CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

XMI-Import und -Export für *Syspect*

4. Juli 2007

Dominik Denker

Carl von Ossietzky Universität Oldenburg
Fakultät II
Department für Informatik
Abteilung Entwicklung korrekter Systeme

Inhaltsverzeichnis

1 Einleitung

Im Wintersemester 2005 und Sommersemester 2006 wurde *Syspect* [1] im Rahmen einer Projektgruppe an der Carl von Ossietzky Universität Oldenburg entwickelt. Aus Zeitmangel musste darauf verzichtet werden ein XMI-Format [2] zum Speichern und Austauschen der Daten mitzuliefern. Ein XMI-Import und -Export macht es möglich auch zwischen unterschiedlichen *Syspect*-Versionen Projekte auszutauschen und potenziell mit anderen Programmen Modelle auszutauschen. Dieses Projekt möchte die Möglichkeit, per XMI Daten auszutauschen, liefern.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen für dieses Projekt vor. Es wird kurz erklärt, was UML [3] und was XMI ist. Auch werden EMF [4] und *Syspect* näher erklärt, sowie ein paar andere Grundlagen. Sie werden meist nur soweit beschrieben, wie für dieses Projekt notwendig ist.

2.1 UML

UML [3] steht für „**U**nified **M**odeling **L**anguage" und ist eine standardisierte Sprache für die Modellierung von Software und anderen Systemen. Durch verschiedene Diagrammtypen können unterschiedliche Aussagen spezifiziert werden. Solche sind zum Beispiel Anwendungsfalldiagramme zum Darstellen von Nutzungsmöglichkeiten und Zustandsdiagramme, welche die Verhaltensweise einer Klasse zeigen. Neben der grafischen Notation definiert die UML auch die Beziehungen zwischen ihren Komponenten. Zur Zeit ist die UML 2.1.1 die neueste Version, die von der OMG, der **O**bject **M**anagment **G**roup [5], zur Verfügung gestellt wird.

> UML wird durch ein Metamodel, die *Meta Object Facility* (MOF) definiert; diese wird im Wesentlichen durch vier Abstraktionsebenen beschrieben:
>
> - M0-Ebene: Konkret. Ausgeprägte Daten.
> - M1-Ebene: Modelle. Z.B. physikalische oder logische Daten-, Prozess- oder UML- bzw. Objekt-Modelle, die die Daten der M0-Ebene definieren.
> - M2-Ebene: Meta-Modelle. Definieren, wie die Modelle aufgebaut und strukturiert sind.
> - M3-Ebene: Meta-Meta-Modelle (bzw. MOF-Ebene). Abstrakte Ebene, die zur Definition der M2-Ebene herangezogen wird.
>
> *Zitiert aus dem Endbericht der Projektgruppe* [1]

2.1.1 UML-Profil

Ein UML-Profil beschreibt eine Teilmenge der UML und definiert wie man diese für eine spezielle Anwendung, zum Beispiel das UML-Profil für *Syspect* in 2.1.2, interpretieren soll. Sie legt Stereotypen fest und wie sich diese zueinander verhalten. Stereotypen sind spezielle Klassen die festlegen, ob bzw. was für Attribute Klassen haben müssen, und die Anzahl eines bestimmten Types den die Klassen haben dürfen, denen dieser Stereotyp zugewiesen wird. Als Beispiel wird in Abschnitt 2.1.2 das UML-Profil für *Syspect* angeführt.

2.1.2 Das UML-Profil für CSP-OZ-DC

Das UML-Profil für CSP-OZ-DC [6], welches *Syspect* (siehe Abschnitt 2.4) verwendet, basiert auf einer im Rahmen des ForMooS Projektes verfassten Arbeit [7] und wurde von der Projektgruppe auf UML 2.0 angepasst [1].

CSP-OZ-DC ist eine Zusammenstellung aus drei verschiedenen Spezifikationssprachen[1]: Communicating Sequential Processes (CSP), Object-Z und Duration Calculus (DC). CSP ist zur Spezifikation von sogenannten reaktiven Systemen, welche aus mehreren Prozessen bestehen und mit ihrer Umgebung kommunizieren. In CSP-OZ-DC dient der CSP Teil dazu, den Programmablauf innerhalb einer Komponente sowie die Kommunikation zwischen Komponenten zu spezifizieren. Object-Z ist eine Objektorientierte Variante der Spezifikationssprache Z. Sequentielle Systeme lassen sich in Z beschreiben, charakterisiert durch ihren Zustandsraum sowie Transformationen auf diesem. Als dritte Komponente der Spezifikationssprache CSP-OZ-DC dient der Duration Calculus zur Beschreibung von zeitlichem Verhalten. Der DC ist eine intervallbasierte Logik, die in CSP-OZ-DC in eingeschränkter Form, den sogenannten Countertraces, benutzt wird, um bestimmtes Systemverhalten auszuschließen.

Aus dem Gesamtbild der UML wurden drei Diagrammtypen ausgewählt: Klassendiagramme (engl. classdiagrams), Zustandsdiagramme (engl. Statemachines) und Komponentendiagramme (engl. Componentdiagrams). Das Profil definiert für das Klassendiagramm die Stereotypen Capsule, Interface und Data. Capsules sind Erweiterungen der UML eigenen Components, Data-Klassen eine reduzierte Variante der Capsules und Interfaces Schnittstellen zwischen diesen. Diese unterschiedlichen Stereotypen sind notwendig, weil zum Beispiel Interfaces keine Attribute haben können, Capsules aber schon. Capsules und Data-Klassen können „Init", „Invariant" und „ZTypeDefintion" besitzen. Init ist ein Prädikat, das die initialen Werte der Attribute spezifiziert. Invariant ist ein Prädikat, das die Werte der Attribute während des ganzen Systemlaufs spezifiziert. Die ZTypeDefinition ist ein zentraler Ort, um Typen der Sprache Z zu definieren. Interfaces können Methodensignaturen, Capsules und Data-Klassen dagegen Methoden haben. Methodensignaturen können „Input"- „Output"- und „Simple"-Parameter besitzen. Methoden haben eine Methodensignatur, „Changes"-, „Enable"- und „Effekt"-Attribute. Enable ist die Vorbedingung der Methode, Effekt definiert die Änderungen der Methode wenn diese Ausgeführt wird. Changes enthält eine Liste mit den Attributen die geändert werden.

2.2 XML und XMI

XML [9] steht für „e**X**tensible **M**arkup **L**anguage". Sie wird dazu benutzt um Daten plattformübergreifend austauschen zu können. Eine XML-Struktur ist ein Baum. Von einem „root"-Element gehen in einer XML-Datei sogenannte Kinder(engl. Children) ab. Diese Kinder können auch Kinder haben. Jedes Element kann unterschiedliche Attribute haben.

Die Elemente werden als Tags bezeichnet. Tags beginnen mit „<" und enden auf „>". Man unterscheidet zwischen öffnenden, schließenden und alleinstehenden Tags. Schließende Tags beginnen mit „</" und alleinstehende Tags enden auf „/>". Tags können Attribute enthalten.

[1] Die Erklärungen zu den Spezifikationssprachen sind dem Vorlesungsskript [8] entnommen.

Tag-Beispiele:

```
<tag version="1.1">
  <empty value1="a" value2="b" />
</tag>
```

Beispiel für eine XML-Datei:

```
<?xml version="1.0" encoding="UTF-8">
<rootelement attribute="attr">
  <child name="Bill" />
  <child name="Linus">
    <invented>Linux</invented>
  </child>
  <computer>
    <OS>Linux</OS>
  </computer>
</rootelement>
```

Durch die Flexibilität der Sprache kann der eigentliche Inhalt selbst definiert werden. Mit XML-Schemata wird validiert, ob XML-Dateien korrekt sind. Eine nur noch selten benutzte Alternative ist das ältere DTD, „Document Type Definition". Das Schema ist in XML geschrieben, die DTD nicht. Ein Beispiel für ein Schema findet sich im Anhang A.

In einem Schema kann ein „targetNamespace" angegeben werden, um einen Namensraum für das Schema zu definieren. Ein Namensraum, auf Englisch Namespace, ist ein Eingrenzung für den Kontext, in dem die Namen in diesem Namensraum gelten. Das Schema im Anhang A benutzt zum Beispiel die Namensräume „xmlns:uml="http://www.eclipse.org/uml2/2.0.0/UML"" und „xmlns:ext="http://www.xmi.syspect.de/extension"", wobei letzterer gleichzeitig targetNamespace ist.

