo [2012]. "SOFAS, Software Analysis as a Service. Improving and Rethinking Software Evolution Analysis". Dissertation. University of Zurich [cit. on pp. 64, 78].

Ghezzi, Giacomo and Harald C. Gall [2013]. "A framework for semi-automated software evolution analysis composition". In: *Automated Software Engineering* 20.3, pp. 463–496 [cit. on pp. 4, 6, 8, 78].

Ghosh, Debasish [2010]. *DSLs in Action*. Manning, Shelter Island [cit. on p. 141].

*GlassFish Server* [2020]. Oracle. URL: `http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html` [visited on 01/15/2020] [cit. on p. 106].

Godfrey, Michael W. and Lijie Zou [2005]. "Using origin analysis to detect merging and splitting of source code entities". In: *IEEE Transactions on Software Engineering* 31.2, pp. 166–181 [cit. on p. 70].

Goldstein, Robert C. and Veda C. Storey [1994]. "Materialization". In: *IEEE Transactions on Knowledge and Data Engineering* 6.5, pp. 835–842 [cit. on p. 136].

Gönczy, László, Ábel Hegedüs, and Dániel Varró [2011]. "Methodologies for model-driven development and deployment: an overview". In: *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software*

*Engineering for Service-Oriented Computing*. Ed. by Martin Wirsing and Matthias Hölzl. Lecture Notes in Computer Science 6582. Springer, Berlin, Heidelberg, pp. 541–560 [cit. on p. 76].

Gonzalez-Perez, Cesar and Brian Henderson-Sellers [2008]. *Metamodelling for Software Engineering*. Wiley, Chichester [cit. on p. 136].

*Google Maps Directions API* [2020]. Google. URL: `https://developers.google.com/maps/documentation/directions/` [visited on 01/15/2020] [cit. on p. 315].

*Google Places API* [2020]. Google. URL: `https://developers.google.com/places/` [visited on 01/15/2020] [cit. on p. 315].

Gorton, Ian, David Thurman, and Judi Thomson [2003]. "Next generation application integration: challenges and new approaches". In: *Proceedings of the 27th Annual International Computer Software and Applications Conference*. IEEE, Los Alamitos, pp. 576–581 [cit. on p. 3].

*Graphical Editing Framework* [2020]. Eclipse Foundation. URL: `https://eclipse.org/gef/` [visited on 01/15/2020] [cit. on p. 143].

*Graphical Modeling Framework* [2020]. Eclipse Foundation. URL: `http://www.eclipse.org/modeling/gmp/` [visited on 01/15/2020] [cit. on pp. 143, 213].

*Graphiti* [2020]. Eclipse Foundation. URL: `https://eclipse.org/graphiti/` [visited on 01/15/2020] [cit. on pp. 143, 213].

Gray, Jim [2006]. "A conversation with Werner Vogels". In: *Queue* 4.4. Ed. by Charlene O'Hanlon, pp. 14–22 [cit. on p. 122].

Greenfield, Jack and Keith Short [2004]. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Chichester [cit. on p. 143].

Grieger, Marvin and Masud Fazal-Baqaie [2015]. "Towards a Framework for the Modular Construction of Situation-Specific Software Transformation Methods". In: *Softwaretechnik-Trends* 35.2, pp. 41–42 [cit. on p. 7].

Gürdür, Didem, Fredrik Asplund, and Jad El-Khoury [2016]. "Measuring Tool Chain Interoperability in Cyber-physical Systems". In: *Proceedings of the 11th System of Systems Engineering Conference*. IEEE, Los Alamitos [cit. on p. 47].

Guttman, Michael and John Parodi [2006]. *Real-Life MDA: Solving Business Problems with Model Driven Architecture*. Morgan Kaufmann, San Francisco [cit. on p. 138].

Hadley, Marc [2009]. *Web Application Description Language*. Standard. W3C, Cambridge [cit. on p. 80].

Hamilton, Graham [1997]. *JavaBeans 1.01*. Specification. Sun Microsystems, Mountain View. URL: `http://www.oracle.com/technetwork/articles/javaee/spec-136004.html` [cit. on p. 100].

Harrison, Robert, Daniel Vera, and Bilal Ahmad [2016]. "Engineering Methods and Tools for Cyber-Physical Automation Systems". In: *Proceedings of the IEEE* 104.5, pp. 973–985 [cit. on p. 343].

Hebig, Kimberly, Andreas Winter, Dilshodbek Kuryazov, and Alexander Sandau [2018]. "Development of a catalog describing and classifying mobility services in the NEMo project". In: *Environmental Informatics: Techniques and Trends – Adjunct Proceedings of the 32nd EnviroInfo conference*. Ed. by Hans-Joachim Bungartz, Dieter Kranzlmüller, Volker Weinberg, Jens Weismüller, and Volker Wohlgemuth. Berichte aus der Umweltinformatik. Shaker, Herzogenrath, pp. 287–292 [cit. on p. 339].

Heckelmann, Kristina [2010]. "Abbildung von Ecore nach grUML". Bachelor's Thesis. University of Koblenz [cit. on p. 211].

Hein, Christian, Tom Ritter, and Michael Wagner [2009]. "Model-Driven Tool Integration with ModelBus". In: *Proceedings of the 1st International Workshop on Future Trends of Model-Driven Development*. Ed. by Slimane Hammoudi and Luís Ferreira Pires. INSTICC, Setubal, pp. 35–39 [cit. on p. 77].

Heineman, George T. and William T. Councill, eds. [2001]. *Component-based software engineering: putting the pieces together*. Addison-Wesley, Boston [cit. on pp. 100, 102, 106].

Heinemann, Lars, Benjamin Hummel, and Daniela Steidl [2014]. "Teamscale: software quality control in real-time". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, pp. 592–595 [cit. on p. 72].

Henning, Michi [2006]. "The Rise and Fall of CORBA". In: *Queue* 4.5, pp. 28–34 [cit. on p. 118].

Hentrich, Carsten and Uwe Zdun [2009]. "A pattern language for process execution and integration design in service-oriented architectures". In: *Transactions on Pattern Languages of Programming I*. Ed. by James Noble and Ralph Johnson. Lecture Notes in Computer Science 5770. Springer, Berlin, Heidelberg, pp. 136–191 [cit. on p. 116].

Hesse, Wolfgang and Heinrich C. Mayr [2008]. "Modellierung in der Softwaretechnik: eine Bestandsaufnahme". In: *Informatik-Spektrum* 31.5, pp. 377–393 [cit. on pp. 65, 132, 133].

Hildebrandt, Stephan, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr [2013]. "A survey of triple graph grammar tools". In: *Proceedings of the 2nd Workshop on Bidirectional Transformations*. Ed. by Perdita Stevens and James F. Terwilliger. Electronic Communications of the EASST 57. European Association of Software Science and Technology [cit. on p. 143].

Hofstede, Arthur H. M. ter, Wil M. P. van der Aalst, Michael Adams, and Nick Russell, eds. [2010]. *Modern Business Process Automation: YAWL and Its Support Environment*. Springer, Berlin, Heidelberg [cit. on pp. 123, 124, 193].

Hohpe, Gregor and Bobby Woolf [2004]. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston [cit. on pp. 124, 193].

Hollingsworth, David [1995]. *The Workflow Reference Model*. Standard TC00-1003. Workflow Management Coalition, Winchester [cit. on p. 123].

Hollingsworth, David [2004]. "The Workflow Reference Model: 10 Years On". In: *Workflow Handbook 2004*. Ed. by Layna Fischer. Future Strategies, Lighthouse Point, pp. 295–312 [cit. on p. 125].

