CARL VON OSSIETZKY universität OLDENBURG

Fakultät II — Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Verifying Properties of Processes, Data, and Time:

Linking Counterexamples to High-Level Specifications

Diplomarbeit

vorgelegt von

Ulrich Hobelmann

Erstgutachter: Prof. Dr. Ernst-Rüdiger Olderog

Zweitgutachter: Dipl.-Inform. Johannes Faber

Oldenburg, April 16, 2007

I hereby certify that the work presented in this thesis is my own and that work performed by others is appropriately cited.

Ich versichere hiermit, diese Arbeit selbständig verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfsmittel bedient zu haben.

Signature of the author:

Oldenburg, April 16, 2007

abstract

Software is ubiquitous in our world and is also becoming prevalent in the realm of mission-critical devices. To prevent lives from being endangered, it is crucial to verify the correctness of such software systems. CSP-OZ-DC is a combination of specification techniques for processes, data, and real-time behavior that admits translation to *Phase Event Automata* (PEAs) and thereby verification using ARMC, an abstraction-refinement model-checker. Given an undesirable condition, ARMC can prove that the condition will not happen, or it can provide a counterexample trace showing how the condition may occur.

Syspect is a modeling application unifying Object-Z, Duration Calculus, and State Machines in a UML (Unified Modeling Language) context. In this work, we integrate ARMC model-checking into Syspect. A lot of information gets lost in the translation process from UML to *Transition Constraint Systems*, ARMC's input language. We devise mechanisms and annotations that enable us to nevertheless relate ARMC transitions to high-level specification elements.

Our extension smoothly integrates model-checking into the design work-flow, granting the developer a bird's eye view of how certain faulty states might be reached in a specification.

Zusammenfassung

Software ist allgegenwärtig in unserer Welt und findet auch verstärkt Einzug in sicherheitskritische Bereiche. Damit keine Leben gefährdet werden, ist es notwendig, die Korrektheit solcher Systeme zu gewährleisten. CSP-OZ-DC ist eine Kombination mehrerer Spezifikationstechniken für Prozesse, Daten und Realzeitverhalten, die eine Übersetzung in *Phasen-Event-Automaten* (PEAs) und von dort die Überprüfung durch den ARMC Abstraction-refinement-Model-checker erlaubt. ARMC kann entweder beweisen, daß ein unerwünschter Zustand in der Spezifikation nicht auftreten kann, oder es kann ein Gegenbeispiel herausfinden, das eine solche Situation hervorruft.

Syspect ist ein Modellierungswerkzeug, das Object-Z, Duration Calculus und Zustandsautomaten in einem UML-Kontext vereinigt. In dieser Arbeit integrieren wir das Model-checking mittels ARMC in Syspect. Bei der Übersetzung von UML zu *Transition-Constraint-Systemen*, der Eingabesprache von ARMC, geht viel Information verloren. Wir entwickeln Mechanismen und Annotationen, die uns dennoch erlauben, ARMC-Transitionen mit Elementen auf Spezifikationsebene in Verbindung zu bringen.

Unsere Erweiterung integriert das Model-checking elegant in den Entwicklungs-workflow und erlaubt dem Entwickler, sozusagen aus der Vogelperspektive zu beobachten, wie in einer Spezifikation bestimmte Fehlerzustände erreicht werden können.

Acknowledgments

I want to thank Ernst-Rüdiger Olderog for introducing me to the specification technologies CSP, Z, and DC in his lectures. Without that background, I probably would not have considered this thesis.

I thank Jochen Hoenicke for writing his dissertation and for developing the concept of Phase Event Automata and their implementation, and Andrey Rybalchenko for writing ARMC and for kindly answering my questions.

I thank everyone at the "Correct System Design" group at the University of Oldenburg, especially Ingo Brückner, for giving me helpful suggestions and proof-reading my thesis, Johannes Faber, for help with the PEA toolkit and insightful comments, Michael Möller for maintaining Syspect and for helpful discussions about the code base, and Roland Meyer, for clarifications about the PEA toolkit.

Thanks also to the students' project group who designed and implemented Syspect. Without that basis, this thesis would not exist.

Last but not least, thanks to Nils Müllner for proof-reading and comments, and thanks to my family and to my girlfriend, Diana Ippen, for supporting me through this whole time.

Contents

List of Figures

Chapter 1

Introduction

Decades ago, nobody would have dreamed of this, but today software is in use everywhere. From huge super computers to cellphones and handheld devices, from embedded devices in washing machines to networking equipment—everything is controlled by computer programs. Increasingly, micro-controllers and software are also being used in the medical domain, in power plants, in factories, even in vehicles, and in all these fields, failure is not an option, as one software fault could cost millions or even put lives at stake.

The software engineering discipline has undertaken great efforts to make software development tractable, efficient, and to control programmer errors. Yet many, like Frederick Brooks [Fre95], have observed that while modern programming technologies and safe programming languages help to avoid some mistakes, *software development is hard*. In other words, errors occur, and it is much harder to contain them compared to other engineering disciplines.

This called for a solution. If mistakes can not be avoided during design, systems have to be *verified* instead, using mathematical rigor. Specifications can be verified, either at compile-time or at run-time, but the various specification techniques differ significantly in their power.

Full verification of a system would amount to simulation. It is well known that this is not feasible—we can not predict the behavior of a program in advance (otherwise we could solve the halting problem!), so it can not be determined if a fault may occur *in general*.

Many specification languages can only operate on restricted parts of a specification, like only data or only the process model. Others try to cover multiple aspects of software specification, like CSP-OZ-DC [HO02], which we will explain in chapter 2. CSP-OZ-DC is amenable to translation into automata, particularly into *Phase Event Automata* (PEAs [Hoe06]) that permit model-checking [HM05] [MFR06] by ARMC, an Abstraction-refinement model-checker [RP07] that operates on *Transition Constraint Systems*.

CSP-OZ-DC has been integrated into a UML-Java tool chain, by translation of UML diagrams into CSP-OZ-DC [MORW06]. The Syspect modeling application (System Specification Tool)[1] supports this integration and is able to export its program data as CSP-OZ-DC, as LaTeX markup, or as a PEA (serialized in XML).

[1] `http://syspect.informatik.uni-oldenburg.de/index.html`

Unlike other model-checkers, ARMC abstracts from the program it is model-checking and therefore can go beyond reachability checking with bounded data types. Instead, given a specification with a defined error state, ARMC may find an abstract stable state and report that the program is correct. Alternatively, it may provide a *counterexample trace*, a list of transitions in the Transition Constraint System leading to the error state.

In this diploma thesis we implement ARMC exporting from Syspect in a way that preserves enough information of the original specification. By keeping this information, ARMC traces can be tracked back to their source in the specification and displayed accordingly in Syspect, unifying specification and model-checking in one tool.

Chapter 2 gives an overview of the CSP-OZ-DC specification language, which forms the basis for a translation into Phase Event Automata. Chapter 3 introduces the Syspect modeling tool and delineates how Syspect's UML models are converted into CSP-OZ-DC.

Chapter 4 describes ARMC and introduces test formulae, Phase Event Automata, and their translation into Transition Constraint Systems. Chapter 5 lays out the annotation and linkage process we developed to be able to relate ARMC counterexample traces back to their source in the specification.

Chapter 6 elucidates our implementation changes in Syspect and addresses the visualization of counterexample traces as well. Chapter 7 concludes and discusses directions for future research and development.

Chapter 2

CSP-OZ-DC

There are numerous specification languages and calculi in use today throughout academia and industry. Some of them are concerned with data modeling, like the Z and Object-Z languages. Others, like CSP (Communicating Sequential Processes [Hoa85]) or the π-calculus, model dynamic processes and their interactions. Again others specify the real-time behavior of processes, among them the Duration Calculus (DC).

CSP-OZ-DC combines all three approaches, so we can model a complex system from several sides at once. In chapter 4 we will explore how to model-check a system from all these viewing angles. The rest of this chapter gives a short introduction to the specification techniques used and to how they are combined.

2.1 CSP

CSP (Communicating Sequential Processes) is a specification language introduced by Hoare in 1978 and published in book form in 1985 [Hoa85]. It describes the parallel execution of sequential processes and their communication via event channels.

CSP allows us to specify a program's protocol, and it permits specifying the concurrent activities of different processes, as well as their communication. A *protocol* defines whether certain methods have to be called in a predefined order. For instance, a network connection might have to be initialized before we can call any communication operations on it.

Since CSP does not offer many useful constructs for the description of sequential processes (like traditional procedural programming languages do), it is mainly used for the specification of the concurrent and communicating parts of a program. In CSP-OZ-DC, it thus performs the role of constraining the process dimension of the described program, while OZ and DC are used to model the data and real-time dimension, respectively.

The basic unit in CSP is a *process*. Processes can perform certain operations, as we will see below, most of which involve *events*. Of course most sequential programs also perform lots of other activities, but for the purpose of modeling the behavior of a process in CSP, we are interested only in the communicating parts of a process.

An *event* is a literal identifier that can be communicated to another process. Since the sending and receiving of an event is always synchronous in CSP, this also yields an easy

way of *synchronization*: a process wanting to send data will block until another process explicitly synchronizes with it. Similarly, a receiving process will wait until some other process sends data.

A plain event does not carry any useful information. In fact, unlike some other process languages (for instance CCS or the π-calculus [Mil99]), CSP does not distinguish between sending and receiving events; an event is merely a communication a process can engage in. Aside from events, CSP defines *channels* which permit communicating data values. In CSP-OZ-DC we will see that a channel will communicate data values of a specific Z type.

Sending a value v over a channel c is written as $c!v$, which is equivalent to the simple communication of v over c, written $c.v$. Receiving is more complex, since we do not yet know which value will be communicated. Receiving a value v and using it is written as $c?v \rightarrow P(v)$ and defined as $\Box_{v \in T}\ c.v \rightarrow P(v)$ (where T denotes the type of data communicated over c), i.e. the receiving process will engage in an appropriate communication and run the process P with the "received", or communicated, value.

2.1.1 Syntax

A CSP process can be constructed from simple basic processes. Let a be an event, P and Q be processes. Figure 2.1 shows the structure of a CSP process P'.

$$\begin{array}{rcl} P' & ::= & STOP \mid SKIP \mid a \rightarrow Q \mid P \,;\, Q \mid P \parallel Q \mid P \,{}_A\|_B\, Q \\ & \mid & P \,|||\, Q \mid P \sqcap Q \mid P \,\Box\, Q \mid P \backslash A \mid X^{A} \end{array}$$

Figure 2.1: Structure of a CSP process

The process *STOP* models *divergence*, or deadlock. It can not take any transitions. No process can continue running in parallel with *STOP*, i.e. it will always deadlock the whole system.

The process *SKIP* terminates immediately, i.e., it will transition into the process Ω which will not do anything (in an actual implementation, the process would usually be deleted at this moment). A process running in parallel with *SKIP* can still run unobstructedly, as it will not notice the termination of its sibling.

That is to say, unlike the Unix operating system [LMKQ89], CSP does not let processes wait for other processes to terminate. *SKIP* communicates a special event $\checkmark$ on termination, but as $\checkmark$ is not a legal event identifier in CSP, there is no way to observe this behavior from another process.

A process $a \rightarrow P$ will communicate the event a, and then transition into process P. In this case, a is called the *prefix* of the process P. The communication will only take place synchronously, however, so P will not run until another process communicates a with this process.

The *sequential composition* of two processes P and Q creates a new process $P \,;\, Q$, that will behave like P until P terminates, and will then behave like the process Q. (We could

say that this is a way we can observe the termination of a process, as the termination of P will cause Q to run.)

Processes can run in parallel and synchronize over sets of events: $P \underset{A}{\parallel} Q$ can take all transitions that P or Q can take, except for events in the set A, on which P and Q have to synchronize. For every event not in A, both processes will continue running individually. A variant of this composition, $P {}_A\|_B Q$, functions likewise, but only events in $A \cap B$ will cause P and Q to synchronize. $P \,|||\, Q$, the *interleaving* operator, is another variation, A being the empty set in this case: P and Q will run parallel without any synchronization.

Sometimes a process can take more than one possible transition: the *internal choice* $P \sqcap Q$ will make a choice internally (perhaps in a non-deterministic way), to behave either like P or like Q. The *external choice* $P \,\Box\, Q$, however, will accept any event that either P or Q accepts, i.e. whatever event arrives decides whether the process should behave like P or like Q. Usually the external choice operator is used in cases, where both P and Q have the form $a \rightarrow R$, i.e., wait to synchronize on an event, but this is not required.

For instance the process

$$(a \rightarrow P) \underset{\{a\}}{\parallel} (b \rightarrow Q \,\Box\, a \rightarrow R)$$

might communicate the event a (as it is allowed by the parallel execution operator) and reduce to

$$P \underset{\{a\}}{\parallel} R.$$

Finally, to hide or encapsulate the events a process will accept, the *event masking operator* can be used: $P \backslash A$ will behave like P, but it will not synchronize on any event in the set A.

CSP also allows recursion, by splitting a definition into *process equations*. Any equation defines a process name and a process definition, following above syntax. The process name, shown as X in the above syntax, can then be used instead of that process' definition.

For instance, in the following example P can communicate the event a and will then continue behaving as P, i.e., process P will accept an infinite number of a events.

$$P := a \rightarrow P$$

2.1.2 Semantics

CSP has a simple (bottom-up) operational semantics. For every process, the semantics specify what transitions the process can take, thus transforming into another process.

STOP can not take any transitions at all. $P \underset{X}{\parallel} STOP$ also reduces to *STOP*, so the whole system will deadlock.

As already noted, *SKIP* takes a ✓ action and transforms into Ω. A prefix operation $a \rightarrow P$ reduces to P while communicating a.

$P\,;Q$ reduces to Q when P terminates using the event $\checkmark$. In that case the process will communication the internal event τ. Otherwise, if P does not terminate, but reduces to another process P' by communicating some event, $P;Q$ will reduce to $P';Q$ communicating that event.