XMI [2] basiert auf XML und ist ein Austauschformat für Metadaten. Mit ihr ist es möglich, UML-Daten auszutauschen.

2.3 Eclipse

Eclipse [10] ist eine Entwicklungsumgebung für Java, wie sie auch bei *Syspect* und für diese Arbeit benutzt wurde. Es wird aber auch die Verwendung von anderen Programmiersprachen, wie zum Beispiel C++, unterstützt. Sie besteht aus verschiedenen PlugIns, die dann insgesamt die Entwicklungsumgebung bilden. PlugIns sind einzelne Komponenten. Je nachdem welche verwendet werden, können unterschiedliche Aufgaben mit der Entwicklungsumgebung vollbracht werden. Es ist auch möglich PlugIns zu schreiben, die ohne eine laufende Eclipse Instanz laufen. Diese benötigen eine Rich Client Plattform. Diese ist ein Grundgerüst auf der das PlugIn dann laufen kann. Dieses Konzept wurde bei *Syspect* verwendet.

2.3.1 Eclipse Modeling Framework

Das Eclipse Modeling Framework[4] (EMF) ist entwickelt worden, um mit Hilfe von strukturierten Modellen automatisch Quellcode zu erzeugen. Dies geht zum Beispiel

mit einem XML-Schema. Verschiedene Projekte können dann durch dieses gleiche Grundgerüst miteinander benutzt werden, auch wenn ihnen unterschiedlichen Modelle zugrunde liegen. Ein grundlegendes Objekt in dem EMF ist das `EObject`. Dies ist eine Erweiterung der Java-Klasse `Object`.

Bei Nutzung eines XML-Schemas kann durch Angeben eines targetNamespace für die gewünschte Paketstruktur gesorgt werden. Hierzu gibt man beim targetNamespace rückwärts die gewünschte Struktur an und stellt ein „www" voran. Zum Beispiel wird aus „www.xmi.syspect.de/extension" die Paketstruktur de.syspect.xmi.extension, wobei dann im Unterpaket extension die generierten Klassen vorhanden sind.

2.3.2 Das UML2-PlugIn für Eclipse

Das UML2-PlugIn [11, 12, 13] ist ein eigenständiges Projekt, welches das Eclipse-PlugIn-Konzept verwendet und mit Hilfe von EMF generiert wurde. Es liefert eine Implementierung des Metamodells für die UML 2.1 und ist eine Grundlage für die Entwicklung von (UML-)Modellierungswerkzeugen. Es liefert ein XMI-Schema um einen Austausch zwischen den entstehen Werkzeugen zu gewährleisten. Das PlugIn bietet allerdings keinerlei Möglichkeiten grafische Daten zu speichern.

2.4 Syspect

Syspect [1] ist ein an der Carl von Ossietzky Universität Oldenburg entwickeltes Werkzeug zur Modellierung von CSP-OZ-DC (2.1.2), entsprechend einem UML-Profil. (siehe 2.1) Es bietet grafische Editoren für die Bearbeitung der Diagramme und benutzt sowohl das Rich Client Plattform-Konzept als auch das PlugIn-Konzept von Eclipse.

Es existieren bereits Exporte in gebräuchliche Bild-Formate sowie LaTeX. Die einzige Möglichkeit zum Austausch zwischen den *Syspect*-Umgebungen verschiedener Benutzer ist ein sehr implementierungsnahe Speicherformat. Dieses ist zu keinem anderen Programm kompatibel. Es greift direkt auf die Implementierung des Modells zu, anstatt auf das Modellinterface, was dazu führen kann, dass man bei leicht geänderten *Syspect*-Versionen die Projekte nicht mehr in anderen *Syspect*-Umgebungen nutzen kann. Das Modellinterface ist eine Möglichkeit, dass Modell ohne größere Änderungen am restlichen Quellcode austauschen zu können, indem alle anderem Komponenten nur auf die Interfaces zugreifen und das neue Modell, genau wie das alte, diese Interfaces implementiert.

Wie im Endbericht der Projektgruppe [1] näher beschrieben, hat *Syspect* zwei View-Ebenen: View und Viewable. Die View-Ebene ist dabei eine abstraktere Variante. Dort existiert keine Capsule und kein Interface sondern nur eine „Node" die eine Repräsentation für beide darstellt. Die Viewable-Ebene enthält direkt Objekte zur Repräsentation von Capsules und Interfaces.

3 Das Projekt

In diesem Kapitel wird das Projekt betrachtet. Was soll es leisten, wie ist es aufgebaut und wie funktioniert es grundlegend.

3.1 Anforderungsanalyse

Die Aufgabe des Projektes ist es, einen vollständigen XMI-Im- und Export von *Syspect*-Projekten zur Verfügung zu stellen. Durch einfache Bedienung über grafische Hilfsmittel soll es möglich sein, XMI-Dateien zu erhalten, die die vollständigen Modell- und Grafikdaten enthalten. Hierbei soll eine hohe Austauschbarkeit zu anderen Programmen erreicht werden. Da das Projekt eine Erweiterung zu *Syspect* ist und es auf das UML2-PlugIn zurückgreift, bot es sich an, auch dieses als Eclipse-PlugIn zu realisieren. Da dem UML2-PlugIn die Möglichkeit fehlt grafische Daten zu verwalten, muss eine andere Möglichkeit gefunden werden, diese zu portieren. Des Weiteren soll nicht direkt auf die Modellimplementierung zugegriffen werden, sondern auf deren Interfaces um bei einem Wechsel der Modellimplementierung keine Änderungen am XMI-PlugIn machen zu müssen.

3.2 Entwurf

Durch die gegebenen Anforderungen wurde ein Eclipse-PlugIn entworfen, dass die Portierung des Modells mit dem UML2-PlugIn löst. Das UML2-PlugIn hat zwar keine Möglichkeit grafische Daten zu speichern, aber durch die sehr genaue Abbildung der UML wird es von anderen Entwicklern benutzt werden und somit eine Austauschbarkeit der Modelle mit diesen ermöglichen. Durch die vorhandene Abhängigkeit von Eclipse wurde beschlossen die Portierung der grafischen Daten mit einem EMF-Modell zu lösen. Dies wird näher in Abschnitt 3.3.1 erläutert.

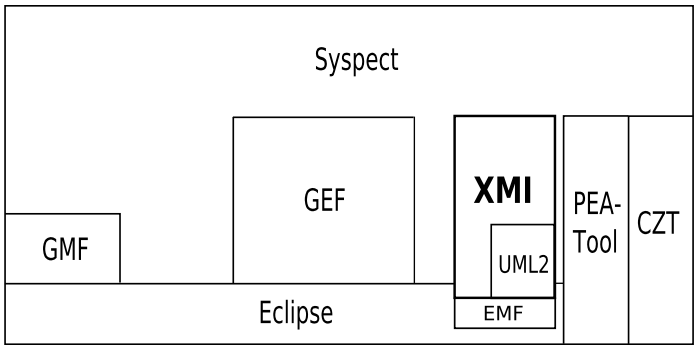

Abbildung 3.1: Syspect Komponenten

Eine vereinfachte Darstellung, wie dieses Projekt mit *Syspect* und anderen Eclipse-PlugIns zusammenhängt, liefert Abbildung 3.1. Nähere Zusammenhänge zwischen *Syspect* und den anderen abgebildeten Komponenten liefert der Endbericht der Projektgruppe [1].