Holt, Richard C., Michael W. Godfrey, and Andrew J. Malton [2003]. "The Build / Comprehend Pipelines". In: *Proceedings of the Second ASERC Workshop on Software Architecture*. Alberta Software Engineering Research Consortium. URL: https://plg.uwaterloo.ca/%7B~%7Dmigod/papers/2003/aserc03.pdf [cit. on p. 70].

Holt, Richard C., Andy Schürr, Susan Elliott Sim, and Andreas Winter [2006]. "GXL: A graph-based standard exchange format for reengineering". In: *Science of Computer Programming* 60.2, pp. 149–170 [cit. on p. 65].

Holt, Richard C., Andreas Winter, and Andy Schürr [2000]. "GXL: toward a standard exchange format". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 162–171 [cit. on pp. 5, 201].

Hull, Duncan, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn [2006]. "Taverna: A tool for building and running workflows of services". In: *Nucleic Acids Research* 34, W729–W732 [cit. on p. 87].

IBM [2016]. *Statements of deprecation and general direction: IBM Rational Application Developer for WebSphere Software and the IBM Rational Software Architect Designer program family*. URL: http://www-01.ibm.com/common/ssi/ShowDoc.wss?docURL=/common/ssi/rep_ca/5/897/ENUS216-245/index.html&lang=en&request_locale=en [visited on 01/19/2017] [cit. on p. 239].

*ICT Services* [2016]. URL: http://www.ikts-niedersachsen.de/en [visited on 01/15/2020] [cit. on p. 309].

*IFML: The Interaction Flow Modeling Language* [2017]. Object Management Group. URL: http://www.ifml.org/ [visited on 07/28/2017] [cit. on pp. 323, 325].

ISO/IEC 18384 [2016]. *Information technology – Reference Architecture for Service Oriented Architecture (SOA RA)*. International Standard. International Organization for Standardization, Geneva [cit. on p. 118].

ISO/IEC 25010 [2011]. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. International Standard. International Organization for Standardization, Geneva [cit. on pp. 8, 12].

ISO/IEC 9126-1 [2001]. *Software Engineering – Product Quality – Part 1: Quality Model*. International Standard. International Organisation for Standardization, Geneva [cit. on p. 8].

ISO/IEC/IEEE 24765 [2017]. *Systems and software engineering – Vocabulary*. International Standard. International Organization for Standardization, Geneva [cit. on pp. 8, 44, 47].

Jakumeit, Edgar, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek [2014]. "A survey and comparison of transformation tools based on the transformation tool contest". In: *Science of Computer Programming* 85, Part A, pp. 41–99 [cit. on pp. 140, 144].

Jamshidi, Pooyan, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov [2018]. "Microservices: The Journey So Far and Challenges Ahead". In: *IEEE Software* 35.3, pp. 24–35 [cit. on p. 6].

*JBoss Developer* [2020]. Red Hat. URL: `https://developer.jboss.org` [visited on 01/15/2020] [cit. on p. 106].

Jelschen, Jan [2013]. "Discovery and Description of Software Evolution Services". In: *Softwaretechnik-Trends* 33.2, pp. 59–60 [cit. on pp. 9, 81, 117, 157, 164, 339].

Jelschen, Jan [2014a]. "SENSEI: Software Evolution Service Integration". In: *Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Ed. by Serge Demeyer, David Binkley, and Filippo Ricca. IEEE, Antwerp, pp. 469–472 [cit. on pp. 9, 339].

Jelschen, Jan [2014b]. *The Q-MIG Data Exchange Format*. Project Report. Carl von Ossietzky University, Oldenburg [cit. on p. 15].

Jelschen, Jan [2015]. "Service-Oriented Toolchains for Software Evolution". In: *Proceedings of the 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments*. Ed. by Andreas Winter, Mike Smit, and Muhammad Ali Barbar. IEEE, Los Alamitos, pp. 51–58 [cit. on pp. 9, 339].

Jelschen, Jan, Marion Gottschalk, Mirco Josefiok, Cosmin Pitu, and Andreas Winter [2012]. "Towards applying reengineering services to energy-efficient applications". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 353–358 [cit. on p. 25].

Jelschen, Jan, Christoph Alexander Küpker, Andreas Winter, Alexander Sandau, Benjamin Wagner vom Berg, and Jorge Marx Gómez [2016]. "Towards a Sustainable Software Architecture for the NEMo Mobility Platform". In: *Proceedings of the 30th International Conference on Environmental Informatics – Stability, Continuity, Innovation: Current trends and future perspectives based on 30 years of history*. Ed. by Volker Wohlgemuth, Frank Fuchs-Kittowski, and Jochen Wittmann. Berichte aus der Umweltinformatik. Shaker, Herzogenrath, pp. 41–48 [cit. on pp. 9, 318, 339, 341].

Jelschen, Jan, Johannes Meier, Marie-Christin Ostendorp, and Andreas Winter [2013]. "A Description Model for Software Evolution Services". In: *1er Congreso Nacional*

*de Ingeniería Informática / Sistemas de Información*. RIISIC, Cordoba. URL: http://www.conaiisi.unsl.edu.ar/ingles/papers.php [cit. on pp. 9, 26, 177, 339].

Jelschen, Jan, Johannes Meier, and Andreas Winter [2015]. "SENSEI Applied: An Auto-Generated Toolchain for Q-MIG". In: *Softwaretechnik-Trends* 35.2, pp. 39–40 [cit. on pp. 9, 339].

Jelschen, Jan and Andreas Winter [2011]. "Towards a Catalogue of Software Evolution Services". In: *Softwaretechnik-Trends* 31.2, pp. 36–37 [cit. on pp. 9, 339].

Jelschen, Jan and Andreas Winter [2012]. "A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems". In: *Softwaretechnik-Trends* 32.2, pp. 67–68 [cit. on pp. 9, 339].

Jelschen, Jan and Andreas Winter [2014]. "Modeling Service Capabilities for Software Evolution Tool Integration". In: *Softwaretechnik-Trends* 34.2, pp. 91–92 [cit. on pp. 9, 154, 339].

*JGroups* [2020]. Red Hat. URL: http://www.jgroups.org/ [visited on 01/15/2020] [cit. on p. 299].

Jin, Dean and James R. Cordy [2003]. "A Service Sharing Approach to Integrating Program Comprehension Tools". In: *Proceedings of the Workshop on Tool Integration in System Development*. Ed. by Heiko Dörr and Andy Schürr. Darmstadt University of Technology, pp. 73–78. URL: https://web.archive.org/web/20070629182619/https://www.es.tu-darmstadt.de/english/events/tis/ [cit. on p. 75].

Jin, Dean and James R. Cordy [2005a]. "Factbase Filtering Issues in an Ontology-Based Reverse Engineering Tool Integration System". In: *Electronic Notes in Theoretical Computer Science* 137.3, pp. 65–75 [cit. on p. 75].

Jin, Dean and James R. Cordy [2005b]. "Ontology-based software analysis and reengineering tool integration: the OASIS service-sharing methodology". In: *Proceedings of the 21st International Conference on Software Maintenance*. IEEE, Los Alamitos, pp. 613–616 [cit. on pp. 4, 5, 8, 65, 75].

Jin, Dean, James R. Cordy, and Thomas R. Dean [2003]. "Transparent reverse engineering tool integration using a conceptual transaction adapter". In: *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. March. IEEE, Los Alamitos, pp. 399–408 [cit. on p. 75].

Jong, Hayco de and Paul Klint [2003]. "ToolBus: The Next Generation". In: *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Lecture Notes in Computer Science 2852. Springer, Berlin, Heidelberg, pp. 220–241 [cit. on p. 75].

Jordan, Diane, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro Guízar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter König, Mike Marin,

Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu [2007]. *Web Services Business Process Execution Language Version 2.0*. Standard. OASIS, Burlington [cit. on p. 124].

Josuttis, Nicolai M. [2007]. *SOA in Practice: The Art of Distributed System Design*. O'Reilly, Sebastopol [cit. on pp. 54, 109, 112–119, 124–126, 201].