If two parallel processes P and Q both communicate an event a that is part of their synchronization set A, while P evolves to P' and Q evolves to Q', their parallel composition evolves to $P' \underset{A}{\parallel} Q'$. The event a is propagated as well, so it is also visible outside the parallel composition.

If one process in a parallel composition reduces to another process (say, P to P'), communicating some event which is not in A, the parallel composition also propagates that event, while the other process remains unchanged. In the case of P the composition becomes $P' \underset{A}{\parallel} Q$.

We should mention the exception to the rule: termination. A termination event $\checkmark$ is not propagated. Instead, the parallel composition simply changes to $\Omega \underset{A}{\parallel} Q$, when P terminates.

The composition itself will terminate when both its sub-processes terminated. By terminating, the composition will itself generate a $\checkmark$ event.

The choice operators only choose one of their sub-processes: $P \sqcap Q$ can become either P or Q, generating a τ event. The external choice only makes this decision when an external event prompts it to. However, the semantics presented here are bottom-up, so this is not really visible. Instead we assume (for the sake of example) that there is an external process that will let us communicate an event α. In that case, if one process in an external choice (say, P) can communicate that event α (by changing into P'), the whole choice will reduce to the resulting process: $P \,\Box\, Q$ will become simply P', propagating the event α.

Since the sub-processes in an external choice might change without communicating any events (i.e., while communicating τ), the choice changes accordingly, communicating τ itself. We could say that internal reduction of sub-processes does not trigger the choice; only regular events do.

The event hiding operator $\setminus$ propagates any event not in its hiding set, but all other generated events will be blocked, changing into a τ. A $\checkmark$ event will terminate the hiding construct, also communicating $\checkmark$.

Probably the only case where a reduction does not really reduce, but expand, a process is the call of an identifier X, which is defined to be some process P. The call will expand to become the process P, communicating a τ.

For our purpose, this is all the CSP we need to know, so we do not discuss other topics about CSP, like CSP traces or failure-divergences semantics in this thesis. We refer the interested reader to [Hoa85].

2.2 Z

The *Z notation* [Spi92] is a specification language in industrial use. In the late 1970s it was proposed by Jean-Raymond Abrial and then further developed at Oxford University.

In the year 2002 Z became an ISO standard.

The purpose of Z is to describe data values and the effect of operations on that data. Unlike many commonly-used programming languages, Z does not build on the concept of nominative types, but specifies permissible values using a variant of Zermelo-Fränkel set theory (the name Z is also derived from Zermelo): it does not matter what a type is named, but what values it contains.

Z is strongly typed, i.e., every expression in a Z specification carries a type. Besides types being good documentation for the software developer, they also help avoid those paradoxes that might otherwise come up in naïve set theory. Z specifications consist of definitions following the *Z mathematical toolkit*, a library that defines commonly used operations on numbers, sets, relations, and functions.

The user can choose between using the built-in types (although Z only really provides numbers, denoted by the type $\mathbb{A}$) and defining her own. The mathematical toolkit provides operations like building the product of two types, as well as more complex constructions, such as relations or functions. In addition, the user can define so-called *free types*, which are the sum of several different types. For instance, an instance of the "tree" type can either be a leaf (of whatever structure), or a node which has other trees as children. This also shows that free types can be recursive, as well as *generic*: the type of a leaf node need not be specified if the tree is annotated as generic. The Z definition of such a type would look as follows; note the X as a generic type:

$$[X]\, Tree ::= Leaf\langle\langle X\rangle\rangle \mid Node\langle\langle Tree \times Tree\rangle\rangle$$

In addition to type and value constructions, the mathematical toolkit of Z also provides lots of predicates — logical predicates, set predicates — and operations on sets and numbers, relations and functions.

A Z program is typically structured as a number of *schemas*, i.e., "boxes" with an optional name consisting of a declaration and a predicate. A schema declares variables with their types and constraints on these declarations. For instance, a variable can be constrained to satisfy a certain predicate; values of a type can be constrained likewise.

Schemas can also modify state, in the form of variables included from other schemas, by declaring variables before the change (including *input parameters*), and after the change (including *output parameters*). The latter variables are *primed*, i.e., a becomes a'. The schema then defines a predicate relating the primed variables to the non-primed variables.

Because we do not want to redefine variables for each place of usage, it is convenient to *include* schemas. We can also include both a schema and the *primed* schema, i.e., the same declarations all suffixed with a prime. This is convenient for the above mentioned operations that modify state.

For instance, figure 2.2 defines a schema for a bank account that is constrained to only allow a positive amount of money on it, i.e., an implementation adhering to this Z specification may never allow the variable *balance* to be less or equal to zero. Figure 2.2 also shows an operation *Deposit*, which takes one *input parameter* and updates the *Account* schema's "balance" attribute. $\Delta Account$ indicates that *Deposit* uses and modifies the declarations of schema Account, in this case *balance* (but of course the constraint

Account

$balance : \mathbb{Z}$

$balance > 0$

Deposit

$\Delta Account$

$amount? : \mathbb{Z}$

$balance' = balance + amount?$

Figure 2.2: Simple bank account in Z

$balance > 0$ is also binding for *Deposit*, so we could not provide a negative number as $amount?$ that would reduce *balance* to below zero). $\Delta Account$ is equivalent to $Account \wedge Account'$.

2.3 Object-Z

Object-Z [Smi00] extends the Z language to include common features of object-oriented languages, like *classes* with subtyping or implementation *inheritance*. Object-Z definitions are structured at the class level, so a typical Object-Z specification uses class schemas instead of plain Z schemas. Classes define private data with accompanied *methods* operating on that data, which helps support data hiding and *encapsulation*.

Methods are functions operating within the class' context, i.e., from a Z point of view they can be said to implicitly include their class' schema. Input and output parameters are declared as in Z, with the schema predicate describing and constraining the primed output values.

Inheritance allows a class to extend an already defined class in an orderly fashion: an overridden method definition has to satisfy its super-class' constraints as well as its own additional ones.

Figure 2.3 shows the Z schema above transformed into Object-Z. Now the state and invariant take the form of a state schema within the class, and the *Deposit* schema has been translated into the operation "deposit," represented by the method `com_`*deposit*. Additionally, there is now an *init schema* that sets the beginning balance of any bank account to be 1.

Z and Object-Z can include all kinds of operators applicable to sets, numbers, and user-defined types, such as arithmetic, logical operations, set operations, or operations working on functions or relations. However, in this thesis, we only allow simple arithmetic and logical operations, because other types of operation are not yet expressible in the input language of the model-checker ARMC.

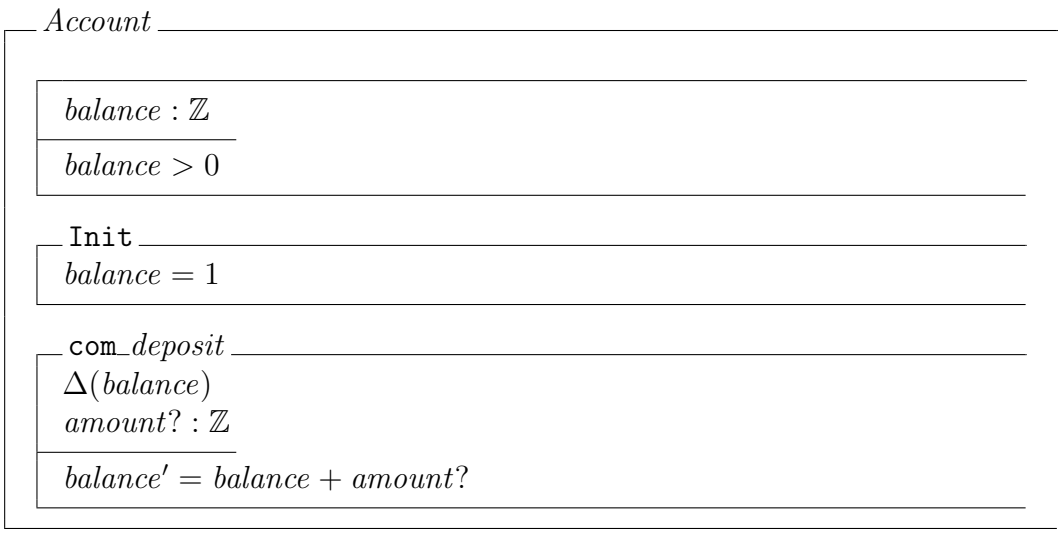

Figure 2.3: Simple bank account in Object-Z

2.4 Duration Calculus

Duration Calculus [ZH04] is an interval logic developed for the design and verification of real-time systems. Its syntax is divided into three layers: time-dependent state expressions, terms, and formulae.

2.4.1 Syntax

A *state expression* at a given point in time evaluates to either 0 or 1:

$$P ::= 0 \mid 1 \mid X = d_i \mid \neg P \mid (P_1 \wedge P_2)$$

State expressions can be zero, or one, can compare a so-called *observable* variable to a value, or can be constructed by using logical operations like negation, conjunction, or the derived operation disjunction. An observable is a mapping of continuous time to a value range. The *interpretation*, or semantics, of an observable is its value at a given point in time.

Terms build on state expressions and evaluate to numbers:

$$\theta ::= x \mid \ell \mid \int P \mid f(\theta_1, \ldots, \theta_n)$$

A Duration Calculus term can either be a number constant (or more precisely, a constant from a useful observable value range), the variable ℓ (which stands for the interval length), an integral of an observable function, or a function on terms. In order for integrals to exist, we must also constrain observable functions to be integrable. For this, the observable function is required to be of finite variability, i.e. the function may only be discontinuous at a finite number of points.

Formulae in Duration Calculus are constructed from terms; they evaluate to a boolean value:

$$F ::= \mathsf{true} \mid p(\theta_1, \ldots, \theta_n) \mid \neg\, F \mid (F_1 \wedge F_2) \mid \forall\, x \bullet F_1 \mid (F_1 \,;\, F_2)$$

The trivial formula is the predicate *true*. Alternatively, a formula can be an n-ary predicate of terms, it can be the logical combination of other formulae (using negation, conjunction, disjunction), or it can be a universally quantified formula or a chopped formula (using the **;** *chop operator*). A chopped formula is true, iff there is a point in the observed interval such that both sub-intervals are true for the respective parts of the chopped formula.

Conventionally, negation $\neg$ binds stronger than the **;** operator, which in turn binds stronger than the other logical operators: $(((\neg\, F) \wedge G) \Rightarrow H)$ can be abbreviated as $\neg\, F \wedge G \Rightarrow H$.

There also exist some abbreviations for Duration Calculus formulae:

point interval:	$\lceil\rceil$	$\stackrel{\text{def}}{=}$	$\ell = 0$
almost everywhere P:	$\lceil P \rceil$	$\stackrel{\text{def}}{=}$	$\int P = \ell \wedge \ell > 0$
F is true on a sub-interval:	$\Diamond F$	$\stackrel{\text{def}}{=}$	$\mathsf{true} \,;\, F \,;\, \mathsf{true}$
F is true on all sub-intervals:	$\Box F$	$\stackrel{\text{def}}{=}$	$\neg \Diamond \neg\, F$

2.4.2 Semantics

The semantics of Duration Calculus formulae, terms, and state expressions are defined inductively on their syntax. The environment of a Duration Calculus specification is provided by observables, i.e., state variables X of a type D dependent on time. Their *interpretation* $\mathcal{I}$ is given as follows:

$$\mathcal{I}(X) : \mathsf{Time} \to D$$

The interpretation of an observable is also denoted as $X_{\mathcal{I}}$. There are also global variables. Their value in a Duration Calculus program is given by a *valuation* $\mathcal{V}(x) \in \mathbb{R}$.

The semantics, or interpretation, of a state expression is given inductively by its structure:

$$\begin{aligned}
\mathcal{I}[\![0]\!](t) &= 0 \\
\mathcal{I}[\![1]\!](t) &= 1 \\
\mathcal{I}[\![X = d_i]\!](t) &= \begin{cases} 1, & \text{if } X_{\mathcal{I}}(t) = d_i \\ 0, & \text{otherwise} \end{cases} \\
\mathcal{I}[\![\neg\, P]\!](t) &= 1 - \mathcal{I}[\![P]\!](t) \\
\mathcal{I}[\![P_1 \wedge P_2]\!](t) &= \begin{cases} 1 & \text{if } \mathcal{I}[\![P_1]\!](t) = 1 \text{ and } \mathcal{I}[\![P_2]\!](t) = 1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Logical operators like $\vee$, $\Rightarrow$, or $\Leftrightarrow$ can be defined in terms of $\wedge$ and $\neg$. For $\mathcal{I}[\![P]\!]$ we also write $P_{\mathcal{I}}$.

The interpretation of a term in a given interval $[b, e]$ is calculated inductively as well, given a valuation $\mathcal{V}$ of variables:

$$\begin{aligned}
\mathcal{I}[\![x]\!](\mathcal{V}, [b, e]) &= \mathcal{V}(x) \\
\mathcal{I}[\![\ell]\!](\mathcal{V}, [b, e]) &= e - b \\
\mathcal{I}[\![\int P]\!](\mathcal{V}, [b, e]) &= \int_b^e P_{\mathcal{I}}(t)\,dt \\
\mathcal{I}[\![f(\theta_1, \ldots, \theta_n)]\!](\mathcal{V}, [b, e]) &= \hat{f}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e]))
\end{aligned}$$

In this context, $\hat{f}$ is the function (outside Duration Calculus) denoted by the symbol f.