Damit das XMI-PlugIn funktioniert, müssen weitere PlugIns zur Verfügung gestellt werden. Es hat zu diesen sogenannte Abhängigkeiten (engl. Dependencies). Dies sind zu den *Syspect*-PlugIns und zu Eclipse:

- de.syspect.core
- de.syspect.utils
- de.syspect.model
- de.syspect.persistence
- org.eclipse.uml.uml2
- org.eclipse.ui.views
- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.draw2d

Das PlugIn besteht aus mehreren Paketen. (Abb.3.2) Neben Controller und View existieren noch Packages für die Converter, die Exceptions und für die automatisch generierte Extension. (3.3.1) Das Modell befindet sich in einem anderem PlugIn. Controller und View beinhalten wiederum Unterpakete zur genaueren Strukturierung. (Abb. 3.3 & 3.4)

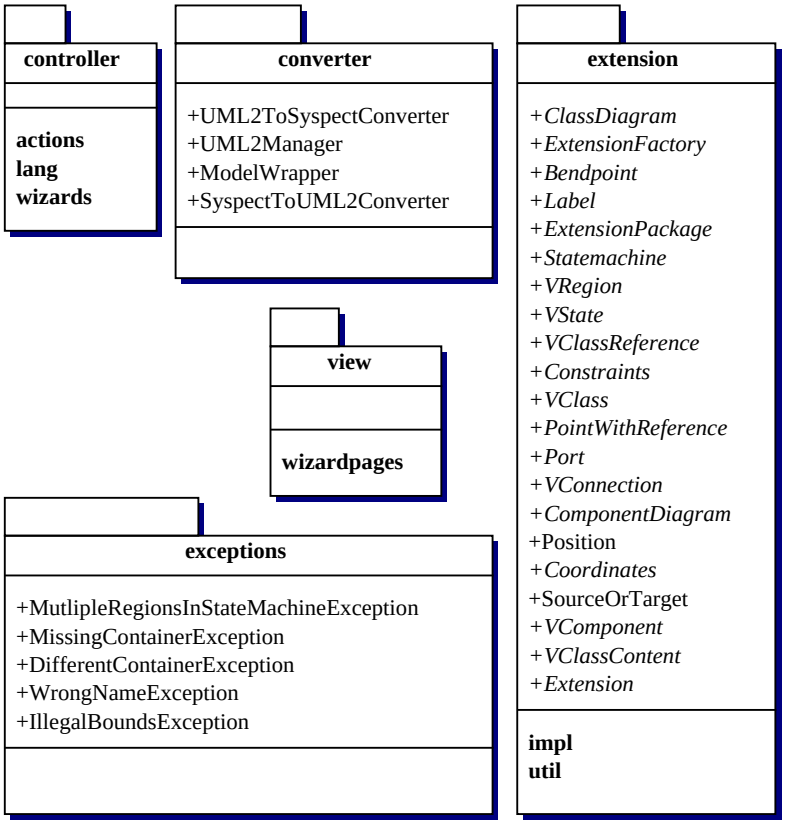

Abbildung 3.2: Die Paketstruktur

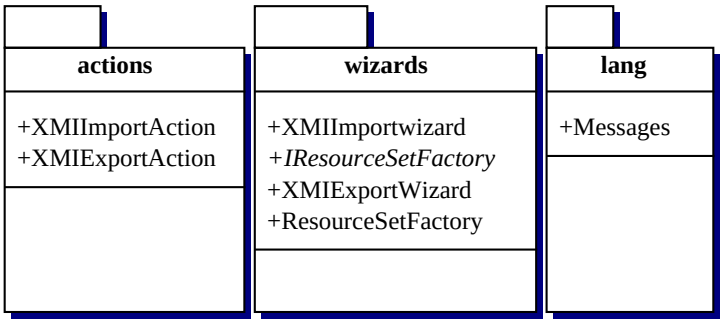

Abbildung 3.3: Die Controller-Unterpaketstruktur

Abbildung 3.4: Die View-Unterpaketstruktur

In einer vereinfachten Betrachtung des XMI-PlugIns kann man die Funktion wie folgt erklären: Mit Hilfe des Exportwizards (Abb. 3.5) wird der `SyspectToUML2Converter` (3.3.4) aufgerufen. Dies ist eine im Rahmen dieses Projektes geschriebene Klasse zur Konvertierung der *Syspect*-Projekte in Modelle des UML2-PlugIns mit grafischen Daten. Das *Syspect*-Modell wird mit Hilfe des `UML2Managers` übersetzt, die grafischen Daten mit der `ExtensionFactory`. (näheres dazu in Kapitel 3.3.1) Die Entwicklung des Datenmodells im Rahmen dieses Projektes für die grafischen Daten hat die Portierung derselben sehr vereinfacht.

Die Initialisierung des Imports ist dem des Exports ähnlich. Durch den `XMIImportWizard` wird der `UML2ToSyspectConverter`(3.3.3), eine andere für das Projekt geschriebene Klasse, gestartet. Dieser benutzt die verschiedenen *Syspect*-Factories, um das *Syspect*-Modell zu erstellen.

Von außen aufgerufen werden kann jeweils nur eine Methode der beiden Converter, welche dann die anderen aufruft. Um auf das UML2-PlugIn zuzugreifen, existiert der `UML2Manager`, welcher in Abschnitt 3.3.2 näher beschrieben wird. Der `ModelWrapper` ist ein Konstrukt um alles, was zum Modell gehört, an einem Ort zugreifbar zu haben. Beim Speichern der XMI-Datei wird ein `ResourceSet` benutzt. Ein `ResourceSet` ist etwas ähnliches wie der `ModelWrapper`, aber nicht von *Syspect* entwickelt, sondern eine offizielle Variante von diesem. Diese war während der Projektgruppe noch nicht bekannt war. Sie kann mehrere `resources` enthalten. Sowohl beim Import als auch beim Export wird eine solche `resource` verwendet. Damit diese `resourcen` jeweils auf die gleichen Fabriken zugreifen, wurde beschlossen, wie auf Abbildung 3.6 zu ver-

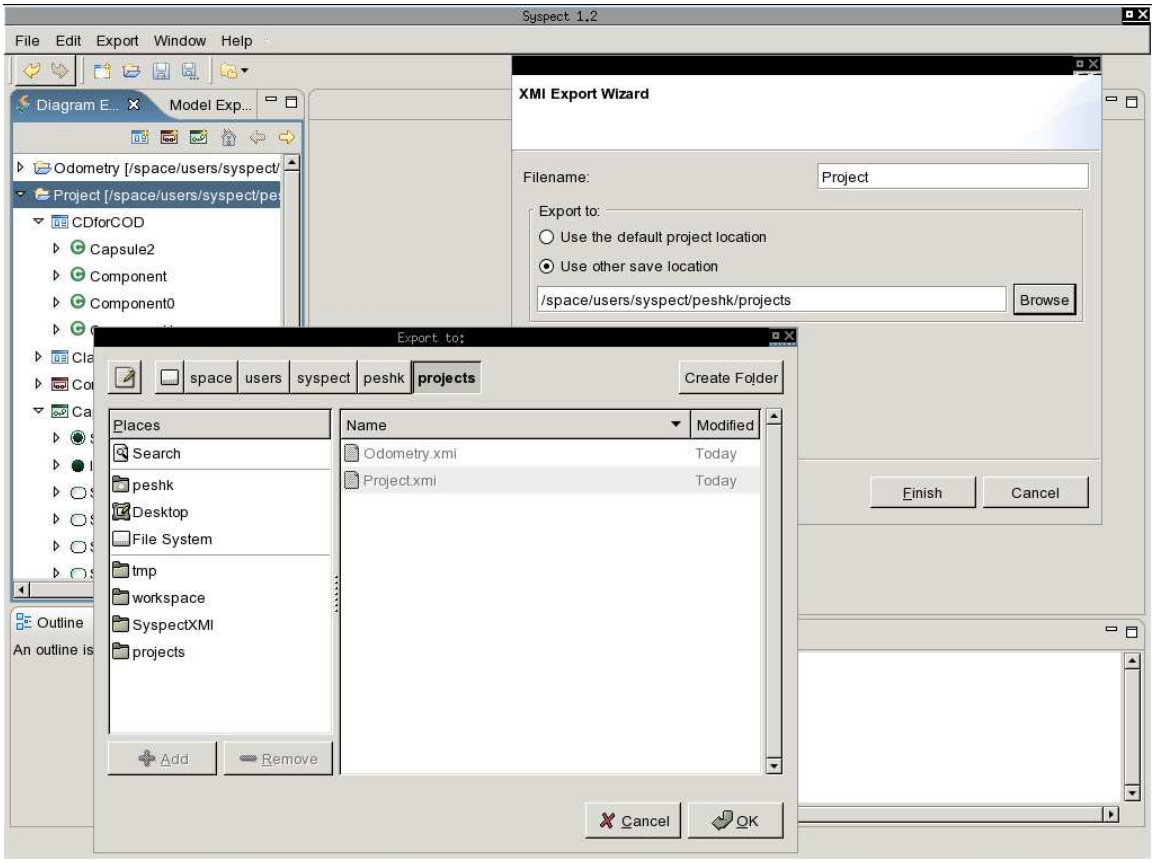

Abbildung 3.5: Der Export-Wizard

fahren. Durch die Abhängigkeiten an dieselbe `ResourceSetFactory` ist sichergestellt, dass keine anderen Daten importiert werden als exportiert wurden, da auf jeden Fall die gleiche Fabrik benutzt wird.