Jouault, Frédéric, F Allilaire, Jean Bézivin, I Kurtev, and P Valduriez [2006]. "ATL: a QVT-like transformation language". In: *Proceedings of the 21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 719–720 [cit. on p. 143].

Jouault, Frédéric, Jean Bézivin, and Mikaël Barbero [2009]. "Towards an advanced model-driven engineering toolbox". In: *Innovations in Systems and Software Engineering* 5.1, pp. 5–12 [cit. on p. 140].

Juergens, Elmar, Florian Deißenböck, and Benjamin Hummel [2009]. "CloneDetective - A workbench for clone detection research". In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 603–606 [cit. on p. 18].

Kamiya, Toshihiro, Shinji Kusumoto, and Katsuro Inoue [2002]. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7, pp. 654–670 [cit. on p. 18].

Kamp, Manfred [1998]. "Managing a multi-file, multi-language software repository forprogram comprehension tools: a generic approach". In: *Proceedings of the 6th International Workshop on Program Comprehension*. IEEE, Los Alamitos [cit. on p. 236].

Kang, Kyo C., Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson [1990]. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Pittsburgh [cit. on p. 142].

Kang, Kyo C., Jaejoon Lee, and Patrick Donohoe [2002]. "Feature-oriented product line engineering". In: *IEEE Software* 19.4, pp. 58–65 [cit. on p. 142].

Karsai, Gabor, Andras Lang, and Sandeep Neema [2005]. "Design patterns for open tool integration". In: *Software & System Modeling* 4.2, pp. 157–170 [cit. on pp. x, 48, 53–55, 76, 167].

Karsai, Gabor, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty [2003]. "Model-integrated development of embedded software". In: *Proceedings of the IEEE* 91.1, pp. 145–164 [cit. on p. 137].

Kats, Lennart C. L. and Eelco Visser [2010]. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 444–463 [cit. on p. 143].

Kavantzas, Nickolas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto [2005]. *Web Services Choreography Description Language Version 1.0*. Standard. W3C, Cambridge [cit. on p. 125].

Kay, Alan C. [1993]. "The early history of Smalltalk". In: *ACM SIGPLAN Notices* 28.3, pp. 69–95 [cit. on pp. 3, 99].

Kazman, Rick and S. Jeromy Carrière [1999]. "Playing detective: Reconstructing software architecture from available evidence". In: *Automated Software Engineering* 6.2, pp. 107–138 [cit. on p. 70].

Keenan, Ed, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn [2012]. "TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions". In: *Proceedings of the 34th International Conference on Software Engineering*. IEEE, Los Alamitos, pp. 1375–1378 [cit. on p. 88].

Kelly, Steven, Kalle Lyytinen, and Matti Rossi [1996]. "Metaedit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment". In: *Proceedings of the 8th International Conference on Advanced Information Systems Engineering*. Ed. by Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou. Lecture Notes in Computer Science 1080. Springer, Berlin, Heidelberg, pp. 1–21 [cit. on pp. 143, 213].

Kelly, Steven and Juha-Pekka Tolvanen [2008]. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, Chichester [cit. on pp. 42, 134, 138, 140].

Kemerer, Chris F. [1992]. "How the Learning Curve Affects CASE Tool Adoption". In: *IEEE Software* 9.3, pp. 23–28 [cit. on p. 42].

Kienle, Holger M. [2006]. "Building Reverse Engineering Tools with Software Components". PhD thesis. University of Victoria [cit. on pp. 89, 91–93].

Kienle, Holger M. [2007]. "Building Reverse Engineering Tools with Software Components: Ten Lessons Learned". In: *Proceedings of the 14th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 289–292 [cit. on p. 91].

Kienle, Holger M. and Hausi A. Müller [2008]. "The Rigi reverse engineering environment". In: *Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques*. University of Bern. URL: `http://scg.unibe.ch/download/wasdett/wasdett2008-paper06.pdf` [cit. on p. 69].

Kindel, Charlie [1997]. "COM: What Makes it Work — black-box encapsulation through multiple, immutable interfaces". In: *Proceedings of the 1st International Enterprise Distributed Object Computing Workshop*. IEEE, Los Alamitos, pp. 68–77 [cit. on p. 99].

Kleppe, Anneke [2009]. "The Field of Software Language Engineering". In: *Proceedings of the First International Conference on Software Language Engineering*. Ed. by Dragan Gašević, Ralf Lämmel, and Eric Van Wyk. Lecture Notes in Computer Science 5452. Springer, Berlin, Heidelberg, pp. 1–7 [cit. on p. 134].

Kleppe, Anneke, Jos Warmer, and Wim Bast [2003]. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston [cit. on pp. 43, 129, 133, 134, 137–139].

Klint, Paul, Ralf Lämmel, and Chris Verhoef [2005]. "Toward an engineering discipline for grammarware". In: *ACM Transactions on Software Engineering and Methodology* 14.3, pp. 331–380 [cit. on p. 142].

Klint, Paul, Tijs van der Storm, and Jurgen Vinju [2011]. "{EASY} Meta-programming with Rascal". In: *Generative and Transformational Techniques in Software Engineering III - International Summer School*. Ed. by João M. Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva. Lecture Notes in Computer Science 6491. Springer, Berlin, Heidelberg, pp. 222–289 [cit. on p. 143].

Knodel, Jens and Matthias Naab [2014]. "Mitigating the Risk of Software Change in Practice". In: *Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Ed. by Serge Demeyer, David Binkley, and Filippo Ricca. IEEE, Antwerp, pp. 2–17 [cit. on pp. 4, 19].

*Knowledge Discovery Metamodel* [2016]. Object Management Group. URL: `http://www.omg.org/spec/KDM/` [visited on 01/15/2020] [cit. on p. 67].

Knuth, Donald E. [2011]. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, Boston [cit. on p. 381].

Kokko, Timo, Jari Antikainen, and Tarja Systä [2009]. "Adopting SOA - Experiences from nine finnish organizations". In: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 129–138 [cit. on pp. 122, 164].

Kolovos, Dimitrios S., Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck [2010]. "Taming EMF and GMF Using Model Transformation". In: *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems, Part I*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Lecture Notes in Computer Science 6394. Springer, Berlin, Heidelberg, pp. 211–225 [cit. on pp. 143, 213].

Kolovos, Dimitrios S., Louis M. Rose, Richard F. Paige, and Fiona A. C. Polack [2009]. "Raising the level of abstraction in the development of GMF-based graphical model editors". In: *Proceedings of the 1st Workshop on Modeling in Software Engineering*. IEEE, Los Alamitos, pp. 13–19 [cit. on p. 213].

Kopp, Oliver and Frank Leymann [2008]. "Choreography Design Using WS-BPEL". In: *Bulletin of the Technical Committee on Data Engineering* 31.3, pp. 31–34 [cit. on p. 125].

Koschke, Rainer [2000]. "Atomic Architectural Component Recovery for Program Understanding and Evolution: Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method". Dissertation. University of Stuttgart [cit. on p. 69].

Kouhen, Amine El, Cedric Dumoulin, Sébastien Gerard, and Pierre Boulet [2012]. *Evaluation of Modeling Tools Adaptation*. Tech. rep. hal-00706701. Hyper Articles en Ligne, Centre pour la Communication Scientifique Directe, Lyon [cit. on p. 213].

Kraft, Nicholas A. [2007]. "An Infrastructure to Support Interoperability in Reverse Engineering". PhD thesis. Clemson University [cit. on pp. 68, 71, 90].

Krafzig, Dirk, Karl Banke, and Dirk Slama [2005]. *Enterprise SOA: Service-oriented Architecture Best Practices*. Prentice Hall, Upper Saddle River [cit. on p. 116].