Formulae work similarly, also being evaluated in the context of a variable valuation and an interval, only that they evaluate to boolean results (the true *result* is to be understood to be distinct from the Duration Calculus *formula* true).

$$\begin{aligned}
\mathcal{I}[\![\mathsf{true}]\!](\mathcal{V}, [b, e]) &= \mathsf{true} \\
\mathcal{I}[\![p(\theta_1, \ldots, \theta_n)]\!](\mathcal{V}, [b, e]) &= \hat{p}(\mathcal{I}[\![\theta_1]\!](\mathcal{V}, [b, e]), \ldots, \mathcal{I}[\![\theta_n]\!](\mathcal{V}, [b, e])) \\
\mathcal{I}[\![\neg\, F_1]\!](\mathcal{V}, [b, e]) &= \mathsf{true} \text{ iff } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, e]) = \mathsf{false} \\
\mathcal{I}[\![F_1 \wedge F_2]\!](\mathcal{V}, [b, e]) &= \mathsf{true} \text{ iff } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, e]) = \mathsf{true} \text{ and } \mathcal{I}[\![F_1]\!](\mathcal{V}, [b, e]) = \mathsf{true} \\
\mathcal{I}[\![\forall\, x \bullet F_1]\!](\mathcal{V}, [b, e]) &= \mathsf{true} \text{ iff for all } d \in \mathbb{R} : \mathcal{I}[\![F_1]\!](\mathcal{V}[x := d], [b, e]) = \mathsf{true} \\
\mathcal{I}[\![F_1 \,;\, F_2]\!](\mathcal{V}, [b, e]) &= \mathsf{true} \text{ iff there exists an } m \in [b, e] \text{ such that} \\
&\quad \mathcal{I}[\![\mathcal{F}_1]\!](\mathcal{V}, [b, m]) = \mathsf{true} \text{ and } \mathcal{I}[\![\mathcal{F}_1]\!](\mathcal{V}, [m, e]) = \mathsf{true}
\end{aligned}$$

Other logical operations are defined accordingly. $\hat{p}$ is the predicate (outside Duration Calculus) denoted by the symbol p in this context.

Since our main objective in this thesis is model-checking, we need to make sure that our programs are checkable. Unfortunately, some Duration Calculus formulae are too complex to allow easy verification. Therefore we have to restrict the structure of allowed formulae.

The main objective in using Duration Calculus to model real-time properties in software is to verify correctness, i.e., the absence of undesired program behavior. This is expressed in the notion of counterexamples. A *counterexample formula* always has a simple structure, following Hoenicke [Hoe06]:

$$\begin{array}{lll}
F & ::= & \neg\,(\mathit{phase}\,;(\mathit{phase} \mid \mathit{events})\,;\ldots;(\mathit{phase} \mid \mathit{events})\,;\mathsf{true}) \\
\mathit{phase} & ::= & (\mathsf{true} \mid \lceil \mathit{Predicate} \rceil)[\wedge\, \ell \sim t][\wedge \boxminus \mathit{Name} \ldots \wedge \boxminus \mathit{Name}] \\
\sim & ::= & \leq \mid < \mid > \mid \geq \\
\mathit{events} & ::= & \updownarrow \mathit{Name} \mid \not\updownarrow \mathit{Name} \mid \mathit{events} \vee \mathit{events} \mid \mathit{events} \wedge \mathit{events}
\end{array}$$

We also limit Duration Calculus formulae to such counterexamples, as they translate well into model-checkable programs.

2.5 CSP-OZ-DC

CSP-OZ-DC [HO02] is a combination of CSP, Object-Z, and Duration Calculus. It extends ordinary Object-Z with a number of features, namely *channels*, *process declarations*, and a number of Duration Calculus counterexample formulae. Methods are implemented as usual, but they are required to carry the name prefix "`com_`" and to have a corresponding channel declaration. For instance a method named *close* has to both have a channel *close* declared and to have an Object-Z method definition of `com_close`. Methods can also be split into their preconditions and effects. The *enable schema* describes the precondition, while the *effect schema* contains the method's effect or postcondition.

In order to be able to work with methods in Duration Calculus, we augment the DC language with a few operators. The basic idea is that a method call corresponds to an event, i.e., to an observable changing its value. If we model a method through an observable variable with the same name, for instance *close*, we can define certain properties of the event. $\nwarrow$ *close* holds on an interval $[b, e]$ iff the interval is of zero-length (i.e., if $b = e$) and if there is an $m < b$ such that $\lceil close \rceil$ on the interval $[m, e]$. Likewise, $\nearrow$ *close* holds for an observable iff there is an $m > e$ such that $\lceil close \rceil$ holds on the interval $[b, m]$.

The occurrence of an event, $\updownarrow$ *close*, can then be defined as $(\neg \nwarrow close \wedge \nearrow close) \vee (\nwarrow close \wedge \neg \nearrow close)$. If we want to specify the non-occurrence of an event, we use $\not\updownarrow$ *close*, equivalent to $\ell = 0 \wedge \neg \updownarrow close$ or a point interval where the observable does not change its value. If a certain event should not occur at all during a longer interval, we can use the operator $\boxminus$ *close*, which is defined as $\neg (\ell > 0 \,;\, \updownarrow close \,;\, \ell > 0)$. $\boxminus$ *close* is true for all point intervals.

Channels define the interface of a class, i.e., they restrict what operations or services are available publicly to other software components. A channel declaration corresponds to the availability of that channel in CSP.

Process declarations constrain the data flow between methods in the class. They consist of CSP declarations (in the form of Z terms) describing process equations. The CSP-OZ-DC class has to satisfy those process equations, starting with the process `main`. A communication in a process declaration corresponds to a method call on that channel. For instance, if a CSP process wants to communicate the event *close*, a call of the method `com_close` will trigger that event, advancing the CSP process. Effectively, the program can only take actions, when both the Object-Z invariants and preconditions are satisfied and the CSP-process can take the corresponding transition.

The Duration Calculus formulae constrain the class's possible behavior even further: as mentioned above, these formulae have to have the form of *counterexamples*, i.e. they express the absence of undesired real-time behavior. The CSP-OZ-DC class can only take an action when the Object-Z and CSP conditions are satisfied (see above paragraph), and when no Duration Calculus formula would evaluate to false (thus causing the unwanted counterexample to happen).

Thus, the program is effectively constrained to only that behavior that is legal in all three dimensions. Because all three facets — data constraints, process protocols, and real-time aspects — can be declared in a self-contained manner, it is very easy to ensure

correct program behavior this way (assuming that the specification is not faulty in itself, and does not allow undesired program behavior).

Chapter 3

Syspect

(System Specification Tool)[1] is an Eclipse RCP (rich client platform [ML06])[2] application developed as a students' group project at the University of Oldenburg. It allows the specification of software using a number of UML (Unified Modeling Language)[3] diagrams and annotations.

The Eclipse rich client platform is a set of Java libraries that tries to facilitate the construction of desktop applications by providing common building parts for such applications. Eclipse also provides its own module system that extends the Java package system: functionality is provide by *plugins*, that plug into the Eclipse platform, and can themselves provide *extension points*. By implementing an extension point (implementing an *extension*), another plugin can then provide functionality in a modular way.

Syspect is divided into several modular units—plugins. We extended Syspect to allow the export of projects into a model-checkable form (see the next chapter), and to integrate model-checking output into Syspect. We also adapted the existing Syspect modules, as necessary, to implement this functionality. This chapter describes Syspect's functionality[4] and how Syspect program specifications are translated into CSP-OZ-DC.

Syspect provides a *model explorer*, which yields a structured view of the whole application, as well as selected views for only parts of the application, to facilitate their editing. The editing in the various views takes place at the UML level. Figure 3.1 shows various *capsules* (a capsule is an *active object*, i.e. an object that runs in its own thread). The *Bank* capsule has two data attributes (*atms* and *resultQ*), several methods (from *auth* to *setup*) and a number of relations to other capsules, like dependencies to other components, the realization of interfaces, and composition information. The *Bank* capsule also contains a state machine, which is shown above the attributes and methods.

The model explorer also has a field *system definitions*, where the user can define data visible to all capsules in a project.

Besides the model explorer, Syspect provides a *diagram explorer* (see figure 3.2). The diagram explorer displays excerpts from the model, structured in terms of editable dia-

[1] See `http://csd.informatik.uni-oldenburg.de/~syspect/`

[2] Also see `http://wiki.eclipse.org/index.php/Rich_Client_Platform`

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

[4] for more detail, the reader is referred to the Syspect final report at `http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf`

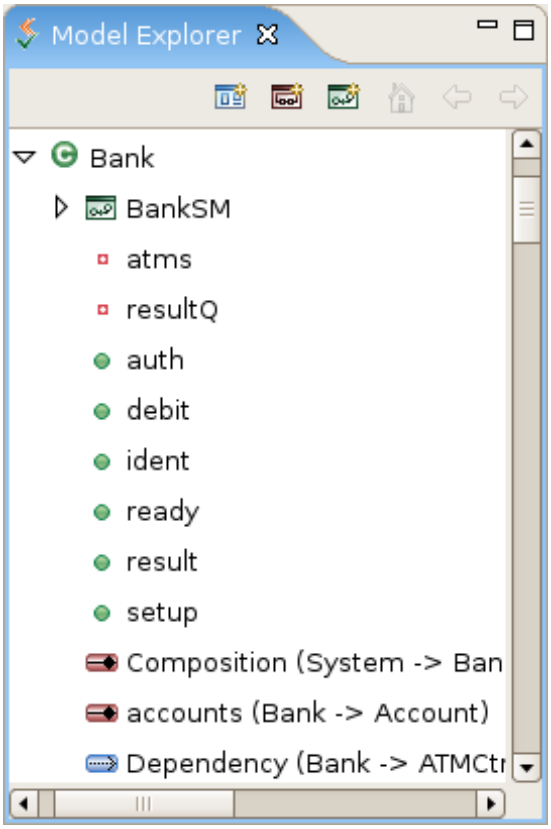

Figure 3.1: Model Explorer

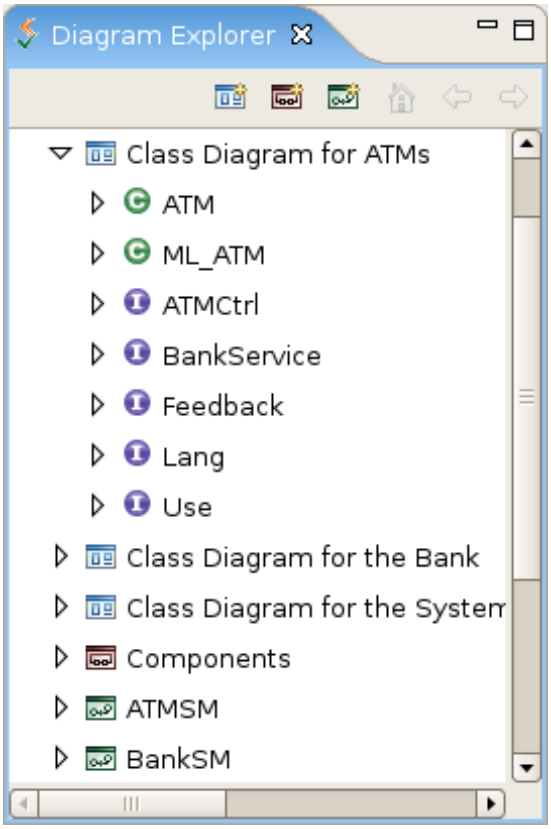

Figure 3.2: Diagram Explorer

grams like composition diagrams, class diagrams, and state machines. Is is possible that the user has decided to exclude certain capsules from a class diagram, but that they are nevertheless part of the application model.

The following sections will describe these diagrams in further detail, sketch their translation into CSP-OZ-DC, and discuss a few examples.

3.1 Class Diagrams

Class diagrams (see figure 3.3 for an example) list the attributes and methods of capsules and interfaces (in the example, the capsule *ATM* does not have any methods, only attributes; *ATM* implements two interfaces, but the methods are not shown is this particular diagram). Additionally, they allow to state dependencies (such as from *ATM* to *BankService* or *Feedback*), aggregation, generalization (such as *ML_ATM* to *ATM*) or realizations (such as the *ATM* capsule realizing the *ATMCtrl* and *Use* interfaces).

In this example, all capsules refer only to interfaces, so that it does not matter how the other components look inside. Likewise, the other class diagrams constituting the project only refer to *ATM* through its published interfaces. This way we could theoretically exchange component implementations (for instance to find out if one component works correctly where another does not).

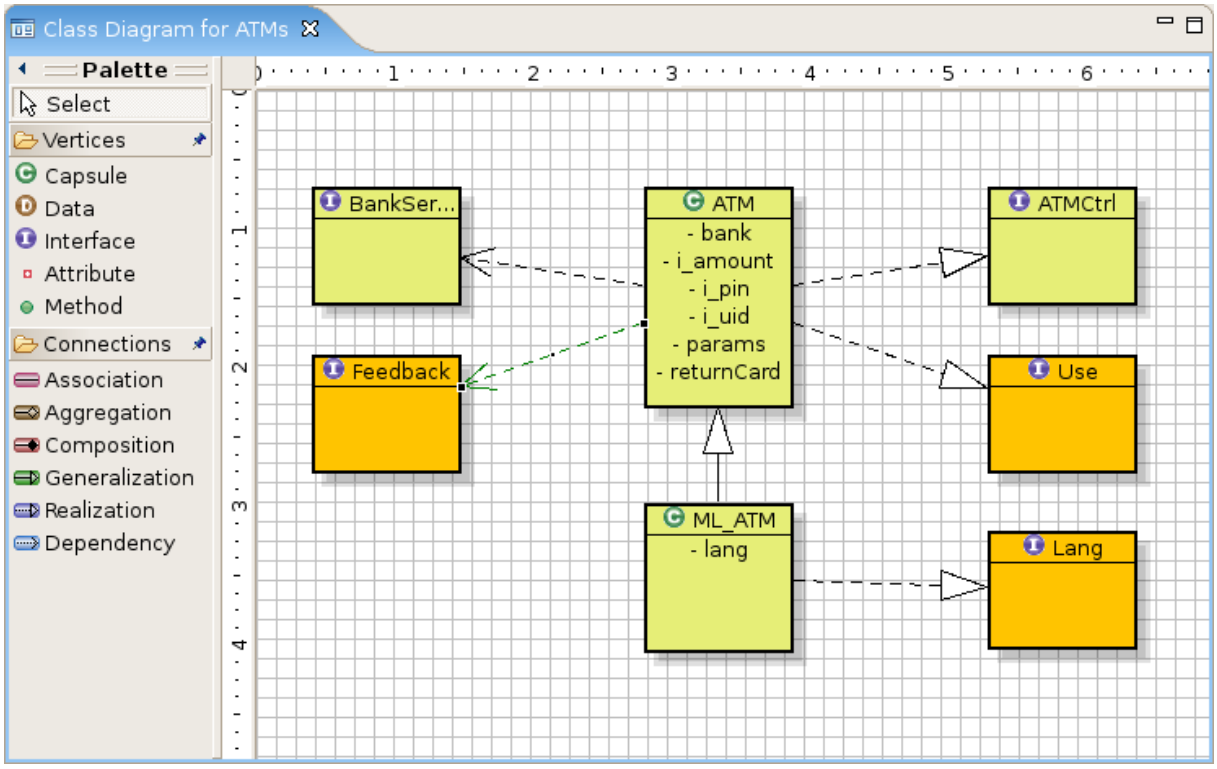

Figure 3.3: A class diagram in Syspect

A palette to the left allows the creation of more capsules or interfaces, as well as the creation of attributes or methods or the connection of these entities.