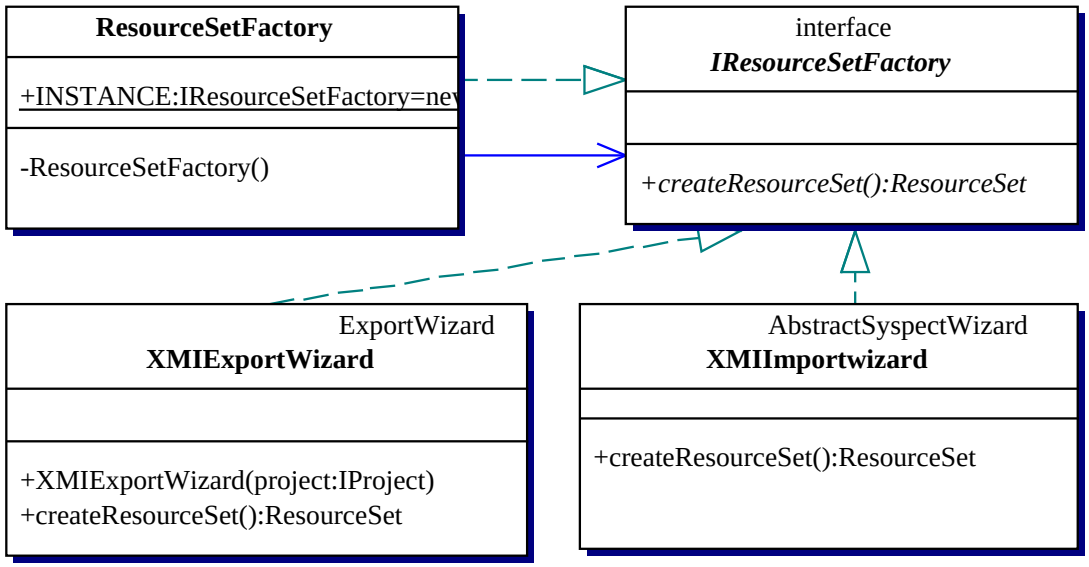

Abbildung 3.6: Das Wizard-Unterpaket

3.3 Implementierung

Die Grafikerweiterung und die drei größten Klassen des Projektes werden im Folgenden näher beschrieben. Hierbei gilt für die beiden Converter, dass fast alle Objekte einen Kommentar haben können. Wenn ein solcher existiert wird für das übersetzte Objekt ein Kommentar mit gleichem Inhalt erstellt. In der weiteren Ausführung wird davon abgesehen, dies bei jedem Vorkommen zu erläutern.

3.3.1 Die Grafik-„Extension”

Das im Anhang A mitgelieferte XML-Schema dient dazu, mit Hilfe von EMF Klassen zu erzeugen, die die grafischen Daten halten können. Hierbei wurde darauf geachtet, dass alle für die Portierung wichtigen Daten abgebildet werden können.

Durch den Eclipsewizard wurden aus dem XML-Schema die Dateien `extension.ecore` und `extension.genmodel` generiert. Der Wizard hilft bei der Erstellung eines neuen Projektes, welches ein EMF-Modell enthalten soll.

Bevor der Code erstellt werden konnte, mussten noch ein paar Änderungen an der `extension.ecore` vorgenommen werden. Das Schema enthält ein „Dummyelement” namens `RefPlaceHolder`. Dieser ist notwendig, weil die Referenzen, die benötigt werden, nur innerhalb des Schemas gesetzt werden können, aber Verweise auf andere Schemata notwendig sind. Aus diesem Grund muss in der `extension.ecore` jeder Verweis auf `RefPlaceHolder`, in einen Verweis auf `EObject` geändert werden. Danach

kann das Kind `RefPlaceHolder` gelöscht werden. Weitere Anpassungen, die vorgenommen wurden, sind die Umbenennung aller Kinder die „Type" im Namen hatten, durch die Löschung dieses Teiles. (Beispiel: Aus CoordinatesType wurde Coordinates) Das Kind `DocumentRoot` ist im Rahmen dieses Projektes nicht notwendig und wurde gelöscht.

Nach diesen Anpassungen konnte aus der `extension.genmodel` Code generiert werden. Durch die Modifikationen des targetNamespace im Schema werden die Paketstrukturen automatisch so erstellt, wie sie im Zielprojekt benötigt werden.

Nach dem Kopieren des `Extension`-Paketes in das Zielprojekt steht der Code zur Verfügung. Eine vereinfachte Darstellung, welche *Syspect*-Elemente auf welche Extensionelemente abgebildet werden, sieht man auf der Abbildung 3.7. Zu Beachten ist dabei, dass zum Beispiel die `VClass` für mehrere Objekte verwendet wird.

3.3.2 Der UML2Manager

Für jedes Element des UML2-PlugIns, das für den Export erzeugt werden muss, enthält der `UML2Manager` eine Methode, die aus den Convertern des Projektes aufgerufen wird. Diese Kapselung ist für bessere Wartbarkeit entworfen worden, da bei Änderungen am UML2-PlugIn nur der Manager angepasst werden muss. Hierbei ist darauf zu achten, dass nicht jedes *Syspect*-Element eine eigene Repräsentation im UML2-PlugIn besitzt. Zum Beispiel werden Data und Capsule beide als `Class` gespeichert. Die Unterscheidung der beiden findet dann mit den im Profil spezifizierten Stereotypen statt. Dieses UML-Profil für *Syspect* wird auch im Manager implementiert. Sämtliche Teilelemente des Profils sind über eine Konstante des Managers zugreifbar. Dadurch ist gesichert, das Import und Export die Elemente mit den gleichen Namen ansprechen.

3.3.3 Der SyspectToUML2Converter

Der SyspectToUML2Converter ist für den Export zuständig. Während der Portierung der Daten muss auf die Reihenfolge geachtet werden, damit alle Daten existieren, auf die verwiesen wird.

Der Export des Modells

Es wird als erstes eine Instanz der Klasse `ModelWrapper` erstellt, um den Wrapper und das Modell zu haben, die bei der Instanziierung erzeugt werden. Der Wrapper ist zur Kapselung der Daten (Modell, Profil und Grafik-Extension), das Modell ist ein Objekt, in das alle weiteren Modelldaten gespeichert werden. Dem Modell wird der *Syspect*-Stereotyp aus dem *Syspect*-Profil zugewiesen. Die *Syspect* Interfaces werden zuerst übersetzt, da *Syspect*-Klassen Methodensignaturen von den Interfaces erben können. Aus diesem Grund werden Klassen und Interfaces in unterschiedliche Listen aufgeteilt. Danach wird die Übersetzung der Interfaces gestartet.