Kruchten, Philippe [2010]. "Software Architecture and Agile Software Development—A Clash of Two Cultures?" In: *Proceedings of the 32nd International Conference on Software Engineering, Volume 2*. ACM, New York, p. 497 [cit. on p. 4].

Kühne, Stefan and Christian Wetzel [2006]. "Metamodellierung am Beispiel der E-Government-Domäne Meldewesen und Eclipse GMF". In: *Integration betrieblicher Informationssysteme: Problemanalysen und Lösungsansätze des Model-Driven Integration Engineering*. Ed. by Klaus-Peter Fähnrich, Stefan Kühne, Andreas Speck, and Julia Wagner. Leipziger Beiträge zur Informatik IV. Leipziger Informatik-Verbund, Leipzig, pp. 59–72 [cit. on p. 212].

Kühne, Thomas [2006]. "Matters of (Meta-) Modeling". In: *Software & Systems Modeling* 5.4, pp. 369–385 [cit. on p. 133].

Kullbach, Bernt and Andreas Winter [1999]. "Querying as an enabling technology in software reengineering". In: *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering*. IEEE, Los Alamitos, pp. 42–50 [cit. on p. 71].

Küpker, Christoph Alexander [2015]. "Applying the SENSEI Service Orchestration Approach to WSO2". Master's Thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 79, 85, 105, 230, 233, 256–258, 277, 312–314, 316, 376–378].

Kurtev, Ivan, Jean Bézivin, and Mehmet Aksit [2002]. *Technological Spaces: An Initial Appraisal*. Paper presented at the 4th International Symposium on Distributed Objects and Applications. University of Twente. URL: `https://research.utwente.nl/en/publications/technological-spaces-an-initial-appraisal` [cit. on p. 142].

Kurtev, Ivan, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez [2006]. "Model-based DSL frameworks". In: *Proceedings of the 21st Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, pp. 602–616 [cit. on p. 140].

Kuryazov, Dilshodbek [2014]. "Delta Operations Language for Model Difference Representation". In: *Informatik 2014: Big Data, Komplexität meistern. Beitragsband der 44. Jahrestagung der Gesellschaft für Informatik*. Ed. by Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull. Lecture Notes in Informatics 232. Gesellschaft für Informatik, Bonn, pp. 2221–2232 [cit. on p. 193].

Kuryazov, Dilshodbek and Andreas Winter [2014]. "Representing Model Differences by Delta Operations". In: *Proceedings of the 18th International Enterprise Distributed*

*Object Computing Conference Workshops and Demonstrations*. IEEE, Los Alamitos, pp. 211–220 [cit. on p. 193].

Kuryazov, Dilshodbek, Andreas Winter, and Alexander Sandau [2019]. "Sustainable Software Architecture for NEMo Mobility Platform". In: *Smart Cities/Smart Regions - Technische, wirtschaftliche und gesellschaftliche Innovationen. Konferenzband zu den 10. BUIS-Tagen*. Ed. by Jorge Marx Gómez, Andreas Solsbach, Thomas Klenke, and Volker Wohlgemuth. Springer Fachmedien, Wiesbaden, pp. 229–239 [cit. on pp. 327, 339].

Laarman, Alfons and Ivan Kurtev [2010]. "Ontological metamodeling with explicit instantiation". In: *Proceedings of the 2nd International Conference on Software Language Engineering*. Ed. by Mark van den Brand, Dragan Gašević, and Jeff Gray. Lecture Notes in Computer Science 5969. Springer, Berlin, Heidelberg, pp. 174–183 [cit. on p. 136].

Lamprecht, Anna-Lena [2013]. *User-Level Workflow Design: A Bioinformatics Perspective*. Vol. 8311. LNCS Programming and Software Engineering. Springer, Berlin, Heidelberg [cit. on pp. 88, 89].

Lamprecht, Anna-Lena, Tiziana Margaria, and Bernhard Steffen [2014]. "Modeling and Execution of Scientific Workflows with the jABC Framework". In: *Process Design for Natural Scientists: An Agile Model-Driven Approach*. Ed. by Anna-Lena Lamprecht and Tiziana Margaria. Communications in Computer and Information Science 500. Springer, Berlin, Heidelberg, pp. 14–29 [cit. on p. 88].

Lamprecht, Anna-Lena, Bernhard Steffen, and Tiziana Margaria [2016]. "Scientific workflows with the jABC framework: A review after a decade in the field". In: *International Journal on Software Tools for Technology Transfer* 18.6, pp. 629–651 [cit. on pp. 88, 89].

Land, Rikard and Ivica Crnkovic [2004]. "Existing Approaches to Software Integration - and a Challenge for the Future". In: *Proceedings of the 4th Conference on Software Engineering Research and Practice in Sweden*. Mälardalen University, Västerås. URL: `http://www.es.mdh.se/publications/642-Existing_Approaches_to_Software_Integration___and_a_Challenge_for_the_Future` [cit. on p. 4].

Laskey, Ken, Peter Brown, Jeff A. Estefan, Francis G. McCabe, and Danny Thornton [2012]. *Reference Architecture for Service Oriented Architecture Version 1.0*. Standard. OASIS, Burlington [cit. on p. 118].

Laws, Simon, Mark Combellack, Raymond Feng, Haleh Mahb, and Simon Nash [2011]. *Tuscany SCA in Action*. Manning, Shelter Island [cit. on pp. 48, 186, 240, 286, 294, 299].

Ledbetter, Lamar and Brad Cox [1985]. "Software-ICs: A plan for building reusable software components". In: *BYTE* 10.6, pp. 307–316 [cit. on p. 99].

Ledeczi, Akos, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi [2001]. "The

Generic Modeling Environment". In: *Proceedings of the 2nd Workshop on Intelligent Signal Processing*. IEEE, Los Alamitos. URL: https://www.isis.vanderbilt.edu/sites/default/files/GME2000Overview.pdf [cit. on p. 143].

Lee, Jay, Behrad Bagheri, and Hung-An Kao [2015]. "A Cyber-Physical Systems architecture for Industry 4.0-based manufacturing systems". In: *Manufacturing Letters* 3, pp. 18–23 [cit. on p. 6].

Lehman, Meir M. [1980]. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9, pp. 1060–1076 [cit. on pp. 3, 333].

Lehman, Meir M. [1996]. "Laws of software evolution revisited". In: *Proceedings of the 5th European Workshop on Software Process Technology*. Lecture Notes in Computer Science 1149. Springer, Berlin, Heidelberg, pp. 108–124 [cit. on pp. 3, 308].

Leitner, Andrea, Beate Herbst, and Roland Mathijssen [2016]. "Lessons Learned from Tool Integration with OSLC". In: *Proceedings of the 22nd International Conference Information and Software Technologies*. Communications in Computer and Information Science 639. Springer, Cham, pp. 242–254 [cit. on p. 68].

Lending, Diane and Norman L. Chervany [1998]. "The use of CASE tools". In: *Proceedings of the ACM SIGCPR Conference on Computer Personnel Research*. Ed. by Fred Niederman and Ritu Agarwal. ACM, New York, pp. 49–58 [cit. on p. 42].

Lethbridge, Timothy C., Sander Tichelaar, and Erhard Ploedereder [2004]. "The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering". In: *Electronic Notes in Theoretical Computer Science* 94, pp. 7–18 [cit. on pp. 65, 66, 68, 70].

Levina, Olga and Vladimir Stantchev [2009]. "Realizing Event-Driven SOA". In: *Proceedings of the 4th International Conference on Internet and Web Applications and Services*. IEEE, Los Alamitos, pp. 37–42 [cit. on p. 125].

Lewis, James and Martin Fowler [2014]. *Microservices*. URL: http://martinfowler.com/articles/microservices.html [visited on 01/15/2020] [cit. on pp. 109, 114, 122].