3.2 Component Diagrams

Component diagrams allow specifying modularity and information hiding at the level of individual components. Component diagrams model which capsules are used is implementations of which interfaces. In figure 3.4 we can see again how the *ATM* component provides the *Use* and *ATMCtrl* interfaces and depends on *BankService* and *Feedback* components. By arranging components and connecting their provides- and requires-interfaces, we can decide what component we want to implement a required interface. In this case, *Bank* provides the *BankService* interface, but we could also use another capsule that satisfies the same interface.

To the right of the diagram we see the external provisions and dependencies of the *System* component, which contains all the other components. *System* can be used by integrating it into even more complex components. If we want to model all possible behaviors of *System*, it is not even necessary to connect it to other services; we can leave these constraints open, thus leaving the details unspecified.

At the left, the component palette offers facilities for working with components and their connections. Individual components can also be created by dragging them from the model- or diagram explorer into the component diagram.

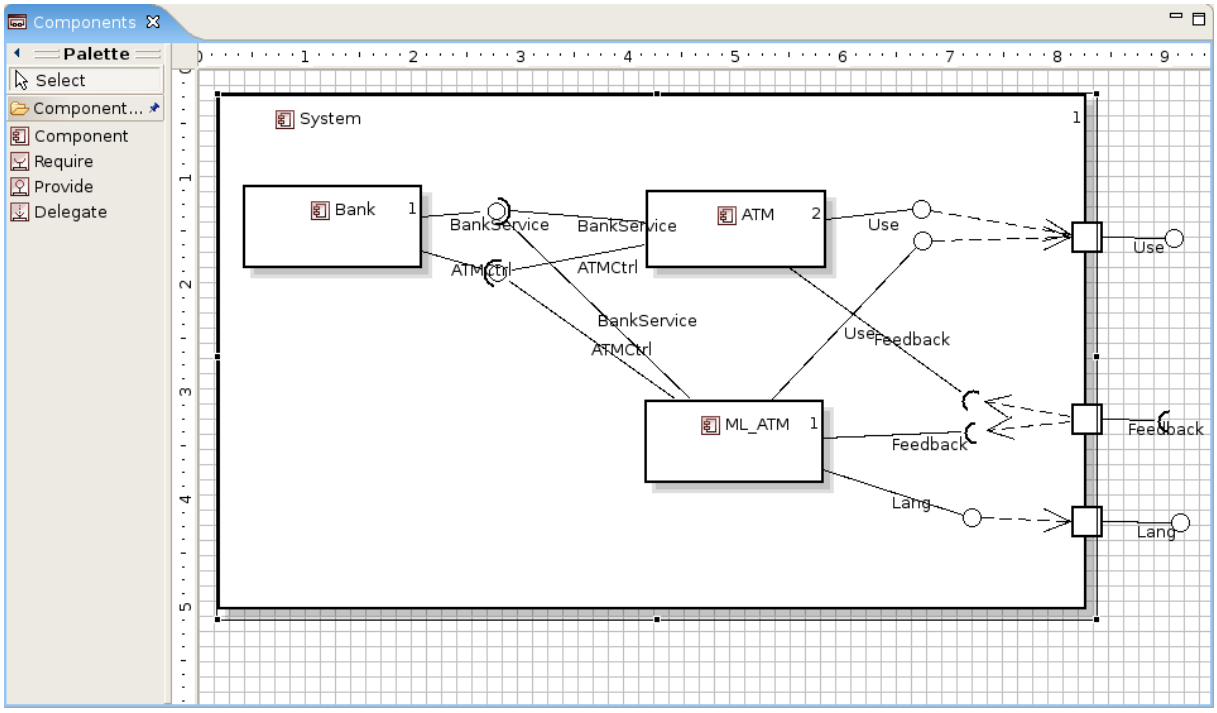

Figure 3.4: A component diagram in Syspect

While component structures are an interesting and important part of developing software, there is as yet no way to instantiate a component (or capsule) multiple times and soundly translate it into *Phase Event Automata* (see section 4.2), so we can not use component diagrams for model-checking. Instead, every capsule is instantiated exactly once, and all capsules are run in parallel. This is unfortunate and also implies that there should not be different capsules implementing the same interface (or when referring to an interface's method, there would be ambiguity which implementation of the method to call). We hope future research will enable a better translation of component setups.

3.3 State Machines

State machine diagrams help structure capsules such that their methods will only be called in a certain, predefined, order (i.e., the state machines express the class's *protocol*). They are also able to model concurrent execution of separate program threads. A *complex state* in a state machine can itself contain different *regions*, which are fully autonomous automata executed in parallel. A non-complex state is called a *simple state*. Only when all sub-automata (regions) reach their final state can the state machine leave the complex state containing the regions. We can think of regions as processes or threads, and of a regions' final states as join points for the concurrent activity.

Figure 3.5 depicts a state machine in Syspect with one initial state and two other states. The complex state *State* contains two regions that execute in parallel, each with their own initial and final states (or fork and join points). The first region will only accept a b event, while the second one will accept any number of as followed by a final b. Only

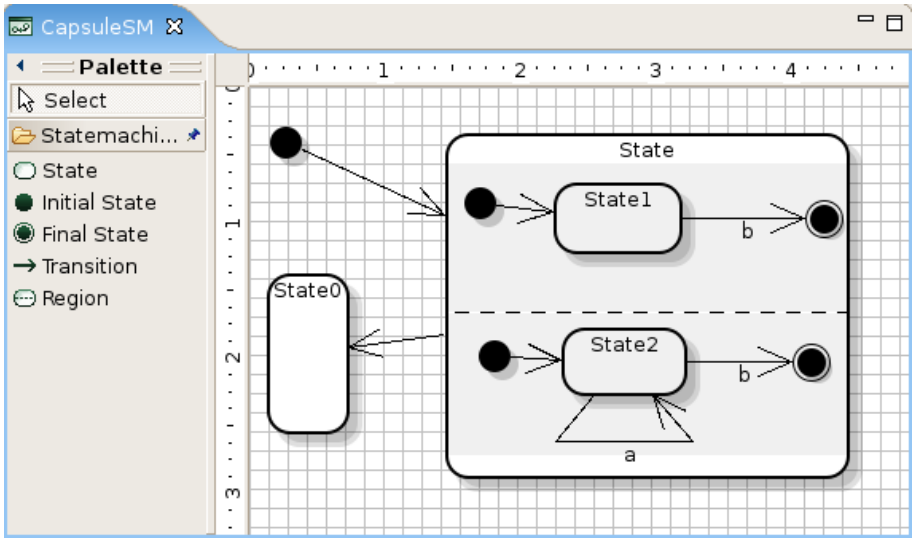

Figure 3.5: A simple state machine

when both regions are at their final state can the state machine take a transition from *State* to *State*0.

At the left of the state machine, there is a palette for creating new states, connecting them, and creating regions.

3.4 Object-Z and Duration Calculus

In addition to editing UML diagrams, Syspect also supports the editing of Z expressions and of Duration Calculus counterexample formulae, both of which are realized as capsule properties. Capsules can have a list of local variables, declared along with their Z type, an initial condition (realized as a Z predicate), and an invariant that needs to be satisfied at all times.

Methods have their own properties in Syspect, in particular Object-Z properties, which specify the usual Z method parameters: what variables change (the Δ set), the method's enabling condition, the method's effect, its input and output parameters, and locally used ("simple") variables. Figure 3.6 shows the Syspect Object-Z properties view for methods. The "edit" buttons invoke custom dialogs for editing the respective entries: for variable lists (changes, input, output, simple) there is a dialog for easily adding or removing variables, and for the enable and effect schemas there is a custom Z editor that allows the convenient insertion of special characters per button-click, or alternatively by typing their LaTeX markup.

Except for being a container to attribute variables and methods, a capsule can also carry Duration Calculus constraints and Test Formulae (the latter of which will be described in the next chapter).

The Duration Calculus part of Syspect allows the editing of counterexample formulae, i.e. undesired real-time conditions that should not happen during execution. A DC counterexample formula is associated with an individual capsule.

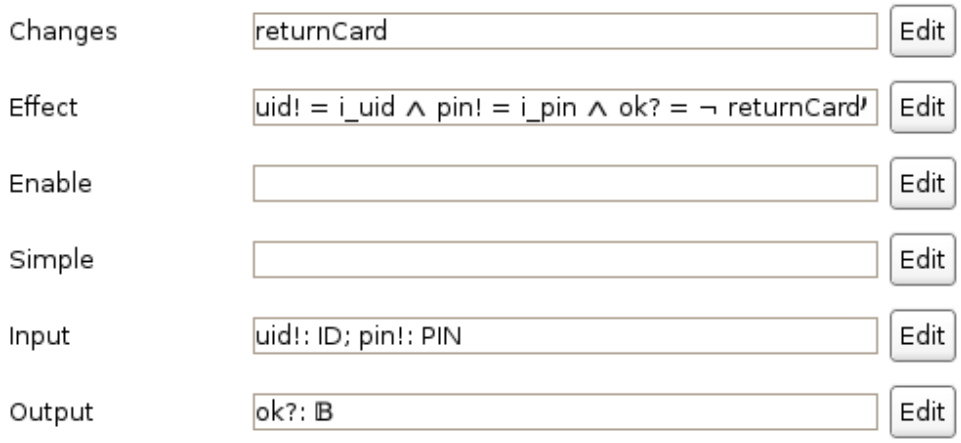

Figure 3.6: Method Object-Z properties

The models edited in Syspect can be exported using one of numerous functions: the various diagrams can be exported in Portable Document Format, or in several image formats; individual classes or whole projects can be exported as LaTeX markup, as a CSP-OZ-DC XML document, or as a PEA XML document.

In this thesis, we present an extension to Syspect that integrates the ARMC model checker and renders model-checking output in an intuitive fashion.

The following section will detail the translation of Syspect's UML models to CSP-OZ-DC.

3.5 Translation to CSP-OZ-DC

The translation process from Syspect's UML diagrams to CSP-OZ-DC is relatively straightforward: capsules are translated into active Object-Z classes. As mentioned above, component diagrams are not translated right now; instead every capsule is instantiated once. Capsules are active objects, so they conceptually run in parallel.

State machines are translated into CSP equations constraining the class accordingly. Duration calculus annotations are simply passed through the translation.

3.5.1 State Machines

The sequential and concurrent behavior of a state machine can easily be expressed in CSP, so the transformation is quite straightforward. Figure 3.7 sketches the translation with the function φ which maps state machines to CSP processes.[5]

(1): A final state in a state machine translates into the *SKIP* process. A state machine's initial state is translated into the CSP *main* process. All of its direct successors

[5]This figure is taken from the Syspect final report at `http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf`.

$$\varphi \equiv \begin{cases} P_s = SKIP & (1)\ \text{if } s \text{ is final} \\ P_s = \Box_{(e,t)\in\mathcal{T}_s} e \rightarrow P_t & (2)\ \text{if } s \text{ is a plain state and } \mathcal{C}_s = \varnothing \\ P_s = \bigsqcap_{t\in\mathcal{C}_s} P_t & (3)\ \text{if } s \text{ is a plain state and } \mathcal{C}_s \neq \varnothing \\ & \quad \text{or if } s \text{ is initial} \\ P_s = (\ |||_{i=1}^{n} P_{M_i})\mathbf{;} & (4)\ \text{if } s \text{ is a complex state} \\ \quad \mathbf{if}\, \mathcal{C}_s \neq \varnothing\, \mathbf{then}\, (\bigsqcap_{t\in\mathcal{C}_s} P_t) & \quad \text{containing regions } M_i, 1 \leq i \leq n. \\ \quad \mathbf{else}\, STOP & \end{cases}$$

Figure 3.7: Translation of state machines

are combined into an internal choice (i.e. the initial CSP process will choose any of them, non-deterministically).

(2) and (3): With $\mathcal{C}_s$ we will denote the direct successors of s, and with $\mathcal{T}_s$ the sets of all pairs (e, t), where t is a direct successor to s with triggering event e. A plain state with direct successors translates into the external choice of its successor processes $e \rightarrow P_t$. If there are successor processes that are not triggered by events, the internal choice is used instead, i.e. a successor process is chosen non-deterministically.