Während dieses Übersetzungsschrittes werden UML2-PlugIn Repräsentationen der Interfaces erstellt, die richtigen Stereotypen angefügt und zur späteren Verwendung eine `sysAClassToUmlClassifierMap` erstellt die von den *Syspect*-Interfaces auf die UML2-PlugIn-Interfaces verweisen. Diese Map wird später auch verwendet, um eine

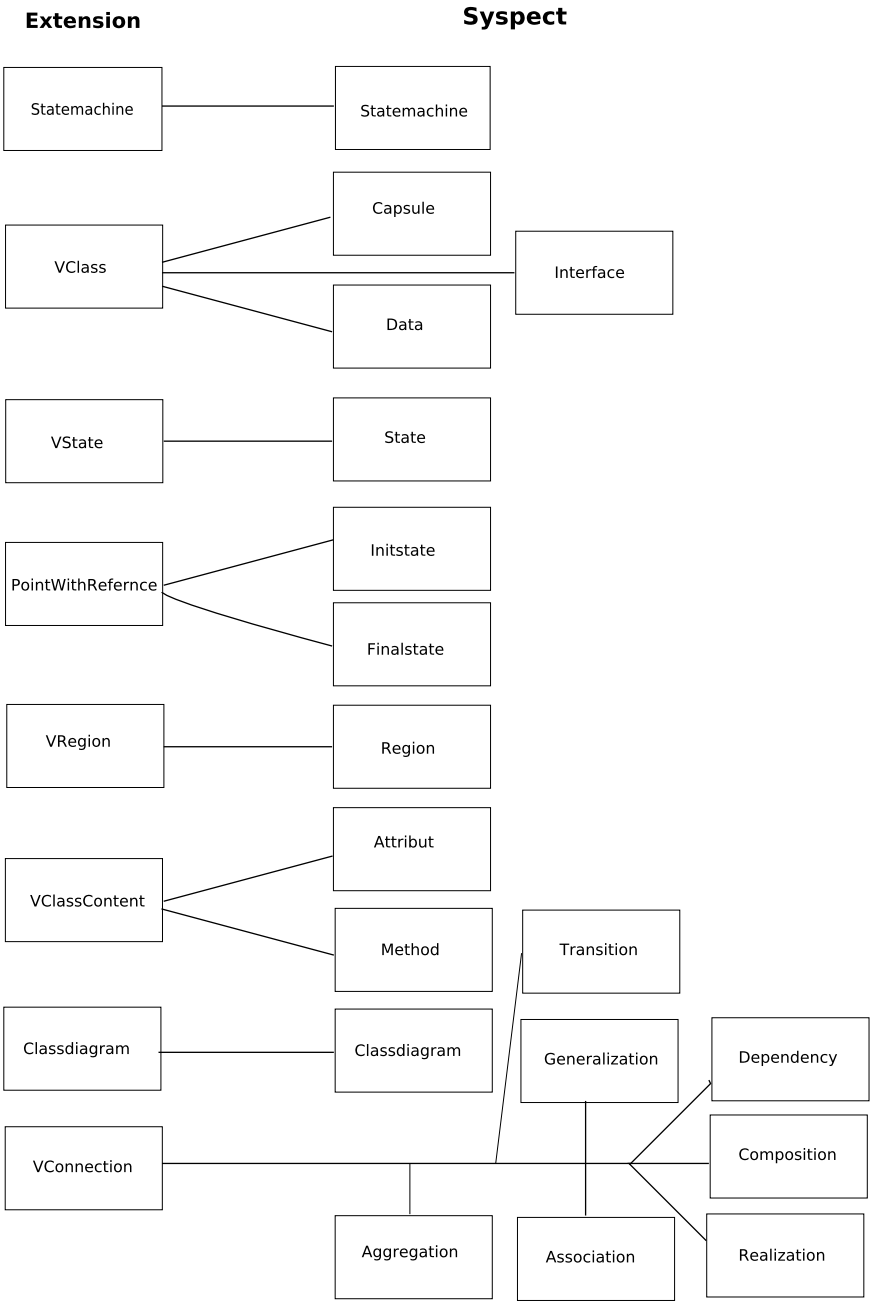

Abbildung 3.7: Vereinfachte Darstellung der Abbildung von Extensionelementen auf Syspectelemente

äquivalente Referenzierung von Klassen zu haben. Des Weiteren werden die Methodensignaturen und zwei weitere Maps für spätere Verwendung erstellt. Zum einen die `sysMethodSigToUmlInterfaceMap`, um von originalen Methodensignaturen auf die neuen Interfaces zu verweisen. Dies wird benötigt um bei der Erstellung der *Syspect*-Klassen-Methoden auf schon vorhandene Methodensignaturen zu verweisen. Zum anderem die `sysClassContentToUMLFeatureMap` um die `toModelReference` der grafischen Repräsentation der Klasseninhalte zu setzen. Für die Methodensignaturen werden dann die Parameter erstellt. In den meisten Schritten werden verschiedene Stereotypen an die erstellten Objekte angefügt, um sie an die Bedürfnisse von *Syspect* anzupassen.

Ist dieser Schritt durchlaufen werden die Klassen übersetzt. Für jede *Syspect*-Klasse wird eine UML2-Klasse erstellt. Diese wird, wie auch schon die Interfaces, in der `sysAClassToUmlClassifierMap` referenziert. Anders als bei den Interfaces bekommt jede dieser Klassen zwei Stereotypen. An jede erstellte Klasse wird der Stereotyp „CLASS" angefügt, und, je nachdem ob das Original eine Capsule- oder eine Data-Klasse war, der dazugehörige Stereotyp. Diese Verwendung von mehrfachen Stereotypen ist nötig, da das Generalisieren von einem Stereotypen zum anderen zu Problemen führt.

Die Erzeugung der Inhalte von Capsules und Datas läuft ähnlich ab. Es werden „Init", „Invariant" und „ZTypeDefintion" gesetzt und dann Attribute und Methoden übersetzt. Die Attribute werden in der `sysClassContentToUMLFeatureMap` gespeichert. Die Typen der Attribute werden dabei in einer Map referenziert, um jeden Typ nur einmal speichern zu müssen. Bei der Erstellung der Methoden wird darauf geachtet, ob die Methodensignatur von einem Interface geerbt ist, oder ob es eine Methode ist, die nur zu dieser Klasse gehört. Hierbei wird die `sysMethodSigToUmlInterfaceMap` verwendet. Ist die Signatur noch nicht vorhanden, wird sie erstellt, ansonsten wird sie dupliziert. Leider geht hierbei die Information verloren, dass es sich um dieselben Signaturen handelt. Bei der Erweiterung von den Signaturen auf Methoden gehen zur Zeit noch die „Changes" verloren. Bei den Capsules können zusätzlich DC-Countertraces und Testformeln und eine `Statemachine` eingebaut werden.

Bei der Übersetzung der `Statemachine` musste wieder auf die Reihenfolge geachtet werden. Nach der Erstellung der `ProtocolStateMachine` muss eine Region erstellt werden, ohne dass es für diese eine Repräsentation im *Syspect* Projekt gibt. Dies ist wichtig, weil in der UML bei jedem Zustandsdiagramm eine alles kapselnde Region existiert, in *Syspect* diese aber weggelassen wurde. Auch für die `Statemachine` existiert eine Map: `sysStatemachineToUmlPStateMachineMap`. Nachdem der zugehörige Kommentar übersetzt wurde, werden alle Zustände übersetzt, inklusive der Zustände in den inneren Regionen. Dies ist nötig damit alle Zustände schon bei der Übersetzung der Verbindungen existieren, da auf sie als Start- oder Endpunkt verwiesen wird. Es werden alle Zustände, inklusive der Sonderformen Intial- und Endzustand, erstellt. Enthält ein Zustand eine Region, wird für jede Region eine Region im neu erstellten Zustand erstellt und die Methode wird erneut aufgerufen (Rekursion). Bei der Übersetzung der Transitionen werden nur alle von Zuständen ausgehenden Transitionen übersetzt. Da jede Transition an einem Zustand ankommt und von einem wegführt, würde eine Übersetzung von beiden zu einem Duplikat führen.

Aus dem selben Grund werden als letztes Element des Modells die Verbindungen übersetzt. Hierbei wird die `sysAClassToUmlClassifierMap` benutzt, um Start- und

Zielzustand der Verbindung zu ermitteln. Jede erstellte Verbindung wird in der `sysConnectionToUMLElementMap` referenziert. Dies wird später beim Exportieren des grafischen Teils benötigt.