Lientz, Bennet P. and E. Burton Swanson [1980]. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, Boston [cit. on p. 3].

Lin, Yuan, Richard C. Holt, and Andrew J. Malton [2003]. "Completeness of a fact extractor". In: *Proceedings of the 10th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 196–205 [cit. on p. 268].

Long, Fred and Edwin J. Morris [1993]. *An Overview of PCTE: A Basis for a Portable Common Tool Environment*. Tech. rep. CMU/SEI-93-TR-001. Software Engineering Institute, Pittsburgh [cit. on p. 40].

Lungu, Mircea, Michele Lanza, and Oscar Nierstrasz [2014]. "Evolutionary and collaborative software architecture recovery with Softwarenaut". In: *Science of Computer Programming* 79, pp. 204–223 [cit. on p. 73].

MacKenzie, C. Matthew, Ken Laskey, Francis McCabe, Peter F. Brown, and Rebekah Metz [2006]. *Reference Model for Service Oriented Architecture 1.0*. Standard. OASIS, Burlington [cit. on pp. 112, 118].

*Mailing List Archives of the SCA-Bindings Technical Committee* [2013]. OASIS. URL: https://lists.oasis-open.org/archives/sca-bindings/201306/maillist.html [visited on 01/15/2020] [cit. on pp. 106, 238].

Manes, Anne Thomas [2009]. *SOA Is Dead; Long Live Services*. URL: https://web.archive.org/web/20160506053536/http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services/comments/page/1/ [visited on 01/15/2020] [cit. on pp. 109, 111, 118, 122].

Manes, Anne Thomas. [2003]. *Web Services: A Manager's Guide*. Addison-Wesley, Boston [cit. on p. 3].

Manolescu, Dragos [2000]. "Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development". PhD thesis. University of Illinois at Urbana-Champaign [cit. on p. 116].

Marchewka, Katrin [2006]. "GReQL 2". Diploma Thesis. University of Koblenz-Landau [cit. on p. 236].

Margaria, Tiziana, Christian Kubczak, Mark Njoku, and Bernhard Steffen [2006]. "Model-based design of distributed collaborative bioinformatics processes in the jABC". In: *Proceedings of the 11th International Conference on Engineering of Complex Computer Systems*. IEEE, Los Alamitos, pp. 169–176 [cit. on p. 88].

Margaria, Tiziana, Christian Kubczak, and Bernhard Steffen [2008]. "Bio-jETI: a service integration, design, and provisioning platform for orchestrated bioinformatics processes". In: *A Semantic Web for Bioinformatics: Goals, Tools, Systems, Applications. Proceedings of the 7th International Workshop on Network Tools and Applications in Biology*. Ed. by Paolo Romano, Michael Schroeder, Nicola Cannata, and Roberto Marangoni. BMC Bioinformatics 9(Suppl 4).S12. BioMed Central, London [cit. on p. 88].

Margaria, Tiziana, Ralf Nagel, and Bernhard Steffen [2005]. "jETI : A Tool for Remote Tool Integration". In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Lecture Notes in Computer Science 3440. Springer, Berlin, Heidelberg, pp. 557–562 [cit. on p. 88].

Marino, Jim and Michael Rowley [2009]. *Understanding SCA (Service Component Architecture)*. Addison Wesley, Boston [cit. on pp. 113, 238, 240, 299].

Martin, Robert C. [2003]. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River [cit. on pp. 122, 292].

Martin, Roger J. [1993]. *Reference Model for Frameworks of Software Engineering Environments*. Technical Report / Special Publication ECMA TR/55, NIST SP 500-211, European Computer Manufacturers Association / National Institute of Standards and Technology [cit. on pp. 40, 42, 89, 90].

Marx Gómez, Jorge [2019]. "Serviceorientierte Architektur". In: *Enzyklopädie der Wirtschaftsinformatik – Online-Lexikon*. Ed. by Norbert Gronau, Jörg Becker, Natalia Kliewer, Jan Marco Leimeister, and Sven Overhage. GITO, Berlin [cit. on pp. 119–121].

McIlroy, Malcolm Douglas [1968]. "Mass-Produced Software Components". In: *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. Ed. by Peter Naur and Brian Randell, pp. 138–155 [cit. on pp. 3, 99].

Meier, Almuth [2014a]. "Ein Composition-Finder für Service-Orchestrierungen". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 202, 204, 207, 208, 243, 256, 382, 385].

Meier, Johannes [2012]. "Eine Fallstudie zur Interoperabilität von Software-Evolutions-Werkzeugen in SCA". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Meier, Johannes [2014b]. "Editoren für Service-Orchestrierungen". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 157, 182, 211–213, 215, 216].

Meier, Johannes, Dilshodbek Kuryazov, Jan Jelschen, and Andreas Winter [2015]. "A Quality Control Center for Software Migration". In: *Softwaretechnik-Trends* 35.2, pp. 19–20 [cit. on p. 12].

Meier, Johannes and Andreas Winter [2016]. "Towards Metamodel Integration Using Reference Metamodels". In: *Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. Ed. by Colin Atkinson, Erik Burger, Thomas Goldschmidt, and Ralf Reussner. Karlsruhe Reports in Informatics 2016.7. Karlsruher Institut für Technologie, pp. 19–22 [cit. on p. 54].

Mell, Peter and Timothy Grance [2011]. *The NIST definition of cloud computing*. Special Publication NIST SP 800-145. National Institute of Standards and Technology, Gaithersburg, MD [cit. on pp. 107, 199].

Melzer, Ingo and Sebastian Eberhard [2010]. *Service-orientierte Architekturen mit Web Services*. 4. Edition. Spektrum, Heidelberg [cit. on pp. 112, 113, 119, 120].

Menascé, Daniel A. [2005]. "MOM vs. RPC: Communication Models for Distributed Applications". In: *IEEE Internet Computing* 9.2, pp. 90–93 [cit. on p. 119].

Mens, Tom and Pieter Van Gorp [2006]. "A Taxonomy of Model Transformation". In: *Proceedings of the International Workshop on Graph and Model Transformation*. Ed. by Gabor Karsai and Gabriele Taentzer. Electronic Notes in Theoretical Computer Science 152. Elsevier, Amsterdam, pp. 125–142 [cit. on pp. 139, 140].

Mens, Tom, Michel Wermelinger, Serge Demeyer, Robert Hirschfeld, Stéphane Ducasse, and M Jazayeri [2005]. "Challenges in software evolution". In: *Proceedings of the 8th International Workshop on Principles of Software Evolution*. Ed. by Motoshi Saeki, Gerardo Canfora, and Shuichiro Yamamoto. IEEE, Los Alamitos, pp. 13–22 [cit. on pp. 5, 8].

*Meta Object Facility* [2013]. Object Management Group. URL: `http://www.omg.org/spec/MOF/2.4.1/` [visited on 01/15/2020] [cit. on pp. 67, 135].

MetaCase [2020]. *MetaCase - Domain-Specific Modeling with MetaEdit+*. URL: http://www.metacase.com/ [visited on 01/15/2020] [cit. on p. 213].

Michelson, Branda M. [2006]. *Event-Driven Architecture Overview: Event-Driven SOA Is Just Part of the EDA Story*. White paper. Patricia Seybold Group, Boston [cit. on p. 125].

Michlmayr, Anton, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar [2007]. "Towards recovering the broken SOA triangle". In: *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering*. ACM, New York, pp. 22–28 [cit. on pp. 120, 121].

Microsoft [2016]. *Overview of Domain-Specific Language Tools*. URL: https://msdn.microsoft.com/en-us/library/bb126327.aspx [visited on 01/15/2020] [cit. on pp. 143, 213].

*Model Driven Architecture* [2020]. Object Management Group. URL: http://www.omg.org/mda/ [visited on 01/15/2020] [cit. on p. 136].