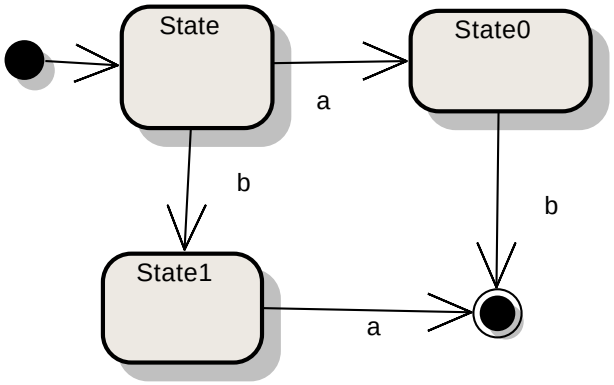

Figure 3.8: State machine with events

For instance the state machine in figure 3.8 translates into the following equation system, which is equivalent to $main = b \rightarrow a \rightarrow SKIP \,\Box\, a \rightarrow b \rightarrow SKIP$:

$$\begin{array}{lcl} State & \stackrel{c}{=} & ((b \rightarrow State1) \,\Box\, (a \rightarrow State0)) \\ State2 & \stackrel{c}{=} & SKIP \\ State1 & \stackrel{c}{=} & (a \rightarrow State2) \\ State0 & \stackrel{c}{=} & (b \rightarrow State2) \\ main & \stackrel{c}{=} & State \end{array}$$

A state machine's "events" correspond to methods in the Object-Z model, so the events translate into communications in the resulting CSP part. A communication in CSP is equivalent to a method call, as described in a previous chapter.

While CSP as described in the previous chapter allows communicating values over channels, we do not make use of this feature. Instead, method parameters are communicated in the Z part by setting the appropriate parameter variables. CSP merely specifies the order or concurrent behavior of method calls in a Syspect specification. In the end, the behavior is the same, as CSP, Object-Z, and DC are combined into a single Phase Event Automaton (see section 4.2).

(4): With $\mathcal{I}_M$ we denote the set of initial states in a region M. $\varphi(M)$, which is P_M, is defined as the internal choice of all initial states of the region $\sqcap_{t \in \mathcal{I}_M} P_t$ for all regions M.

A state containing parallel-executing regions translates into a parallel CSP process: every region is translated individually, and the containing state is transformed into the interleaved execution of all regions. Once all regions (or all interleaved CSP processes) have terminated, the state's successor(s) is/are called: a CSP sequential combination (;) appends either a *STOP* or an internal choice containing the possible successor states to the region-state.

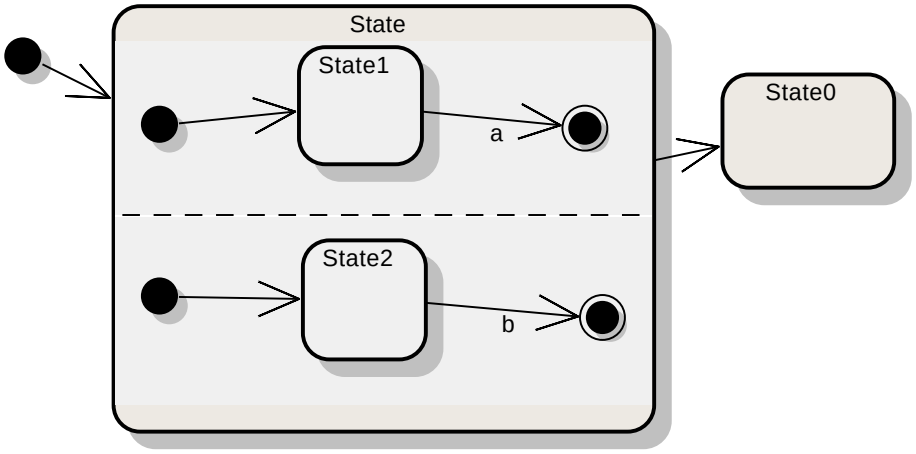

Figure 3.9: State machine with regions

The state machine in figure 3.9, for example, translates into a process equivalent to $(a \rightarrow SKIP \;|||\; b \rightarrow SKIP);\; State0$, where $State0$ is defined as $STOP$ (because it has no outgoing transitions and is not final):

$$
\begin{array}{lcl}
InitialState0 & \stackrel{c}{=} & State3 \\
State & \stackrel{c}{=} & ((InitialState0 \;|||\; InitialState1);\; State0) \\
State4 & \stackrel{c}{=} & (b \rightarrow State2) \\
InitialState1 & \stackrel{c}{=} & State4 \\
State3 & \stackrel{c}{=} & (a \rightarrow State1) \\
State2 & \stackrel{c}{=} & SKIP \\
State1 & \stackrel{c}{=} & SKIP \\
State0 & \stackrel{c}{=} & STOP \\
main & \stackrel{c}{=} & State
\end{array}
$$

A capsule can only have one associated state machine in Syspect, but a state machine is free to have several subsystems executing in parallel. In effect, the CSP-OZ-DC class

will only contain one CSP equation system, but this system is free to express any desired concurrent behavior.

3.5.2 Component Diagrams

As described above, multiple instances of Object-Z classes are not translatable into lower-level phases, so we are not concerned with component diagrams here. However, there exists a translation of component diagrams that further constrains the CSP part of the specification, as it specifies what capsule instances can interact with what other instances. Since we restrict ourselves to one instance of each class, this does not really have an impact.[6]

3.5.3 Class Diagrams

A class diagram consists mainly of interconnected capsules and interfaces. There are also *data classes*, which are not active like a capsule, i.e., they do not have their own thread and do not have methods. They only encapsulate data and are not used in our translation.

A capsule's attributes generate matching declarations in the Object-Z specification. The initial constraint, as well as the capsule invariant, translate into an init schema and an invariant predicate.

Every method in the capsule generates a corresponding *effect* schema. Since all predicates and invariants are already specified in Object-Z, this translation step is an easy one.

In addition to invariants and methods, a capsule can contain multiple Duration Calculus counterexample formulae, all of which will be copied verbatim into the CSP-OZ-DC class as real-time constraints. A capsule also holds test formulae, which will be described in the next chapter.

A given CSP-OZ-DC class only allows behavior that will respect both its data constraints, will be legal in the CSP equation system, and will honor all real-time constraints specified in DC.

3.6 Examples

We will now look at some simple examples that illustrate the translation—and later model-checking—process.

3.6.1 Counter

We will start with the easy example of a counter (see figure 3.10). The *Counter* capsule contains one attribute n of type $\mathbb{Z}$ and two methods *increment* and *decrement*. Figure

[6] For the translation see section 15.3.13 in the Syspect final report at `http://syspect.informatik.uni-oldenburg.de/doc/Endbericht.pdf`.

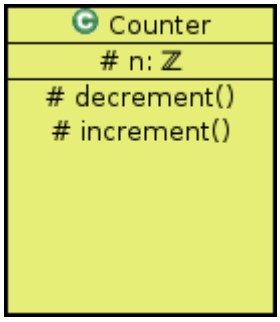

Figure 3.10: The *Counter* capsule

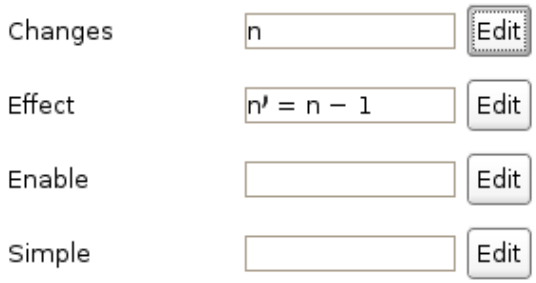

Figure 3.11: Properties for the *decrement* method

3.11 shows the properties for the *decrement* method: the attribute n is modified with the effect of $n' = n - 1$. There is no enable clause, i.e. the method is always applicable. Similarly, the *increment* method has an effect of $n' = n + 1$. The *Counter* capsule also has an init condition specified: $n = 0$. The capsule has no global invariant, so the methods are always applicable.

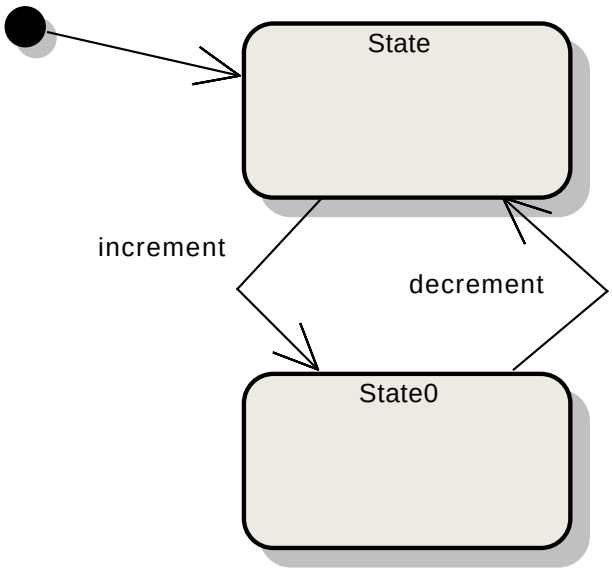

Figure 3.12: State machine for the *Counter* capsule

Figure 3.12 presents the state machine for the *Counter* capsule. We constrained the behavior of the counter to only allow a decrement operation after each increment operation. Accordingly, the counter's attribute n can only hold the values 0 or 1, which we will verify in section 6.4.

Figure 3.13 shows the *Counter* capsule exported as CSP-OZ-DC, using Syspect's LaTeX export. Note how the attribute declaration appears as a declaration as well. The initial condition translated to an init schema, and the two methods translated to method schemas with matching `local_chan` declarations.

The state machine translated into a CSP equation system with the start equation *main*. Since we did not provide a real-time constraint, there is no Duration Calculus

section in this example.

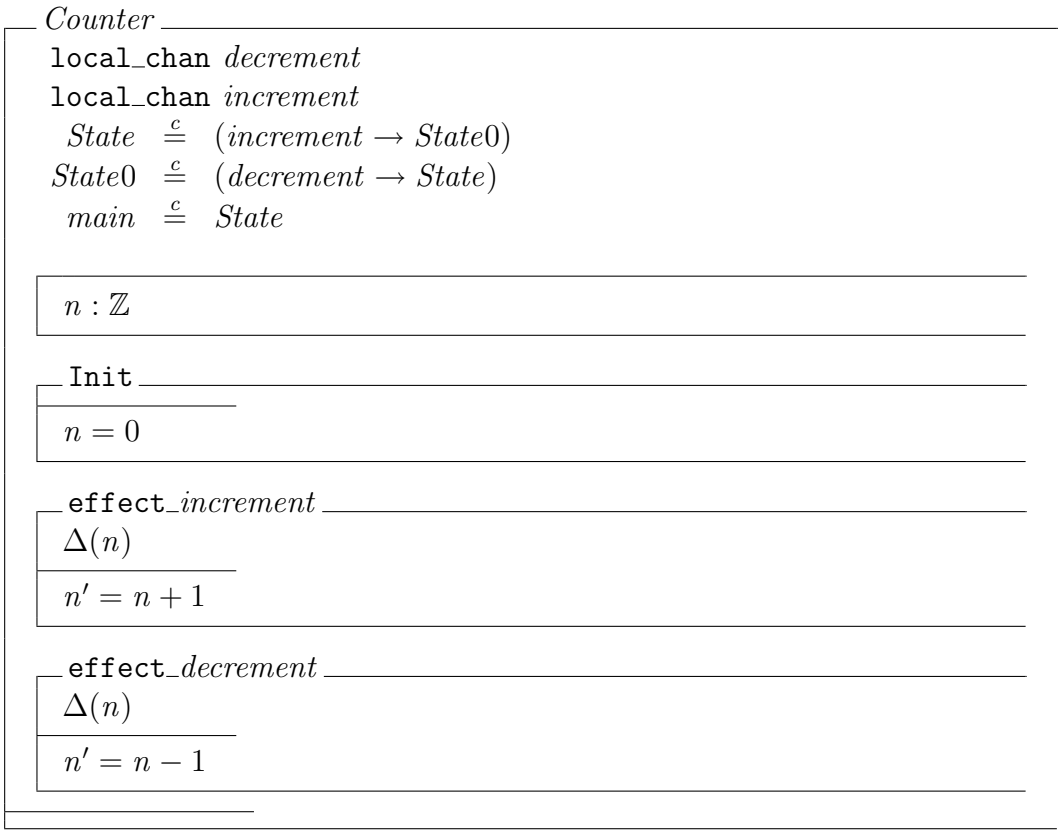

Figure 3.13: CSP-OZ-DC translation of the *Counter* capsule

3.6.2 CounterDC

Our next example is *CounterDC*, which is very similar to the *Counter* example. It has the same attributes (only n), the same two methods *increment* and *decrement*, and also initializes n to 0. However, *CounterDC* is not constrained by a state machine. Instead, we added a real-time constraint.

We defined the Duration Calculus formula *ThreeSecsPerInc* as $\neg \Diamond(\updownarrow \mathit{increment} \,;\, \ell \leq 3 \,;\, \updownarrow \mathit{increment})$, so there can be only one increment operation every three seconds (the number or frequency of decrement operation is not specified or constrained anywhere, so there can be between zero or infinitely many of them). As with the previous example, we will use this DC constraint to show some properties of *CounterDC* during model-checking.

Figure 3.14 summarizes the *CounterDC* capsule. There is no state machine, so there is no CSP specification part, either. Observe the Duration Calculus formula at the bottom.