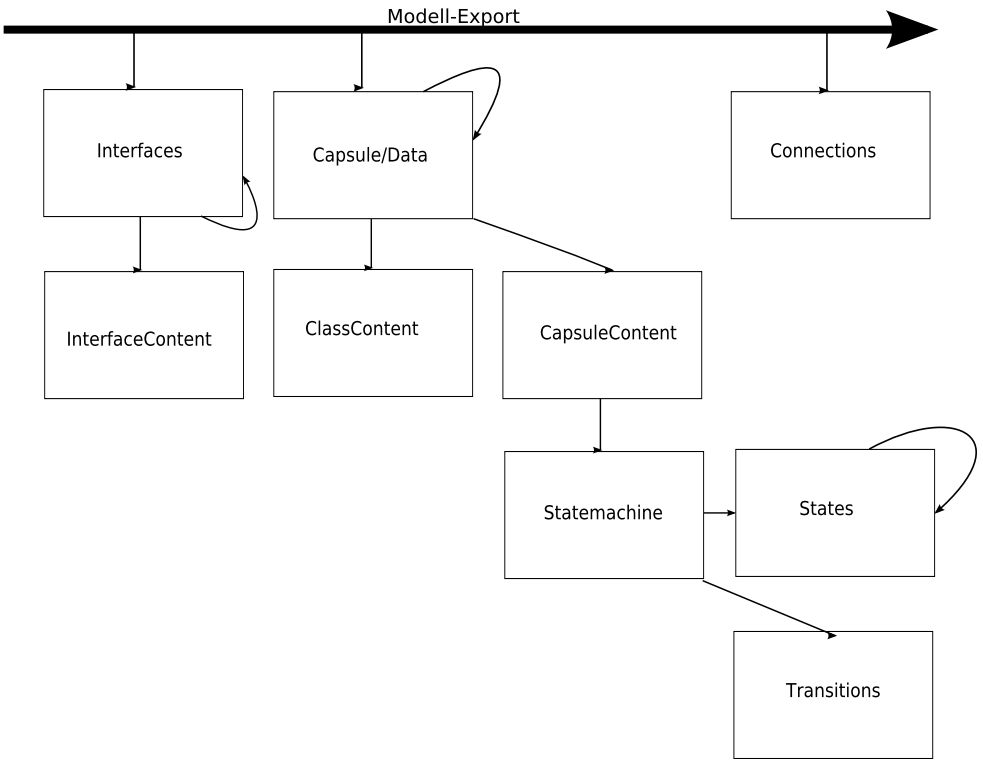

Abbildung 3.8: Vereinfachte Darstellung des Modell-Export-Ablaufs

Der Export der grafischen Daten

Nach den Modellelementen wird der grafische Teil exportiert, indem die eigens dafür erstellte Extension benutzt wird. Die Extension bekommt ein „Extender"-Attribut, in diesem Fall „syspect", um anderen Programmen mitzuteilen, dass dieser Teil des XMI-Dokumentes nicht wichtig für sie ist. Danach werden die `Classdiagrams` und die `Statemachines` aus dem Projekt exportiert. Anders als bei der Modell-Übersetzung werden hier Diagramme erzeugt. Jedes Grafikelement bekommt während des Exportvorgangs ein Modellelement zugewiesen. Ausgenommen davon sind die Repräsentationen der `Classdiagrams`, da für diese kein Modellement existiert. Zusätzlich werden für alle Diagramme deren Namen gespeichert.

Da die `VClass` der Extension die grafische Darstellung für sowohl die Interfaces und Capsules als auch für die Data-Klassen ist, wird diese erstellt und dann der richtigen Liste des Diagramms hinzugefügt. Für jede der drei Arten existiert eine Liste. Alle ausgehenden Verbindungen jedes Diagrammkindes werden in einem Set gespeichert, um später auf diese zugreifen zu können, ohne alles noch einmal durchlaufen zu müssen. Während der Erzeugung der `VClass` werden Daten wie Höhe, Breite und Koordinaten gesetzt und die schon erwähnte Referenz auf ihr Modellelement. Hierfür wird die `sysAClassToUmlClassifierMap` verwendet. Auch wird in diesem Schritt die

Erzeugung des Inhaltes der `VClass` in Gang gesetzt. Die Attribute und Methoden werden erzeugt und bekommen ihre Referenz und werden jeweils der für sie zuständigen Liste der `VClass` hinzugefügt. Als letztes wird die `VClass` der `sysVClassToExtVClassMap` hinzugefügt, welche später beim Export der grafischen Repräsentation der Verbindungen benötigt wird. Bei der Erstellung der `VConnections` werden Anfangs- und Endpunkt gespeichert, so wie die Referenz auf die Modellrepräsentation. Es werden auch sämtliche „Bendpoints", Zwischenpunkte der Verbindungen, mitgespeichert. Da beim Import die Art der Verbindung über die Referenz auf das Modellelement bekannt ist, muss nicht gespeichert werden, um was für eine Art Verbindung es sich handelt. Bei der Speicherung der Anfangs- und Endpunkte werden jeweils Referenzen auf die richtige `VClass` gesetzt. Hierzu wird die `sysVClassToExtVClassMap` verwendet. Als letzter Schritt wird die fertig erstelle Grafikrepräsentation des `Classdiagrams` der Liste der `Classdiagrams` der `Extension` hinzugefügt.

Die Extension entspricht bei den Zustandsdiagrammen eher der UML als *Syspect*. Dies äußert sich darin, dass `Statemachines` immer eine alles kapselnde äußere Region haben. Diese Region speichert die Referenz auf das Modell der `Statemachine`. Dies ist so gelöst, weil Regionen immer eine Referenz auf ihr Modell besitzen, aber für diese kein Modellelement existiert. So muss die `Statemachine` Klasse der `Extension` ein Element weniger speichern, und man umgeht das Problem, dass genau eine Region keine Referenz braucht. Bei der Erstellung der `VStates` werden wieder Höhe, Breite und Koordinaten gespeichert und die Referenz auf das Modell gesetzt. Bei vorhandenen Regionen werden auch für diese grafische Repräsentationen erstellt und wie bei der Erstellung des Modells per Rekursion deren Kinder erzeugt. Da Initial- und Endzustände keine Regionen enthalten können, werden sie als `PointWithReference`, einem weiterem Element der Extension, anstatt als `VState`, behandelt. Sie haben Höhe, Breite und Koordinaten und eine Referenz auf ihr Modell. Zum Zeitpunkt der Arbeit war es in *Syspect* möglich, Höhe und Breite aller Zustände zu verändern, obwohl diese bei Initial- und Finalzuständen eigentlich fix sein sollten. Aus Gründen der Vollständigkeit werden diese Werte daher mitgespeichert. Jede Art von Zustand wird der für sie vorgesehenen Liste der `Statemachine` hinzugefügt. Ist alles erstellt, wird die Extension dem Modelwrapper hinzugefügt.

Die einzelnen Teile des ModelWrappers werden dann einem ResourceSet hinzugefügt und als XMI gespeichert.

3.3.4 Der UML2ToSyspectConverter

Am Anfang des Imports werden die Daten aus der zu importierenden Datei in eine `Resource` geladen und es wird ein neues *Syspect*-Projekt erzeugt.

Die erhaltene `resource` wird im Converter in seine Bestandteile aufgeteilt und diese zwischengespeichert. Derzeit werden Modell und Extension herausgegriffen. Es wäre auch möglich das Profil auszulesen, allerdings ist das bisher nicht nötig. Problem beim Datenformat des Modells ist, dass es noch in JAVA 1.4 entwickelt wurde und somit keine typisierten Listen hat. Aus diesem Grund muss in sehr vielen Fällen abgefragt werden, was für ein Objekt man gerade erhalten hat, und dieses dann typisiert werden.

Der Import des Modells

Das Modell wird wie auch beim Export als erstes übersetzt. Es werden Listen mit Klassen und Interfaces erstellt, die danach abgearbeitet werden. Die Verbindungen werden als letztes generiert, damit alles, auf das sie Referenzen benötigen, schon existiert.

Als erstes Modellelement werden die Interfaces erstellt. Sie bekommen den Namen der original Klasse und ihr Inhalt wird erstellt. Ganz wichtig für *Syspect*: die „Parent und Child"-Beziehungen werden gesetzt. Dies wird der Einfachheit halber in der weiteren Arbeit nicht mehr explizit erwähnt, ist aber bei fast allen Objekten nötig. Außerdem wird, wie auch schon beim Export, wieder mit Maps gearbeitet und die neuen Interfaces in der `umlClassifierToSysAClassMap` referenziert. Der Inhalt besteht aus einem Kommentar und Methodensignaturen. Die Methodensignaturen werden in einer Map referenziert, wo man den Namen der Methode als Schlüssel benutzt. Dies wird später bei der Erstellung der Methoden in Klassen benötigt.