*ModelBus* [2017]. Fraunhofer Institute for Open Communication Systems. URL: http://www.modelbus.org/ [visited on 01/15/2020] [cit. on p. 77].

*MOFM2T 1.0* [2008]. Object Management Group. URL: http://www.omg.org/spec/MOFM2T/1.0/ [visited on 01/15/2020] [cit. on p. 212].

*Moose* [2020]. Moose Community. URL: http://www.moosetechnology.org/ [visited on 01/15/2020] [cit. on p. 73].

Morris, Edwin, Linda Levine, Patrick R. Place, Daniel Plakosh, and B. Craig Meyers [2004]. *Systems of Systems Interoperability*. Tech. rep. CMU/SEI-2004-TR-004. Software Engineering Institute, Pittsburgh [cit. on p. 47].

Müller, Hausi A., Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong [2000]. "Reverse engineering: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM, New York, pp. 47–60 [cit. on pp. 4, 5, 8].

Nassi, Isaac and Ben Shneiderman [1973]. "Flowchart techniques for structured programming". In: *SIGPLAN Notices* 8.8, pp. 12–26 [cit. on p. 85].

Naur, Peter and Brian Randell [1968]. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*. Conference Report. NATO Scientific Affairs Division, Brussels [cit. on pp. 3, 39].

Neumayr, Bernd, Katharina Grün, and Michael Schrefl [2009]. "Multi-level domain modeling with m-objects and m-relationships". In: *Proceedings of the 6th Asia-Pacific Conference on Conceptual Modeling*. Ed. by Markus Kirchberg and Sebastian Link. Conferences in Research and Practice in Information Technology 96, Australian Computer Science Communications 31.6. Australian Computer Society, Darlinghurst, pp. 107–116 [cit. on p. 136].

Newman, Sam [2015]. *Building Microservices*. O'Reilly, Sebastopol [cit. on pp. 6, 114, 118, 122, 125, 126].

Nierstrasz, Oscar [2012]. "Agile software assessment with Moose". In: *ACM SIGSOFT Software Engineering Notes* 37.3, pp. 1–5 [cit. on pp. 65, 67].

Nurkiewicz, Tomasz [2015]. *RESTful Considered Harmful*. URL: `https://dzone.com/articles/restful-considered-harmful` [visited on 01/15/2020] [cit. on p. 122].

*Object Constraint Language* [2014]. Object Management Group. URL: `http://www.omg.org/spec/OCL/2.4` [visited on 01/15/2020] [cit. on p. 175].

Ocampo, Camilo, Begoña Albizuri, and Pere Botella [1998]. "Is CASE Technology Still Alive?" In: *Actas de las III Jornadas de Ingeniería del Software*. Ed. by José Ambrosio Toval Álvarez and Joaquín Nicolás Ros. Diego Marín, Murcia, pp. 127–139 [cit. on p. 42].

*Open Services for Lifecycle Collaboration* [2020]. OSLC Community. URL: `http://open-services.net/` [visited on 01/15/2020] [cit. on pp. 65, 85].

*OSGi* [2020]. OSGi Alliance. URL: `https://www.osgi.org` [cit. on pp. 100, 102, 105, 237].

Pandey, Gaurav [2014]. *Short Report on Clone Detection Tools*. Tech. rep. Carl von Ossietzky University, Oldenburg [cit. on pp. 18, 32].

Parker, Burt [1992]. "Introducing EIA-CDIF: the CASE Data Interchange Format Standard". In: *Proceedings of the 2nd Symposium on Assessment of Quality Software Development Tools*. IEEE, Los Alamitos, pp. 74–82 [cit. on pp. 40, 65].

Parnas, David Lorge [1972]. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12, pp. 1053–1058 [cit. on p. 36].

Pastor, Óscar and Juan Carlos Molina [2007]. *Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling*. Springer, Berlin, Heidelberg [cit. on pp. 136, 138].

Pech, Vaclav, Alex Shatalin, and Markus Völter [2013]. "JetBrains MPS as a tool for extending Java". In: *Proceedings of the 10th International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools*. ACM, New York, pp. 165–168 [cit. on pp. 143, 213].

Peirce, Charles Santiago Sanders [1906]. "Prolegomena to an Apology for Pragmaticism". In: *Monist* 16.4, pp. 492–546 [cit. on p. 133].

Pérez-Castillo, Ricardo, Ignacio García Rodríguez De Guzmán, and Mario Piattini [2011]. "Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems". In: *Computer Standards & Interfaces* 33.6, pp. 519–532 [cit. on p. 67].

Pike, Rob and Brian W. Kernighan [1984]. "The UNIX System: Program Design in the UNIX Environment". In: *AT&T Bell Laboratories Technical Journal* 63.8, pp. 1595–1605 [cit. on pp. 5, 70].

*Q-MIG* [2015]. Software Engineering Group of Carl von Ossietzky University and pro et con Innovative Informatikanwendungen GmbH. URL: `http://se.uni-oldenburg.de/Q-MIG` [visited on 01/15/2020] [cit. on p. 25].

Rajlich, Václav [2014]. "Software evolution and maintenance". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM, New York, pp. 133–144 [cit. on p. 6].

Rajlich, Vaclav and Keith Bennett [2000]. "A Staged Model for the Software Life Cycle". In: *Computer* 33.7, pp. 66–71 [cit. on p. 309].

Ran, Shuping [2003]. "A model for web services discovery with QoS". In: *ACM SIGecom Exchanges* 4.1, pp. 1–10 [cit. on p. 120].

Raza, Aoun, Gunther Vogel, and Erhard Plödereder [2006]. "Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering". In: *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies*. Ed. by Luís Miguel Pinho and Michael González Harbour. Lecture Notes in Computer Science 4006. Springer, Berlin, Heidelberg, pp. 71–82 [cit. on p. 69].

Red Hat [2015]. *JBoss Fuse Service Works 6.0 Development Guide Volume 1: SwitchYard*. Development Guide. Red Hat, Raleigh. URL: `https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse_service_works/6.0/pdf/development_guide_volume_1_switchyard/Red_Hat_JBoss_Fuse_Service_Works-6.0-Development_Guide_Volume_1_SwitchYard-en-US.pdf` [cit. on p. 200].

Resende, Luciano and Raymond Feng [2007]. "Handling Heterogeneous Data Sources in a SOA Environment with Service Data Objects (SDO)". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Volume 1*. Ed. by Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou. ACM, New York, pp. 895–897 [cit. on p. 201].

Richardson, Leonard and Sam Ruby [2008]. *RESTful Web Services*. O'Reilly, Sebastopol [cit. on pp. 109, 119, 120, 122, 125].

Ringe, Mathias [2013]. "Vergleich komponentenbasierter Frameworks zur Werkzeugintegration". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 105, 106, 237–239].

Roy, Chanchal K., James R. Cordy, and Rainer Koschke [2009]. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Science of Computer Programming* 74.7, pp. 470–495 [cit. on p. 31].

Royce, Winston W. [1970]. "Managing the development of large software systems". In: *WESCON technical papers. Papers presented at the Western Electronic Show and Convention*. Los Angeles. IEEE, Los Alamitos, pp. 328–338 [cit. on p. 39].

Russell, Nick, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed [2006]. "On the suitability of UML 2.0 activity diagrams for business process modelling." In: *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modeling*. Ed. by Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki. Conferences in Research and Practice in Information Technology 53, Australian Computer Science Communications 28.6. Australian Computer Society, Darlinghurst, pp. 95–104 [cit. on p. 124].

Sanner, Michael F. [1999]. "Python: a programming language for software integration and development". In: *Journal of molecular graphics & modelling* 17.1, pp. 57–61 [cit. on p. 4].