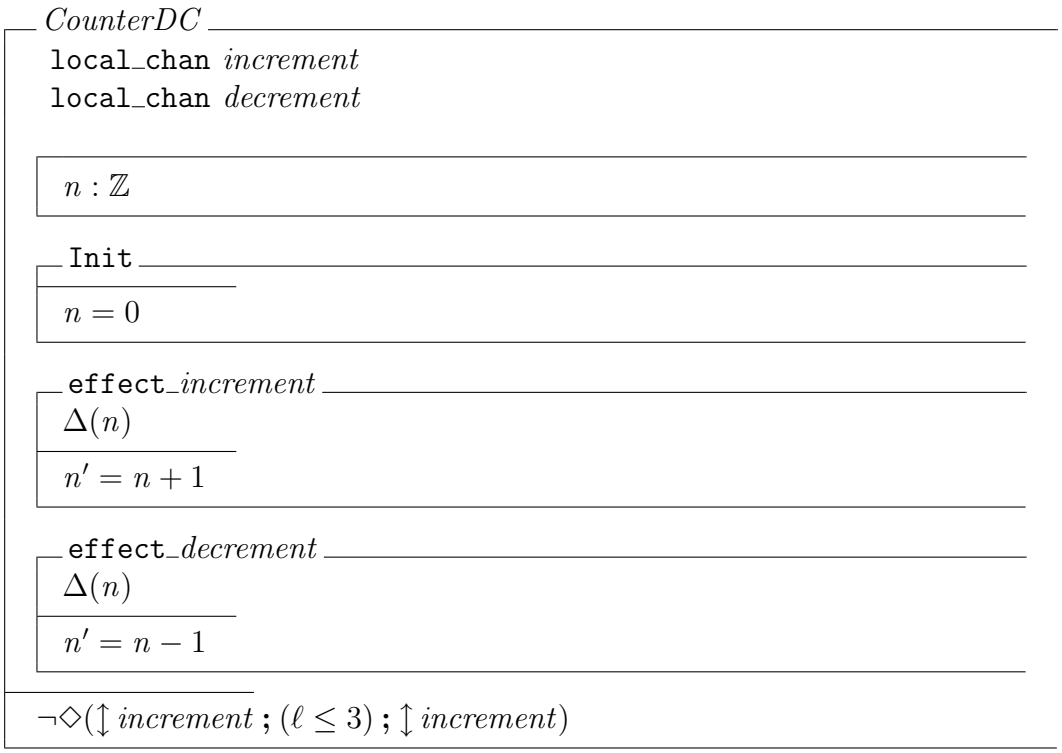

Figure 3.14: CSP-OZ-DC translation of the *CounterDC* capsule

Chapter 4

Model-checking with ARMC

Model-checking is the process of verifying if a formally defined *model* satisfies certain properties. The model can be a computer chip design that is being tested for proper behavior, or it can be a software program that is checked against a set of behavioral constraints. In our case it is a UML specification, and we want to find out if this specification allows *unsafe* states.

Before we describe what characterizes *unsafe* states, we have to distinguish between design properties and behavioral properties. The former are what is designed into the model: attributes we deem crucial to the correctness of the model, like limiting a vehicle's maximum speed. The latter are what follows from these specifications: properties whose correctness we want to verify. A behavioral property might or might not follow from the design specification. Model-checking allows us to find out if they do.

For instance we might stipulate that a vehicle be limited to some maximum speed, so that even in a sharp turn an acceptable centrifugal force will not be exceeded. The maximum speed is a model property, but the resulting centrifugal force is a behavioral limitation that follows from this design. That is, we would not design a car and designate that it can only have a specific centrifugal force; as we can not control this force directly, it would be useless to constrain the model accordingly. On the other hand, we have control over the vehicle's speed, so it makes sense to constrain this parameter. Effectively, a speed limitation is a constraint that is implementable (maybe by regulating the amount of fuel injected into the engine, or by braking if the speed limit is exceeded).

4.1 Test Formulae

While we will not have particular qualities laid out inside the model, we would like to ensure that they will be met nevertheless. In our extension to Syspect, we allow *test formulae* to be defined for each capsule in the model. This is analogous to specifying *unit-tests* in object-oriented software applications. The tests are not part of the program itself, but they permit checking characteristics of it. Test formulae are associated with capsules, so it is easy to define tests for each capsule individually, just like with unit testing, where developers write a test class for every program class that is to be tested.

For testing different characteristics of a capsule, there can be multiple test formulae, analogous to multiple test methods in a unit-test. In the model-checking process, we can decide which test formulae we want to check. The program is deemed correct if none of the selected test formulae is true.

Usually, only one test formula should be selected for checking, as otherwise we check their conjunction, not their disjunction. Effectively, the model-checker will only signal an error if *both* test formulae are violated. To test multiple test formulae, they should be tested individually.

Alternatively, a test formula might hold true, i.e. the model-checker may reach a bad state in its input program. In this case we would like to find out in which part of the program specification the error occurs, and what steps the model-checker took until it hit the bad state. This would help us correct the specification, perhaps by adding a few safety measures, or by correcting existing ones.

We should mention that there is a crucial difference between *testing* and model-checking, though. While a test can only prove the presence of certain problems, model-checking can prove their absence—assuming, of course, that the specification is correct and adequate. On the other hand, probing for lots of problems still takes lots of test formulae, just like with unit testing.

Syntactically, test formulae are a simplified form of Duration Calculus. Being not as expressive as unrestricted Duration Calculus, they permit translation to Phase Event Automata. Furthermore they need only express a few elementary conditions right now—those supported in ARMC—, so their syntax can be simpler than the syntax of Duration Calculus. A test formula consists of a sequence of conditions, namely *phases* and *events*. An event designates the occurrence of a CSP signal (or the calling of an Object-Z method). A phase consists of more elaborate specifiers: a phase can have a time bound, i.e. have a certain minimum or maximum duration. It can also have a list of events that may not occur during that phase. Additionally, the phase is constrained by a boolean formula consisting of boolean operations and arithmetic on the real numbers and on the capsule's variables.

$$
\begin{array}{rcl}
\mathit{Phase} & := & \ell > 0 \land \ell \sim k \mid \mathit{Ph} \land \lceil \varphi \rceil \mid \mathit{Ph} \land \boxminus \mathcal{E} \\
\mathit{Trace} & := & \mathit{Ph} \mid \updownarrow \mathcal{E} \mid \not\updownarrow \mathcal{E} \mid T_1 \,;\, T_2 \\
\mathit{BForm} & := & T \mid \neg\, F \mid F_1 \land F_2 \\
\mathit{Testform} & := & F \mid \mathit{TF}_1 \,;\, \mathit{TF}_2 \mid \mathit{TF}_1 \land \mathit{TF}_2 \mid \mathit{TF}_1 \lor \mathit{TF}_2
\end{array}
$$

where $\mathit{Ph} \in \mathit{Phase}$, $T, T_i \in \mathit{Trace}$, $F, F_i \in \mathit{BForm}$, $\mathit{TF}, \mathit{TF}_i \in \mathit{Testform}$, $k \in \mathbb{R}_{>0}$, φ is a state expression involving time, $\mathcal{E}$ a boolean observable, and $\sim \in \{\varnothing, \leq, <, >, \geq\}$, and a trace's first element is a phase.

Figure 4.1: Syntax of Test Formulae

Figure 4.1 lists the syntax as EBNF (as defined in [Mey05], p. 28). Obviously, the syntax has some ambiguities. For instance it is not clear if the test formula $\updownarrow e \,;\, \lceil a < b \rceil$ should be interpreted as a test formula containing a trace with two chopped phases, or rather as a test formula of two chopped test formulae that are simple phases.

In our implementation of test formulae in Syspect we eliminated the *BForms* for simplicity and instead allow a trace to be negated. A *BForm* that uses a negation around other constructs such as a conjunction can instead be expressed as two disjuncted negated test formulae, following de Morgan's laws. Additionally, since ambiguities only occur between traces and test formulae, we require all traces to be surrounded and delimited by parentheses. On the other hand, we recognize operator precedence, so a conjunction always binds closer than a disjunction. To avoid ambiguities, parentheses are not allowed in this case.

For example, the formula $(\updownarrow e)\,;(\lceil \mathit{true} \rceil\,;\mathit{true}) \wedge (\boxminus e)$ is legal. Note how all traces are clearly delimited, and how the first chop works at the test formula level, but the second one at the trace level.

Of course, having all these restrictions to the syntax is unfortunate but due to limited time for this task and to our using the JavaCC LL parser generator.[1] We used JavaCC mainly because other Syspect components already used it and because the focus of this thesis is on model-checking and we did not want to spend too much time on researching other tools. An improved future implementation might permit more liberal placement of parentheses by using an LALR parser generator. Where an LL parser has to take decisions based on only seeing the beginning of a term, an LR parser only needs to take decisions after all tokens and subterms have been successfully parsed.

In his diploma thesis [Mey05], Roland Meyer describes the representation of test formulae in XML, and their translation into PEA test automata. We extend this functionality by translating our test formula syntax into XML test formulae that can then be translated further by the existing implementation.

4.1.1 Examples

In the last chapter we developed some simple examples. This subsection will provide some test formulae for those examples. The *Counter* capsule had two operations, *increment* and *decrement*, and a state machine constraining the capsule, so that an incrementing operation can only take place after a decrementing operation.

We would try to test if the counter variable n can ever exceed 3, after an unspecific time interval: $\lceil \mathit{true} \rceil\,;\lceil n > 3 \rceil$. Since this is a single trace with two chopped phases, we have to surround it with parentheses, so Syspect will understand it: $(\lceil \mathit{true} \rceil\,;\lceil n > 3 \rceil)$.

The *CounterDC* capsule has the same two methods, as well as a Duration Calculus counterexample formula prescribing that two *increment* operations could only occur with an interval of more than three seconds between them. We would like to find out if the counter can change from 0 to greater than 2 in seven seconds, starting with an indefinite *true* phase. Again, since the formula is a trace, it has to be parenthesized: $(\lceil \mathit{true} \rceil\,;\lceil n \leq 0 \rceil\,;\ell \leq 7\,;\lceil n > 1 \rceil)$.

[1] `https://javacc.dev.java.net/`

4.2 Phase Event Automata

ARMC, which we will describe in the next section, is a model-checker that works on an operational program description. CSP-OZ-DC, on the other hand, describes a declarative specification of a program, so we have to translate the specification into an operational form to be able to model-check it. In his PhD thesis, Jochen Hoenicke develops *Phase Event Automata* (PEAs) for exactly this purpose [Hoe06]. He describes translations from CSP, Object-Z, and Duration Calculus into PEAs, as well as a translation from PEAs into ARMC's input language, transition constraint systems.

In this section we will describe Phase Event Automata, we will briefly sketch the translation of CSP-OZ-DC into PEAs, and we will illustrate the modifications necessary to the implementation in Syspect.

A Phase Event Automaton is an automaton incorporating concepts of concurrent, communicating systems, of real-time, and of data. The states of a PEA are called its *phases* and are annotated with data and clock invariants. PEAs can run concurrently, synchronizing their execution by communicating events (similar to the way CSP processes do [Hoa85]). A transition in the concurrent system can only be taken if every interested PEA in the system can take the transition. We call a PEA *interested* in an event if it uses the event in any context.

Definition 4.2.1 (Phase Event Automaton)**.** A *Phase Event Automaton* (PEA) is an 8-tuple $(P, V, A, C, E, s, I, E_0)$, where:

- P is the set of phases
- V is a set of variables; since ARMC only supports real-number arithmetic at the time we are writing this thesis, we require elements of V to denote real numbers
- A is a set of event variables (of boolean type)
- C is a finite set of clocks (where time is expressed as a real number)
- $E \subseteq P \times \mathcal{L}(V \cup A \cup C \cup V') \times \mathbb{P}(C) \times P$ is a set of edges
- $s : P \to \mathcal{L}(V \cup \mathbb{Z})$ is a function annotating states with state invariants
- $I : P \to \mathcal{L}^c(\mathsf{Clocks})$ is a function annotating states with clock invariants
- $E_0 \subseteq \mathcal{L}(V) \times P$ is a set of initial edges

A PEA can start in a state $(g, p) \in E_0$ if it satisfies the initial state invariant g and the state invariant of the initial phase: $s(p)$. All clocks in C are initialized to zero.

From there on, a PEA can take a transition over an edge (p, g, X, p') if the current valuation of clocks and variables satisfies the transition constraint g. When a transition is taken, the phase changes from p to p' and all clocks in X are reset to zero.

A PEA can not traverse several edges at once, because the semantics of CSP-OZ-DC prescribe that every transition takes a non-zero amount of time, i.e., a PEA has to remain in a phase for a non-zero duration before taking any further transitions.

A system consisting of multiple PEAs behaves like a single PEA that is the product of its constituting automata. The product automaton will take a transition of any sub-automaton, as long as all other sub-automata can take a related transition: if the edge to be taken is labeled by an event, this event has to be taken by all automata that reference that event. Thus all automata participating in the system are run synchronously.

For any event it might be the case that an automaton is not interested in it: it never communicates that event, nor its negation. To be able to run in parallel with other automata, it has to be able to take some transition. Therefore we require all PEAs to be *stuttering invariant*, i.e. have a stuttering edge in every phase.

A *stuttering edge* is an edge from a phase to itself with all events used in that automaton negated. So a Phase Event Automaton can take its stuttering edge if no event of interest to it is fired (otherwise it has to synchronize using that event). For instance an automaton P_1 that only uses events a and b will have a stuttering edge $\neg\, a \wedge \neg\, b$ for every phase. Should another PEA P_2 communicate c, it can synchronize with the stuttering edge of P_1, because the two do not conflict. On the other hand, another automaton P_3 that uses c somewhere could not use its stuttering edge to synchronize with P_2, because an edge labeled $\neg\, c \ldots$ would conflict with the event c. P_3 would have to transition using an edge that also communicates c, i.e. it has to synchronize on c with P_2.

Phase Event Automata can be built by hand,[2] but in Syspect they are generated from a CSP-OZ-DC specification, and thus from the original UML design. Every CSP-OZ-DC part of the definition translates to exactly one Phase Event Automaton. Later we will use this to our advantage, to associate parts of the UML definition with individual PEA edges.

4.2.1 Translating CSP

The relation between CSP equation systems and Phase Event Automata is rather straightforward: every CSP process has a corresponding PEA phase. Starting from the process represented by the *main* equation, a CSP process is converted to a PEA phase, and the same is done for all its successor processes. If a process already has a phase associated with it, that phase is used. Otherwise a new phase is created for that process.

Direct successors (reachable through τ transitions) transform into PEA edges labeled with the negation of all events that occur in the capsule (like stuttering edges), while prefix expressions ($a \rightarrow P$) translate into edges labeled with the corresponding event (a) and all other events negated.

There is no need for a special rule for parallel CSP compositions, as a parallel process behaves just like a sequential process to the outside—it merely has different ways of taking transitions.