Bei der Übersetzung der Klassen wird unterschieden, ob es sich um eine Data oder eine Capsule handelt. Dies geschieht über die Stereotypen. Nach der Erstellung verläuft die Übersetzung von Data und Capsule sehr ähnlich. Bei beiden wird der Name gesetzt und „Init", „Invariant" und die „ZTypeDefinition" übersetzt. Der Kommentar, Attribute und Methoden werden übersetzt. Beide werden für spätere Benutzung in jeweils einer Map referenziert. Bei den Methoden wird darauf geachtet, ob Methodenisignaturen mit einem solchen Namen schon existieren. In diesem Fall wird die schon existente Signatur als Signatur für die Methode der Klasse referenziert. Dies funktioniert, weil als erstes die Interfaces samt Inhalt übersetzt wurden. Durch dieses Verfahren ist der Verlust des Wissens, dass es ich bei Vererbung der Methoden von Interfaces zu Klassen um die selben Methoden handelt, aufgefangen. Zusätzlich werden bei den Capsules DC-Countertraces, Testformeln und falls nötig `Statemachines` übersetzt.

Bei den `Statemachines` werden wie zuvor Name und Kommentar gesetzt. Dann werden ihre Kinder übersetzt. Da die `Statemachine` aus dem UML2-PlugIn ist, ist ihr einziges direktes Kind eine Region. Dies wird ausgenutzt, um leicht eine Lösung mit Rekursion aufzubauen. Alle Zustände werden übersetzt und in einer Map referenziert. Die Zustände, die Regionen enthalten können, werden durch rekursiven Aufruf der Methode nach gleichem Schema erstellt. Transitionen sind auch Kinder der Regionen, werden aber zur späteren Verarbeitung in einer Liste gespeichert und nicht gleich übersetzt.

Verbindungen werden in drei Schritten importiert. Da manche Verbindungsarten in der XMI-Datei an den Klassen und Interfaces gespeichert werden und manche direkt im Modell, müssen diese unterschiedlich behandelt werden. Diese beiden Arten werden zuerst übersetzt, als Drittes dann die Transitionen. Alle im Modell übersetzen Verbindungen werden in Maps auf ihre Modellelemente referenziert.

Der Import der grafischen Daten

Nach der kompletten Übersetzung der vorhandenen Modellelemente werden die grafischen Daten übersetzt.

Die Diagramme werden in der Reihenfolge übersetzt in der sie in der XMI-Datei stehen. Bei allen Grafikelementen muss darauf geachtet werden, dass nicht nur die

„Parent-Child"-Beziehung gesetzt wird, sondern auch das `View` und `Viewable` beide erzeugt und einander bekannt sind.

Ist das Element ein `Classdiagram` werden die benötigten Elemente, `View` und `Viewable`, erzeugt und der Name gesetzt. Dann wird der Inhalt des Diagramms erstellt. Es werden Höhe, Breite und Koordinaten des Diagrammkindes gesetzt, sowie die Referenz auf das zugehörige Modellement. Beim Setzen der Referenz werden immer die vorher erstellen Maps benutzt. Im gleichen Schritt werden noch die grafischen Elemente für Attribute und Methoden erzeugt, wobei unterschieden wird zwischen der Erstellung von grafischen Elementen für Methodensignaturen und von Methoden. Verbindungen werden durch ihre `toModelReference` unterschieden und somit die richtige `View` für das entsprechende Modell erstellt. Die Verbindungen erhalten ihre ursprünglichen Bendpoints, Labels und Anfangs- und Endpunkt werden gesetzt. Auch hier beim Setzen der Anfangs- und Endpunkte werden die vorher erstellten Maps benutzt.

Bei der Erstellung der grafischen Daten der `Statemachine` wird die `toModelReference` der ersten Region als Referenz zwischen Modellelement und grafischer Repräsentation der `Statemachine` benutzt. Danach werden die Kinder, Transitionen und alle Arten von Zuständen, dieser Region und eventuelle Regionen in den Kindern rekursiv erzeugt. Auch hier werden Höhe, Breit,e Koordinaten und Modellelement gesetzt.

3.4 Tests

Getestet wurde an verschiedenen *Syspect*-Projekten, mit jeweils unterschiedlichen Inhalten. Diese waren voll zufriedenstellend. Es war geplant, mit dem Omondo- Produkt „EclipseUML" zu testen. Allerdings ist es mit mehrfachen Versuchen nicht gelungen die, kostenfrei verfügbare Version in einen Zustand zu bringen, mit dem ein Test möglich gewesen wäre.

4 Fazit

Das Projekt hat einen Grossteil der Anforderungen erfüllt. Es können *Syspect*-Projekte in einem XMI-Format exportiert und wieder importiert werden. Es wurde ein Weg gefunden, die grafischen Daten in ein einfaches Datenmodell zu übersetzen und dieses in der gleichen Datei wie die Modell-Daten zu speichern.

Zur Zeit ignorieren sowohl Im- als auch Export die Componentdiagramms. Diese müssen noch integriert werden. Dafür wurden sie schon beim Erstellen der Extension für die grafischen Daten mit berücksichtigt. Es muss auch noch ein Verfahren gefunden werden, die „Changes" zu portieren.

Auch könnte der `ModelWrapper` durch eine Lösung, die direkt auf ResourceSets arbeitet, ersetzt werden, um dort ein standardisiertes Verfahren zu nutzen.

A XML-Schema für die Grafik-Erweiterung

Das hier abgebildete Schema wurde im Rahmen dieses Projektes entwickelt, um die grafischen Daten von Syspect in einem Datenmodell speichern zu können. Aus ihm wird, wie in Anhang B beschrieben, ein EMF-Datenmodell erzeugt. Eine vereinfachte Darstellung welche Syspectelemente auf welche Extensionelemente abgebildet werden findet man in Abbildung 3.7.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema  targetNamespace="http://www.xmi.syspect.de/
   extension" xmlns:ext="http://www.xmi.syspect.de/extension"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ecore="
   http://www.eclipse.org/emf/2002/Ecore" xmlns:uml="http://
   www.eclipse.org/uml2/2.0.0/UML">

<xsd:complexType name="ClassDiagram">
  <xsd:choice minOccurs="0" maxOccurs="unbounded" >
    <xsd:element name="capsule" type="ext:VClass" />
      <xsd:element name="data" type="ext:VClass" />
      <xsd:element name="interface" type="ext:VClass" />
    <xsd:element name="vConnection" type="ext:VConnection" />
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="comment" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="Statemachine">
  <xsd:all>
    <xsd:element name="region" type="ext:VRegion" />
  </xsd:all>
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="ComponentDiagram">
  <xsd:choice>
    <xsd:element ref="ext:toModelReference" />
    <xsd:element name="component" type="ext:VComponent"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="require" type="ext:PointWithReference"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="provide" type="ext:PointWithReference"
       minOccurs="0" maxOccurs="unbounded" />
```

```
    <xsd:element name="delegate" type="ext:VConnection"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="port" type="ext:Port" minOccurs="0"
       maxOccurs="unbounded" />
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="VClass">
  <xsd:choice>
    <xsd:element ref="ext:toModelReference" />
    <xsd:element ref="ext:constraints" />
    <xsd:element name="vAttribute" type="ext:VClassContent"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element name="vMethod" type="ext:VClassContent"
       minOccurs="0" maxOccurs="unbounded" />
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="VComponent">
  <xsd:choice>
    <xsd:element ref="ext:toModelReference" />
    <xsd:element ref="ext:constraints" />
    <xsd:element name="component" type="ext:VComponent"
       minOccurs="0" maxOccurs="unbounded"  />
    <xsd:element name="require" type="ext:PointWithReference"
       minOccurs="0" maxOccurs="unbounded"  />
    <xsd:element name="provide" type="ext:PointWithReference"
       minOccurs="0" maxOccurs="unbounded"  />
    <xsd:element name="delegate" type="ext:VConnection"
       minOccurs="0" maxOccurs="unbounded"  />
    <xsd:element name="port" type="ext:Port" minOccurs="0"
       maxOccurs="unbounded"  />
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="VState">
  <xsd:choice >
    <xsd:element ref="ext:toModelReference" />
    <xsd:element ref="ext:constraints" />
    <xsd:element name="region" type="ext:VRegion" minOccurs="0
       " maxOccurs="unbounded" />
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="VRegion">
  <xsd:choice >
    <xsd:element ref="ext:toModelReference" />
    <xsd:element name="initState" type="ext:PointWithReference
       " minOccurs="0" maxOccurs="unbounded" />
```