Schlömer, Timo [2017]. "Modellgetriebene GUI-Erstellung für serviceorientierte Anwendungen". Master's thesis. Carl von Ossietzky University, Oldenburg [cit. on pp. 324–326, 342].

Schmidt, Alexander, Boris Otto, and Hubert Österle [2010]. "Integrating information systems: Case studies on current challenges". In: *Electronic Markets* 20.2, pp. 161–174 [cit. on p. 116].

Schmidt, Douglas C. [1999]. "Why Software Reuse has Failed and How to Make it Work for You". In: *C++ Report* 11.1 [cit. on p. 20].

Schmidt, Douglas C. [2006]. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2, pp. 25–31 [cit. on pp. 42, 309].

Schulte, W. Roy and Yefim Natis [1996]. *"Service Oriented" Architectures, Part 1*. Research Note SPA-401-068, G0029201. Gartner, Stamford [cit. on p. 109].

Schwaber, Ken and Mike Beedle [2002]. *Agile software development with Scrum*. Prentice Hall, Upper Saddle River [cit. on p. 4].

Seacord, Robert C., Daniel Plakosh, and Grace A. Lewis [2003]. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, Boston [cit. on p. 3].

Seidewitz, Ed [2003]. "What models mean". In: *IEEE Software* 20.5, pp. 26–32 [cit. on p. 133].

*SEON - Software Evolution ONtologies* [2016]. Software Evolution and Architecture Lab. URL: `http://se-on.org` [visited on 01/15/2020] [cit. on p. 67].

*Service Component Architecture (SCA)* [2015]. OASIS Open CSA. URL: `http://oasis-opencsa.org/sca` [visited on 01/15/2020] [cit. on pp. 79, 105, 231].

Sharon, David and Rodney Bell [1995]. "Tools that bind: Creating Integrated Environments". In: *IEEE Software* 12.2, pp. 76–85 [cit. on p. 41].

Siedersleben, Johannes [2004]. *Moderne Software-Architektur: Umsichtig planen, robust bauen mit Quasar*. Dpunkt, Heidelberg [cit. on pp. 117, 118].

Siegel, Jon M. [2014]. *MDA Guide Rev. 2.0*. White paper ORMSC/2014-06-01. Object Management Group, Needham [cit. on pp. 138, 140].

Sim, Susan Elliott [2000]. "Next Generation Data Interchange: Tool-to-Tool Application Program Interface". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 278–280 [cit. on pp. 4, 5, 7, 52, 93].

*Sirius* [2020]. Eclipse Foundation. URL: `https://eclipse.org/sirius/index.html` [visited on 01/15/2020] [cit. on pp. 143, 211, 213].

*Sirius Specifier Manual* [2020]. Eclipse Foundation. URL: `https://www.eclipse.org/sirius/doc/specifier/Sirius%20Specifier%20Manual.html` [visited on 01/15/2020] [cit. on pp. 211, 220].

Skyttner, Lars [2005]. *General Systems Theory: Problems, Perspectives, Practice*. 2nd ed. World Scientific, Singapore [cit. on p. 132].

*Smart Modeling* [2020]. Carl von Ossietzky University of Oldenburg and Urgench branch of Tashkent University of Information Technologies named after Muhammed al-Khorazmiy. URL: `https://smart-modeling.ubtuit.uz/` [visited on 01/15/2020] [cit. on p. 343].

Sneed, Harry M., Ellen Wolf, and Heidi Heilmann [2010]. *Softwaremigration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. Dpunkt, Heidelberg [cit. on pp. 5, 7].

Sommerville, Ian [2011]. *Software Engineering*. 9th ed. Addison-Wesley, Boston [cit. on pp. 103, 110, 112, 118, 120, 122].

Sprinkle, Jonathan, Marjan Mernik, Juha-Pekka Tolvanen, and Diomidis Spinellis [2009]. "What Kinds of Nails Need a Domain-Specific Hammer". In: *IEEE Software* 26.4, pp. 15–18 [cit. on p. 141].

Stachowiak, Herbert [1973]. *Allgemeine Modelltheorie*. Springer, Wien [cit. on p. 132].

Stahl, Thomas, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen [2006]. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester [cit. on pp. 42, 135–138].

Statistisches Bundesamt [2015]. *Statistisches Jahrbuch Deutschland 2015*. Statistisches Bundesamt, Wiesbaden [cit. on p. 308].

Steinberg, Dave, Frank Budinsky, Marcelo Paternostro, and Ed Merks [2008]. *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. The Eclipse Series. Addison-Wesley, Boston [cit. on pp. 73, 143, 211, 213].

Storey, Margaret-Anne, Casey Best, and Jeff Michand [2001]. "Shrimp views: An interactive environment for exploring java programs". In: *Proceedings of the 9th International Workshop on Program Comprehension*. IEEE, Los Alamitos [cit. on p. 71].

Strittmatter, Misha, Michael Junker, Kiana Rostami, Sebastian Lehrig, Amine Kechaou, Bo Liu, and Robert Heinrich [2016]. "Extensible Graphical Editors for Palladio". In: *Softwaretechnik Trends* 36.4, pp. 49–51 [cit. on p. 214].

*SWAG Tools* [2020]. SWAG. URL: `http://www.swag.uwaterloo.ca/tools.html` [visited on 01/15/2020] [cit. on p. 70].

*SwitchYard* [2020]. Red Hat. URL: `http://switchyard.jboss.org/` [visited on 2020] [cit. on pp. 106, 239].

Szyperski, Clemens [1997]. *Component software: beyond object-oriented programming*. ACM Press Books. Addison Wesley, Boston [cit. on pp. 3, 99, 100, 102, 103].

Terry, B. and D. Logee [1990]. "Terminology for Software Engineering Environment (SEE) and Computer-Aided Software Engineering (CASE)". In: *ACM SIGSOFT Software Engineering Notes* 15.2, pp. 83–94 [cit. on p. 45].

Thomas, Ian and Brian A. Nejmeh [1992]. "Definitions of tool integration for environments". In: *IEEE Software* 9.2, pp. 29–35 [cit. on pp. 45–48, 50, 52].

Thompson, Henry S., David Beech, Murray Maloney, and Noah Mendelsohn [2004]. *XML Schema Part 1: Structures Second Edition*. URL: https://www.w3.org/TR/xmlschema-1/#normative-schemaSchema [visited on 01/15/2020] [cit. on p. 142].

Tichelaar, Sander, Stéphane Ducasse, and Serge Demeyer [2000]. "FAMIX and XMI". In: *Proceedings of the 7th Working Conference on Reverse Engineering*. IEEE, Los Alamitos, pp. 296–299 [cit. on p. 67].

Tihonov, Sergej [2013]. "Servicebasierte Refactorings". Bachelor's thesis. Carl von Ossietzky University, Oldenburg [cit. on p. 237].

Tolk, Andreas and James Muguira [2003]. "The Levels of Conceptual Interoperability Model". In: *Proceedings of the Fall Simulation Interoperability Workshop*. Curran Associates, Red Hook, pp. 53–62 [cit. on p. 47].

Tsang, Edward. [1993]. *Foundations of constraint satisfaction*. Academic Press, London [cit. on p. 243].

Tunjic, Christian and Colin Atkinson [2015]. "Synchronization of Projective Views on a Single-Underlying-Model". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. Ed. by Uwe Aßmann, Colin Atkinson, Erik Burger, Thomas Goldschmidt, and Ralf Reussner. ACM, New York, pp. 55–58 [cit. on p. 54].

Vaughan-Nichols, Steven J. [2002]. "Web services: beyond the hype". In: *Computer* 35.2, pp. 18–21 [cit. on p. 109].

Visser, Eelco [2004]. "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9". In: *Domain-Specific Program Generation*. Ed. by Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky. Lecture Notes in Computer Science 3016. Springer, Berlin, Heidelberg, pp. 216–238 [cit. on p. 144].