Take figure 4.2, which depicts a capsule with a CSP process equation system. The process *main* is translated into an initial state for the PEA. As *main* can take a transition to *Init*, we create a new PEA phase for *init* (as it does not already have one) and create a PEA transition from *main* to *init* with events a and b negated (because the transition

[2] for instance using the Moby/PEA tool, available from the Moby website `http://csd.informatik.uni-oldenburg.de/~moby/`

Capsule

$$
\begin{array}{rcl}
\texttt{local\_chan}\ a & & \\
\texttt{local\_chan}\ b & & \\
\mathit{InitialState}0 & \stackrel{c}{=} & \mathit{Par}1 \\
\mathit{InitialState}1 & \stackrel{c}{=} & \mathit{Par}2 \\
\mathit{State}2 & \stackrel{c}{=} & \mathit{SKIP} \\
\mathit{State}1 & \stackrel{c}{=} & \mathit{SKIP} \\
\mathit{Init} & \stackrel{c}{=} & (a \rightarrow \mathit{Par}) \\
\mathit{Par} & \stackrel{c}{=} & ((\mathit{InitialState}1 \mid\mid\mid \mathit{InitialState}0)\ ;\ \mathit{Init}) \\
\mathit{Par}1 & \stackrel{c}{=} & (b \rightarrow \mathit{State}1) \\
\mathit{Par}2 & \stackrel{c}{=} & (b \rightarrow \mathit{State}2) \\
\mathit{main} & \stackrel{c}{=} & \mathit{Init}
\end{array}
$$

Figure 4.2: Example capsule with CSP process equation system

is labeled with τ). The PEA phase for *Init* can take a transition labeled by $a \wedge \neg\ b$ to another phase, which represents the process *Par*.

Par's phase is equivalent to $(\mathit{InitialState}1 \mid\mid\mid \mathit{InitialState}0)$;*Init*, so it can choose to take any transition that either *InitialState*1 or *InitialState*0 can take: $(\mathit{Par}2 \mid\mid\mid \mathit{InitialState}0)$; *Init* or $(\mathit{InitialState}1 \mid\mid\mid \mathit{Par}1)$; *Init*, and so on. Again, both edges are labeled with all events negated, as the transitions are τ transitions.

In the end, the phase representing $(\mathit{State}2 \mid\mid\mid \mathit{Par}1)$;*Init* can take a $b \wedge \neg\ a$ transition to *Init*, because *Par*1 can take a b transition to *State*1, which is defined as *SKIP*, so the parallel composition can terminate.

It is not relevant if the concrete implementation of the translation process will choose many small τ steps, or only larger steps, as this will not change the behavior of the translated CSP system—τ transitions will simply result in more "event-less" transitions in the combined Phase Event Automaton.

In addition to these directly translated edges, the CSP PEA is enhanced by stuttering edges, so that every phase has an edge to itself, labeled with all events negated.

Figure 4.3 shows the translation of figure 4.2 generated by Syspect. For the sake of readability, we left out the stuttering edges (labeled $\neg\ a \wedge \neg\ b$), but of course every phase in the PEA has one. Additionally, we only mention events that are not negated: an edge reading a should be understood to read $a \wedge \neg\ b$, and vice versa. The various phases in the graph represent states in the CSP model, reachable by subsequent reductions.

As we can see, Syspect does not unify equivalent processes, so in this contrived example there are many equivalent ones (i.e. processes that can accept the same events in the same order). Fortunately, most real-life specifications will not contain automata that fire the same event in multiple concurrent automata-regions, so the blowup will not be excessive. However, interleaving concurrent regions will still generate many states, as it has to use a product construction.

Figure 4.4 presents the translation of the *Counter* example's CSP part, which looks

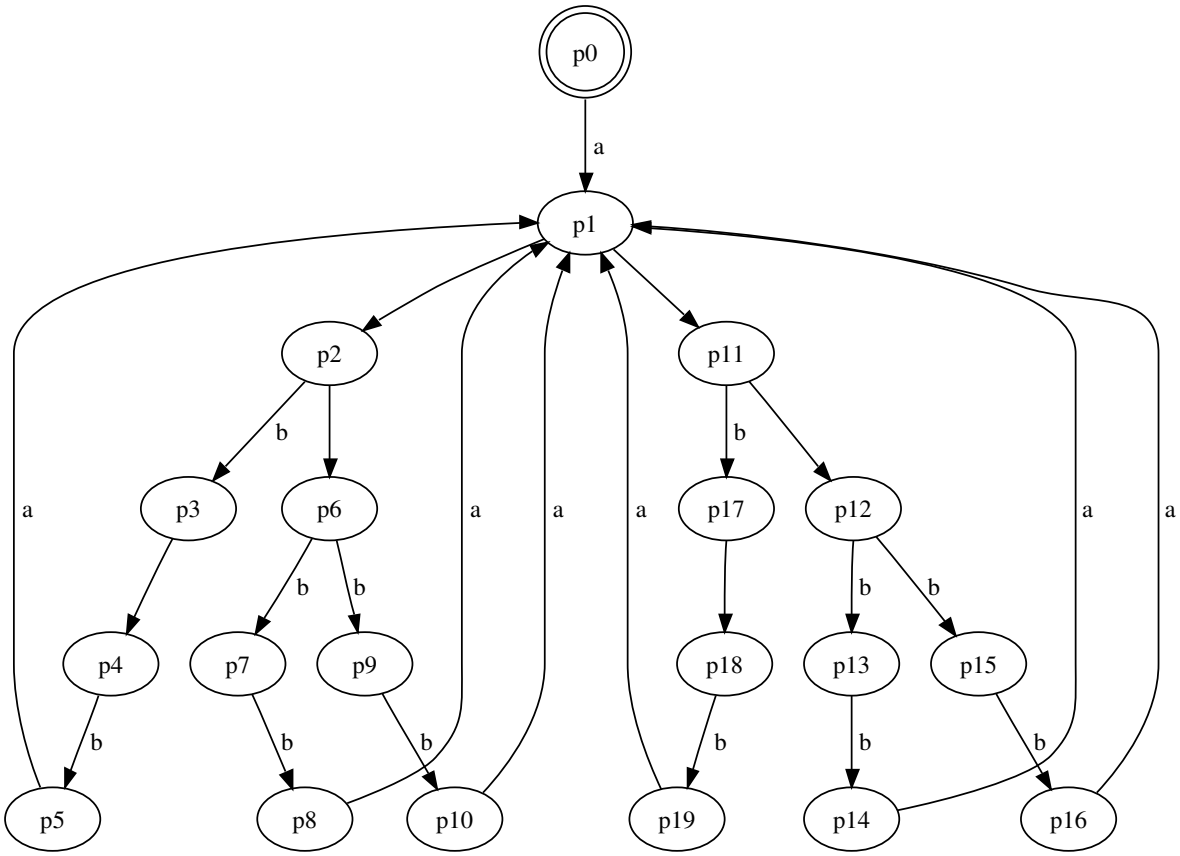

Figure 4.3: PEA Translation of figure 4.2

just like the original state machine defined in Syspect. The graph should be read like the one in figure 4.3: all stuttering edges and negated events are implied.

4.2.2 Translating Object-Z

Object-Z does not specify any sequential behavior, it merely constrains on what data a method can operate. For that reason we do not need many different phases in the Object-Z Phase Event Automaton, only two: the initial phase and the main phase.

In its initial state an Object-Z PEA can either stutter, i.e. take a transition labeled by the negation of all operations, or it can initialize, i.e. transition into the main phase by following the capsule's *init* schema. The invariant must also hold, so if it would conflict with the init schema, the PEA could not take any transitions (except for the stuttering one).

The main phase holds one stuttering edge, as well as one edge per Object-Z operation, as specified in the capsule. Each such operation edge is annotated with the conjunction of the capsule's invariant, the method's *enable* schema, and the method's *effect* schema. Because the semantics of Phase Event Automata do not specify what happens to the values of a variable if it is not explicitly assigned, we conjunct all edges in the PEA with $x' = x$, for each variable x that is not already being assigned in that edge (as x').

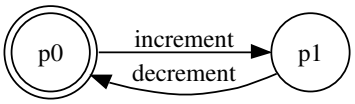

Figure 4.4: PEA Translation of *Counter*'s CSP part

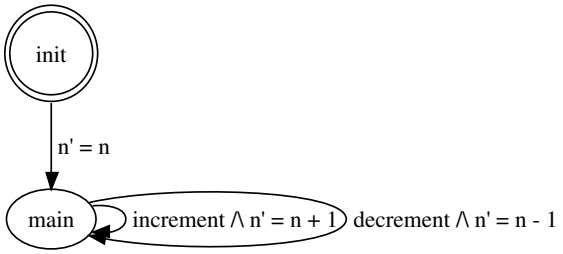

Figure 4.5: PEA Translation of *Counter*'s Object-Z part

Recall example 3.10. Figure 4.5 depicts the generated PEA. As usual, the stuttering edges ware left out for the sake of readability. *init* has a stuttering edge labeled $\neg$ *decrement* $\wedge$ $\neg$ *increment* $\wedge$ $n' = n$. The edge from *init* to *main* is labeled the same, as is the stuttering edge on *main*. The two operations in the PEA should be understood as negating all methods they do not use, i.e. the *increment* edge should read *increment* $\wedge$ $\neg$ *decrement* $\wedge$ $n' = n + 1$, so that only one method can occur at a given time.

4.2.3 Translating Duration Calculus

A Syspect specification can hold multiple Duration Calculus formulae that constrain the real-time behavior of the model. Each formula is translated into one Phase Event Automaton and later combined with the other generated PEAs.

Since a general Duration Calculus formula would not be implementable as a PEA, we only permit *counterexample* formulae. Following [Hoe06] we also prohibit the duration predicate $\ell = n$ for any number n, as the duration could not be measured by a PEA: if we build a non-deterministic PEA for a Duration Calculus formula, like for instance $\neg \Diamond(\updownarrow e ; \ell = 1 ; \updownarrow e)$, we may not be able to determine when the $\ell = 1$ phase ends. As there may occur an unbounded number of e events in an interval, we would need an unbounded number of clocks to measure the interval following *each* such e event, just in case that two e events have a duration of exactly 1 between them (the formula does not prohibit e events in the $\ell = 1$ interval, so *any* two e events having a duration of 1 between each other would violate the counterexample). By limiting the duration operators to upper and lower bounds, we eliminate this problem and are able to translate the DC counterexample to a Phase Event Automaton.

All counterexample formulae have the form $\neg$ (true ; ($\updownarrow e_1$ | $phase_1$) ; ... ; ($\updownarrow e_m$ | $phase_n$) ; true), i.e. they are negated chop-sequences of phases and events. While an event

occurs at a point interval, a phase always has a positive duration (except for true, which can be of zero duration). Hoenicke's algorithm operates on a sequence of extended phases, each of which is a phase with optional *entry events*, events that have to fire when that phase is entered. In treating Duration Calculus, we only consider phases (which we will number from zero) and associate events with their succeeding phases.

By saying that a Phase Event Automaton can be *in a phase*, or that it *satisfies* a DC trace, we mean that the automaton has reached the trace's last phase from a previous one. To be able to enter a phase, the PEA has to encounter all entry events, and the state invariant of the entered phase must hold. To stay in that phase, the automaton must not fire any prohibited events (as specified by $\boxminus e$) and has to respect both the phase's state invariant and its clock invariant.

For phases with upper time bounds, we need no further restrictions; as long as the time spent is beneath the threshold, the phase can be left, and the automaton transitions into the next phase (if the clock exceeds the upper bound, the automaton does not any longer accept the trace). However, for lower time bounds, like $\ell \geq 5$, the automaton has to *wait*. The current phase is still considered active, but not completely. As soon as the clock exceeds the lower bound, the phase is fully active.

Sometimes we do not want a clock to be lower or higher than a bound, but we also accept an exact match. In the Java implementation of the translation algorithm (class `Trace2PEACompiler` in the PEA toolkit), this is represented by the `exactbound` property (in class `PhaseBits`), while Hoenicke mentions it as two separate properties ("gteq" for lower bounds, and "less" for non-exact upper bounds, i.e. an exact upper bound would not have the "less" property).

In most cases it is not determined in what phase an automaton can be. Take for instance the countertrace formula $\neg \Diamond(\lceil A \rceil \,;\, \ell < 5)$. If at any point we encounter a phase where A holds, the countertrace would already be violated, as we could split that phase into two phases, the first of which would still uphold A, and the second of which would have a duration of less than 5. To implement a deterministic Phase Event Automaton, it is necessary to decide when to transition from one (PEA) phase to another.

Hoenicke's construction in [Hoe06] is similar to the construction used to transform non-deterministic finite automata into deterministic ones: instead of being in one DC phase, the PEA can be in several such phases at a time, as long as it satisfies all invariants and transition properties required to satisfy the whole DC trace. In many cases, it is undetermined if the automaton has to be in a specific phase, especially since the starting point of the countertrace is not usually specified (since its first phase is usually true). Instead the product constructions specifies that the PEA will be in multiple phases at once, and we can choose any starting point we want for the trace to begin.

For every phase in the countertrace with a time bound there is a unique clock variable to measure the duration of that phase. In phases with lower bounds, we have to conservatively assume that the automaton might have entered the phase at the earliest possible moment, i.e. the clock is reset only when the phase has to be left. In phases with upper time bounds, the clock is reset whenever the phase can be entered, even if it was already active. This keeps the clock value as small as possible, because we have to assume that the phase was entered at the latest possible moment.

The details of the construction are out of the scope of this thesis; the interested reader is referred to [Hoe06].

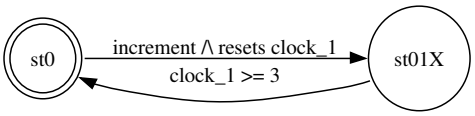

Figure 4.6: PEA Translation of *Counter*'s DC part

Figure 4.6 illustrates the translated PEA for *Counter*'s only Duration Calculus counterexample formula. All edges that do not mention the *increment* event should be understood to negate it, as usual, just as it should be understood that both states have implicit stuttering edges. While $st0$ has the usual stuttering edge, in this case the state $st01X$ has to respect its clock invariant, which is $clock_1 <= 3$ and thus has a stuttering edge labeled $\neg$ *increment* $\wedge$ $clock_1 < 3$ (smaller, because the invariant has to hold even after taking the stuttering edge).