```
    <xsd:element name="finalState" type="
       ext:PointWithReference" minOccurs="0" maxOccurs="
       unbounded" />
    <xsd:element name="state" type="ext:VState" minOccurs="0"
       maxOccurs="unbounded" />
    <xsd:element name="transition" type="ext:VConnection"
       minOccurs="0" maxOccurs="unbounded" />
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="PointWithReference">
  <xsd:all>
    <xsd:element ref="ext:toModelReference" />
    <xsd:element ref="ext:constraints" />
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="Port">
  <xsd:all>
    <xsd:element ref="ext:coordinates" />
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="Bendpoint">
  <xsd:all>
    <xsd:element ref="ext:coordinates" />
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="VConnection">
  <xsd:choice >
    <xsd:element ref="ext:toModelReference" />
    <xsd:element name="bendpoint" type="ext:Bendpoint"
       minOccurs="0" maxOccurs="unbounded" />
    <xsd:element ref="ext:ending" minOccurs="2" maxOccurs="2"
        />
    <xsd:element ref="ext:label" minOccurs="0" maxOccurs="
       unbounded"  />
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="VClassContent">
  <xsd:all>
    <xsd:element ref="ext:toModelReference" />
  </xsd:all>
  <xsd:attribute name="typeVisible" type="xsd:boolean" />
  <xsd:attribute name="isSignature" type="xsd:boolean" />
</xsd:complexType>

<xsd:complexType name="Extension">
```

```
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="classdiagram" type="ext:ClassDiagram"
       minOccurs="0" />
    <xsd:element name="Statemachine" type="ext:Statemachine"
       minOccurs="0" />
    <xsd:element name="componentdiagram" type="
       ext:ComponentDiagram" minOccurs="0" />
  </xsd:choice>
  <xsd:attribute name="extender" type="xsd:string" use="
     optional" />
  <xsd:attribute name="extenderID" type="xsd:string" use="
     optional" />
</xsd:complexType>

<xsd:element name="toModelReference" type="xsd:IDREF"
   ecore:reference="ext:RefPlaceHolder" />

<xsd:element name="ending">
  <xsd:complexType>
    <xsd:choice>
       <xsd:element name="toVClassReference" type="xsd:IDREF"
          ecore:reference="ext:VClass"/>
           <xsd:element name="toVStateReference" type="
              xsd:IDREF" ecore:reference="ext:VState"/>
       <xsd:element name="toVPointWithRerefenceReference" type
          ="xsd:IDREF" ecore:reference="ext:PointWithReference
          "/>
    </xsd:choice>
  <xsd:attribute name="type" type="ext:sourceOrTarget" />
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="sourceOrTarget">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="source" />
    <xsd:enumeration value="target" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="position">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="source" />
    <xsd:enumeration value="target" />
    <xsd:enumeration value="center" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name="coordinates">
  <xsd:complexType>
    <xsd:attribute name="x" type="xsd:integer" />
```

```
    <xsd:attribute name="y" type="xsd:integer" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="constraints">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="ext:coordinates" />
    </xsd:all>
    <xsd:attribute name="height" type="xsd:integer" />
    <xsd:attribute name="width" type="xsd:integer" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="label">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="ext:coordinates" />
    </xsd:all>
    <xsd:attribute name="label" type="xsd:string" />
    <xsd:attribute name="position" type="ext:position" />
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="RefPlaceHolder">
</xsd:complexType>

</xsd:schema>
```

B HOWTO: Wie mit Eclipse das „Extension"-EMF-Modell aus dem XML-Schema generiert wird

Das im Anhang A mitgelieferte XML-Schema wurde erstellt, um über EMF Klassen zu erzeugen, die die grafischen Daten halten können. Hierbei wurde darauf geachtet, dass alle wichtigen Daten, die für die Portierung wichtig sind, abgebildet werden können. Auch die noch nicht implementierte Componentdiagrams-Portierung wurde hierbei schon berücksichtigt.

Zur Erstellung des Codes geht man, sobald man ein Schema hat, wie folgt vor:

File > New > Project > Eclipse Modeling Framework > EMF Project. Nach klicken auf „Next" muss ein Name eingegeben werden. Dieser ist aber nur wichtig um im nächsten Schritt die importierten Daten wiederzufinden. Nach einem weiteren Klicken auf „Next" wählt man nun aus, aus was man sein „EcoreModel" erstellen möchte. In diesem Fall „XML Schema". Nach einem weiterem „Next" muss nun das Schema ausgewählt werden. Das Schema dieses Projektes liegt im PlugIn Ordner unter „resources". Nach einem weiteren „Next" und dem „Finish" werden zwei Dateien erstellt, mit denen dann weitergearbeitet werden kann. Die Dateien sind im Package Explorer in dem Projekt mit dem Namen, der eben Vergeben wurde, im Paket „Model" zu finden und heißen, sofern nicht anders angegeben: `extension.ecore` und `extension.genmodel`.

Bevor der Code erstellt werden kann, müssen noch ein paar Änderungen an der `extension.ecore` vorgenommen werden.

Das Schema enthält einen `RefPlaceHolder`. Dieser ist notwendig, weil Referenzen in der Art wie dieses Projekt sie benötigt innerhalb des Schema nicht auf andere Dateien verweisen können. Das nötig wäre, um die Referenzen auf die benötigten Objekte verweisen zu lassen. Aus diesem Grund muss nach dem Einlesen des Schema in Eclipse alles, was auf `RefPlaceHolder` verweist, auf `EObject` geändert werden. Dies geht in den Preferences unter „EType".Danach kann das Kind `RefPlaceHolder` gelöscht werden. Weitere Anpassungen, die vorgenommen werden müssen, sind die Umbenennung aller Kinder die „Type" im Namen haben. Dieser Teil des Namens wurde, wiederum in der Propertiesview, gelöscht. (Beispiel: Aus CoordinatesType wurde Coordinates) Das Kind `DocumentRoot` ist im Rahmen dieses Projektes nicht notwendig, und wurde gelöscht.

Nach diesen Anpassungen kann in der `extension.genmodel` per Rechtsklick auf das oberste Element „Generate Modell Code" der gewünschte Code erzeugt werden. Nun ist im Projektordner ein „src"-Package vorhanden, welches unseren gewollten Code enthält. Durch die Modifikationen des targetNamespace im Schema sind sogar die gewünschten Paketstrukturen schon vorhanden. Jetzt muss nur noch das `Extension`-Paket in das richtige Projekt kopiert werden, und dann steht der Code zur Verfügung.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Projektgruppe Syspect. Endbericht. http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf.

[2] OMG. MOF 2.0/XMI Mapping Specification, v2.1. http://www.omg.org/technology/documents/formal/xmi.htm.

[3] http://www.uml.org.

[4] http://www.eclipse.org/modeling/emf/.

[5] http://www.omg.org.

[6] J. Hoenicke and E.-R. Olderog. CSP-OZ-DC: A Combination of Specification Techniques for Processes, Data and Time. *Nordic Journal of Computing*, 9(4):301–334, 2002. appeared March 2003.

[7] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In E. Boiten, J. Derrick, and G. Smith, editors, *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286. Springer-Verlag, March 2004.

[8] E.-R. Olderog. Spezifikation von Daten und Prozesse mit Z und CSP, 2003. Skript zur Vorlesung.

[9] W3C. Extensible Markup Language (XML) 1.0 (fourth edition). http://www.w3.org/TR/REC-xml/.

[10] http://www.eclipse.org/.

[11] Eclipse Community. MDT-UML2-UML. http://wiki.eclipse.org/index.php/MDT-UML2-UML.

[12] Kenn Hussey. Getting Started with UML2. http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Getting_Started_with_UML2/article.html.

[13] Kenn Hussey. Introduction to UML2 Profiles. http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Introduction_to_UML2_Profiles/article.html.

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt zu haben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

Oldenburg, den 4. Juli 2007

_____