Vogel, Oliver, Ingo Arnold, Arif Chughtai, and Timo Kehrer [2011]. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. Springer, Berlin, Heidelberg [cit. on pp. 112, 113].

Vujović, Vladimir, Mirjana Maksimović, and Branko Perišić [2014]. "Comparative analysis of DSM Graphical Editor frameworks: Graphiti vs . Sirius". In: *Proceedings of the 23rd International Electrotechnical and Computer Science Conference*. Ed. by Baldomir Zajc and Andrej Trost. University of Ljubljana, Ljubljana, pp. 7–10. URL: https://erk.fe.uni-lj.si/2014/index.html [cit. on p. 214].

W3C OWL Working Group [2012]. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. URL: https://www.w3.org/TR/owl2-overview/ [visited on 01/15/2020] [cit. on p. 65].

Wagner, Christian [2014]. *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Fachmedien, Wiesbaden [cit. on p. 136].

Wagner vom Berg, Benjamin [2015]. *Konzeption Eines Sustainability Customer Relationship Managements (SusCRM) Für Anbieter Nachhaltiger Mobilität*. Shaker, Herzogenrath [cit. on p. 309].

Wasserman, Anthony I. [1990]. "Tool Integration in Software Engineering Environments". In: *Software Engineering Environments. Proceedings of the International Workshop on Environments*. Lecture Notes in Computer Science 467. Ed. by Fred Long, pp. 137–149 [cit. on pp. x, 32, 39, 41, 44, 45, 48–51].

Wasserman, Anthony I. [1996]. "Toward a discipline of software engineering". In: *IEEE Software* 13.6, pp. 23–31 [cit. on p. 49].

Wegner, Peter [1996]. "Interoperability". In: *ACM Computing Surveys* 28.1, pp. 285–287 [cit. on pp. 46, 47].

Weinreich, Rainer and Johannes Sametinger [2001]. "Component models and component services: concepts and principles". In: *Component-based software engineering: putting the pieces together*. Ed. by George T. Heineman and William T. Councill. Addison-Wesley, Boston, pp. 33–48 [cit. on p. 104].

Wettel, Richard and Radu Marinescu [2005]. "Archeology of code duplication: Recovering duplication chains from small duplication fragments". In: *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE. Los Alamitos, pp. 63–70 [cit. on pp. 18, 284, 285].

Whittle, Jon, John Hutchinson, and Mark Rouncefield [2014]. "The State of Practice in Model-Driven Engineering". In: *IEEE Software* 31.3, pp. 79–85 [cit. on p. 141].

Wicks, Michael N. and Richard G. Dewar [2007]. "A new research agenda for tool integration". In: *Journal of Systems and Software* 80.9, pp. 1569–1585 [cit. on pp. 41, 46].

Wielemaker, Jan [2020]. *SWI-Prolog*. URL: `http://www.swi-prolog.org/` [visited on 01/15/2020] [cit. on p. 243].

Wienands, Christoph and Michael Golm [2009]. "Anatomy of a Visual Domain-Specific Language Project in an Industrial Context". In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Lecture Notes in Computer Science 5795. Springer, Berlin, Heidelberg, pp. 453–467 [cit. on p. 213].

Winter, Andreas [2000]. *Referenz-Metaschema für visuelle Modellierungssprachen*. Deutscher Universitätsverlag, Wiesbaden [cit. on p. 118].

Winter, Andreas and Jürgen Ebert [2005a]. "Metamodel-driven Service Interoperability". In: *Pre-Proceedings of 13th International Workshop on Software Technology and Engineering Practice*. Ed. by Ying Zou and Massimiliano Di Penta. Queen's University, Kingston, pp. 167–176. URL: `http://post.queensu.ca/%7B~%7Dzouy/files/preproc-step-2005.pdf%7B%5C#%7Dpage=178` [cit. on p. 75].

Winter, Andreas and Jürgen Ebert [2005b]. "Using metamodels in service interoperability". In: *Proceedings of 13th International Workshop on Software Technology and*

*Engineering Practice*. Ed. by Kostas Kontogiannis, Ying Zou, and Massimiliano Di Penta. IEEE, Los Alamitos, pp. 147–156 [cit. on pp. 75, 76].

Winter, Andreas, Bernt Kullbach, and Volker Riediger [2002]. "An Overview of the GXL Graph Exchange Language". In: *Software Visualization*. Ed. by Stephan Diehl. Lecture Notes in Computer Science 2269. Springer, Berlin, Heidelberg, pp. 324–336 [cit. on p. 193].

Wirsing, Martin and Matthias Hölzl, eds. [2011]. *Rigorous Software Engineering for Service-Oriented Systems: Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*. Vol. 6582. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg [cit. on p. 76].

Wirth, Niklaus [1986]. *Algorithms and Data Structures*. Prentice Hall, Upper Saddle River [cit. on p. 166].

Wirth, Niklaus [2008]. "A Brief History of Software Engineering". In: *IEEE Annals of the History of Computing* 30.3, pp. 32–39 [cit. on p. 39].

Wohed, Petia, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell [2006]. "On the Suitability of BPMN for Business Process Modelling". In: *Proceedings of the 4th International Conference on Business Process Management*. Ed. by Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth. Lecture Notes in Computer Science 4102. Springer, Berlin, Heidelberg, pp. 161–176 [cit. on p. 124].

Wolstencroft, Katherine, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P Balcazar Vargas, Shoaib Sufi, and Carole Goble [2013]. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." In: *Nucleic acids research* 41, W557–W561 [cit. on p. 87].

*WSO2* [2020]. URL: `http://wso2.com` [visited on 01/15/2020] [cit. on pp. 48, 105, 256, 310].

Würsch, Michael, Giacomo Ghezzi, Matthias Hert, Gerald Reif, and Harald C. Gall [2012]. "SEON: a pyramid of ontologies for software evolution and its applications". In: *Computing* 94.11, pp. 857–885 [cit. on pp. 65, 67, 201].

*XMI (XML Metadata Interchange)* [2015]. Object Management Group. URL: `http://www.omg.org/spec/XMI/` [visited on 01/15/2020] [cit. on pp. 40, 65].

*Xtext - Language Engineering for Everyone!* [2020]. Eclipse Foundation. URL: `http://www.eclipse.org/Xtext/` [visited on 01/15/2020] [cit. on p. 212].

Yandell, Henri [2016]. *Apache Tuscany retired*. URL: `http://mail-archives.apache.org/mod_mbox/www-announce/201608.mbox/browser` [visited on 01/15/2020] [cit. on p. 238].

Yang, Yun and Jun Han [1996]. "Classification of and Experimentation on Tool Interfacing in Software Development Environments". In: *Proceedings of the 3rd Asia-Pacific*

*Software Engineering Conference*. IEEE, Los Alamitos, pp. 56–65 [cit. on pp. x, 46, 48, 55–58].

Zeppenfeld, Klaus and Regine Wolters [2005]. *Generative Software-Entwicklung mit der MDA*. Spektrum, Heidelberg [cit. on pp. 137, 138].

Zhu, Haibin [2005]. "Challenges to Reusable Services". In: *Proceedings of the International Conference on Services Computing, Volume II*. IEEE, Los Alamitos, pp. 243–244 [cit. on p. 120].

Zimmermann, Olaf [2017]. "Microservices Tenets: Agile Approach to Service Development and Deployment". In: *Proceedings of the 10th Advanced Summer School on Service-Oriented Computing*. Computer Science - Research and Development 32.3-4. Springer, Berlin, Heidelberg, pp. 301–310 [cit. on p. 6].

Zimmermann, Olaf, Mark Tomlinson, and Stefan Peuser [2005]. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. 2nd correc. Springer, Berlin, Heidelberg [cit. on pp. 119, 125].