Note that the clock refers to phase 1 of the countertrace, and that the second state is a product state of phase 0 and phase 1, where the X refers to phase 1's exact bound ($clock_1 \leq 3$ rather than $clock_1 < 3$).

4.2.4 Translating Test Formulae

In order to test if certain requirements hold in a PEA context, we have to go beyond Phase Event Automata. PEAs transition from phase to phase, but they have no way of deciding if a state is good, or if it represents an undesirable condition.

In his diploma thesis ([Mey05]) Roland Meyer develops the concept of *PEA Test Automata* (PTAs). PTAs are equivalent to PEAs, except for specifying a *bad state*. The underlying idea is that a PTA that goes through all phases of a test formula reaches the bad state. A model-checker could then verify if this bad state is indeed reachable, or if it is not.

When two PTAs are combined, the product's bad state is the product of both PTAs' bad states. The Java implementation in the PEA toolkit allows PTAs to have more than one bad state, which is not a problem, as these bad states could be merged to have a single bad state, to match the definition. The product of a PTA with a normal Phase Event Automaton is defined in the implementation as a PTA whose bad states are all product states of the original bad state, i.e., combining PEAs and PTAs can cause the number of bad states to increase, if all of them are reachable.

A Duration Calculus PEA can be understood as a test automaton without final states and without transitions leading to them. Thus the behavior that would violate the DC counterexample is prohibited. This also illustrates the different purposes of the two: DC counterexamples constrain the behavior of the model by disallowing certain transitions—typically in ways that could also be controlled in an actual implemented system. Test formulae, however, test, or measure, if certain error conditions can occur in an already specified system.

Just as PEA Test Automata extend Phase Event Automata, the PTA compiler extends the countertrace-to-PEA compiler. The PTA compiler takes as input one XML specification of a test formula, normalizes it, and generates test automata for all elements in the normal form.

Because the parallel composition of PEAs expresses conjunction, not disjunction, we can only take the first disjunct of a normalized test formula file to compose with the Syspect specification—luckily, most test formula specifications only carry one disjunct to begin with. If more than one test formula disjunct is desired, multiple test formulae should be used and be checked individually. Checking all formulae at once will not yield the desired results, since the effect will be checking if *all* test formulae (i.e. their conjunction) hold at once, not if one of them holds.

A disjunct may contain multiple conjuncted test automata, all of which we will use to run in parallel with the transformed Syspect specification. Each individual test automaton only represents a single test formula trace, and as such is structured like a Duration Calculus formula. This will allow us to treat TFAs rather similar to DC PEAs later on.

4.3 ARMC

ARMC [RP07] is an abstraction-refinement model-checker. By working on a set of relations, a *transition constraint system*, ARMC can try to determine if the program can reach certain user-defined error states (in our case). Generally, ARMC can be used to determine reachability in any context. By defining our error states as final states, we can use ARMC's reachability checking to find out if the error may actually occur.

We define *transition constraint systems* (TCS) in the context of ARMC. For a more general definition of transition constraint systems see [Hoe06]. Note that we only allow real numbers in variables (and only arithmetic operations), because so far ARMC does not support more advanced data types.

Definition 4.3.1 (Transition Constraint System)**.** A Transition Constraint System (TCS) is a 4-tuple $(st, v, init, tr, final)$, where:

- st is a finite set of states
- v is a finite set of variables representing real numbers
- $init$ is an initial state from the set st
- $final$ is a set of final states from st
- tr is a set of numbered transitions

A transition is a 7-tuple $(s, d, s', d', pre, post, n)$, where:

- s is a state from st
- s' is also a state (can be the same as s)

- d is a set of variables from the TCS's set v
- d' is a set of primed variables, which are from v. Usually d' contains the same variables as d, only primed
- *pre* is a set of preconditions (predicates over variables from v)
- *post* is a set of postconditions (predicates over variables from v)
- n is the transition number

A transition can *fire* when its precondition set *pre* (involving the set of variables d) and its postcondition set (depending on d and d') can be satisfied and the evaluator is in state s. This will transition the system into state s' with the variables changed from the non-primed values to their primed values. In effect, the pre- and postcondition sets are conjunctions over its elements. In particular, a transition can not express disjunction; instead, the disjunction is split into multiple transitions that reflect the individual conditions.

For instance, a transition $(a, \{x, y\}, b, \{x', y'\}, \{\}, \{x = x + y, y = 0\}, 5)$ will change the value of x to the sum of x and y and the value of y to zero. The TCS will transition from state a to state b in the process.

ARMC does not simulate these systems, though (which would be equivalent to simply executing a program), but abstracts the system's states. By looking at these abstract states and their transitions, ARMC either finds a *fixpoint*, a stable set of states the system will end up in, or it can yield a counterexample trace, i.e. a sequence of transitions that will link the system's abstract initial state to one of the abstract final states.

In some cases such a trace is only *spurious*, i.e. it occurs only in the abstract system, but does not lead to a final state in the original, concrete, system. In that case ARMC attempts to find another, more appropriate, abstraction and retries.

The interesting result to the user is either a message that the ARMC program is correct—i.e. that it can not reach an error state—or a message listing a series of transitions leading to an error state.

4.4 Translating Phase Event Automata into Transition Constraint Systems

Transition Constraint Systems have already been defined above (see definition 4.3.1). In order to translate a number of Phase Event Automata into a TCS, we have to apply some transformations to them. We have a parallel semantics for PEAs, but we have no such semantics for TCSs.

Like a PEA, a TCS has to choose from a set of transitions at any given time. Unlike a PEA, however, a TCS can not express disjunctions inside its transitions. So after calculating the parallel PEA, we have to compute the disjunctive normal form for every edge and generate the TCS's transitions from that. A PEA edge containing a disjunction

will then expand into more than one TCS transitions, each one containing one alternative from the disjunction.

All these steps, translating a list of Phase Event Automata to a Transition Constraint System, are performed using functions from the PEA Toolkit, which was developed at the University of Oldenburg.

Unfortunately, ARMC does not support inequality operations. Therefore we have to eliminate these from the PEA edges. Constructing the disjunctive normal form for a PEA edge might involve negating individual predicates, however (using de Morgan's laws). To avoid that an equality might be turned into an untranslatable inequality, we have to remove both $=$ and $\neq$ operations. $F = G$ translates to $F < G \vee F > G$, while $F \neq G$ translates to $\neg\ (F < G) \wedge (F \leq G)$. These operations are ARMC-specific, so they are only performed when a Phase Event Automaton is further processed for ARMC model-checking. The PEA export facility will not perform these expansions.

Of course a machine-generated ARMC program, computed from the normalized parallelized product automaton, can not be called human-readable, and even when presented with a series of transitions, the user will find it hard to detect which parts of the high-level specification caused the error. Therefore in the next chapter we will present ways to relate low-level transitions back to high-level definitions.

Chapter 5

Linking Counterexamples to High-Level Specifications

When we model-check a Syspect specification with respect to a test formula in ARMC, we receive either a correctness result or a *counterexample* trace leading to a faulty state. This trace only consists of a sequence of ARMC transitions, though, and the ARMC program does not look anything like the original specification. The Syspect specification was translated into individual Phase Event Automata, which were then combined into one parallel automaton and transformed into a Transition Constraint System.

Instead of having to look at the Transition Constraint System and trying to find out what effects the transitions in the counterexample trace had, it would be much more informative to see *which part* of the original specification is involved at all. A TCS transition will contain a lot of information that was generated during the translation process, such as clock variables, it will contain lots of assignments of the type $var' = var$, which are in fact expanded even further, to $\neg\ (var' < var) \wedge (var' \leq var)$, as explained in section 4.4.

Figure 5.1 illustrates the translation process from a UML specification to a transitions constraint system. Finding the interesting parts in a transition, and uncovering which specification part they belong to, is very hard in such a context. Our contribution in this thesis is a way to preserve as much information as possible, so that it will not get lost in translation.

With this information available, we will then reconstruct the specification behavior that led to the faulty state and link it to the original specification. The following sections will outline the reconstruction in reverse direction and introduce the mechanisms and annotations we employ to be able to restore the high-level information.

5.1 Linking ARMC Counterexamples to Phase Event Automata

As seen in figure 5.1, we build one Phase Event Automaton for the Object-Z part of the specification, one PEA for the (optional) state machine (CSP), and one PEA for each

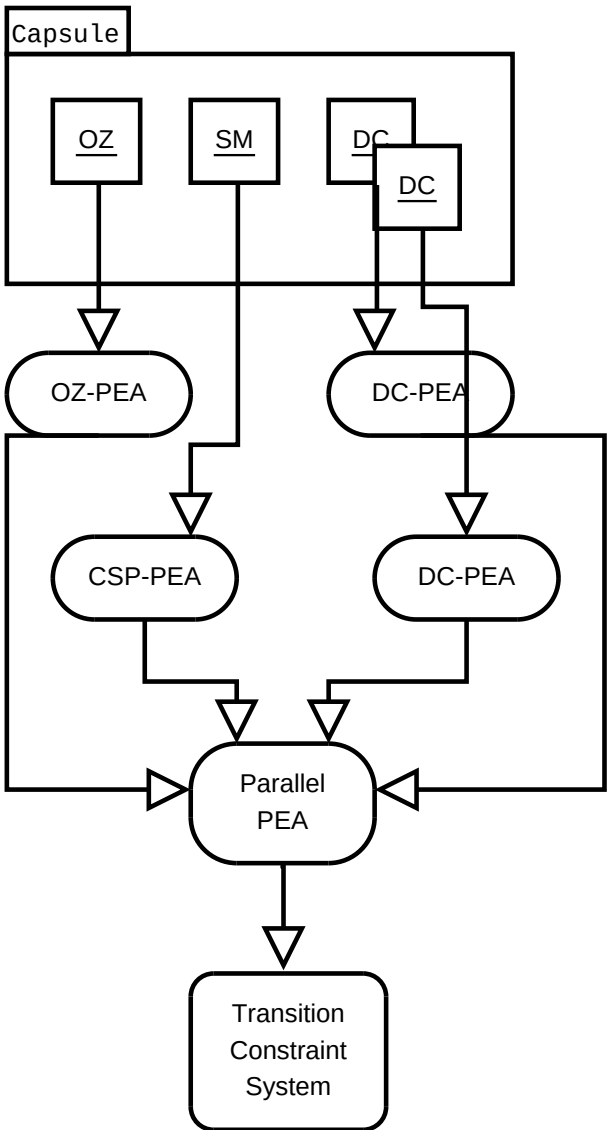

Figure 5.1: Translation from Syspect to TCS

Duration Calculus real-time restriction. To this we add one PEA Test Automaton for every test formula we want to verify.

All these PEAs are combined into one parallel Phase Event Automaton, which is then used to generate a Transition Constraint System. While we know how the automata and the TCS relate, we want to relate *individual parts* of these entities.

We can give each PEA edge a unique *edge identifier* (p, e), where p is the PEA number and e is the edge number within the PEA. As we are processing a list of PEAs, we can produce the first number by simple enumeration. The edge number is easy to obtain as well: we can iterate through the list of automata and number every PEA edge.

The product PEA has edges that are the products of the input PEAs' edges. Observing the edge identifiers, we can see that a product edge of n PEAs has the identifier product $\Pi_{i=1}^{n}(i, e_i)$.

A PEA edge may contain disjunctions, while an ARMC transition can only express conjunction. For that reason, PEA edges are translated into disjunctive normal form (DNF) and then split as appropriate. Each disjunction in a PEA edge will translate into one ARMC transition, i.e. one PEA edge may be associated with one or more ARMC transitions. We shall refer to this relation from PEA product edges to ARMC transitions as $tr \subseteq \mathit{edges} \times \mathit{transitions}$. Each PEA edge maps to unique ARMC transitions, i.e. for all edges e and f with $e \neq f$: $tr(e) \cap tr(f) = \varnothing$. Since all transitions in the TCS derive from PEA edges, the relation is surjective as well. Conversely, it follows that there exists a functional mapping from ARMC transitions to PEA edges in the product automaton.

It would be possible to follow through the whole translation process and take note which transitions are generated for which product PEA edge, but it would take a lot of time and be error-prone. It would also complexify any changes or refactorings of the PEA to ARMC converter in Java.

Instead we decided to analyze the generated Transition Constraint System and extract information from there. A Transition Constraint System is structured just like described in definition 4.3.1. Specifically, every transition is numbered, which is important since counterexample traces in ARMC's output contain transition numbers.

By the contents of an ARMC transition it is impossible to infer which PEA edges it was generated from. Two transitions may have the same source state, the same target state, they might even have the same effects and preconditions—just not all at the same time. Given a list of predicates like in a TCS postcondition we would have to guess which PEA each predicate was from.

Because clearly this would not be feasible, we annotate PEA edges with their unique identifier, as noted above. For every automaton we generate a variable—*_pea_trans_i_*, where i is the PEA number—and assign it the number of the current edge, for each edge in the automaton. Since there is no way to define variables privately, so they do not interfere with user-declared variables, we have to prohibit that the user names any variable *_pea_trans_i_* for any number $i \in \mathbb{N}_0$.

An edge in the product PEA built from edge 1 in PEA 1, edge 5 in PEA 2, and edge 19 in PEA 3 would be annotated with $\mathit{\_pea\_trans\_1\_}' = 1 \wedge \mathit{\_pea\_trans\_2\_}' = 5 \wedge \mathit{\_pea\_trans\_3\_}' = 19$, for instance. Such an annotation is simply conjuncted with the PEA edge's predicate. Since this annotation does not really have any effect on the program—for instance, an edge number is never part of a precondition, nor is it ever used in any other context than assignment—, it will not modify the possible behavior of a specification. Formally, if a PEA edge can fulfill its transition constraint g, which is not constrained in any way by any *_pea_trans_i_*, it can also fulfill $g \wedge \mathit{\_pea\_trans\_i\_}' = x$. Since the following transition will not be constrained by any *_pea_trans_i_* either, the assignment will have no effect on the automaton's behavior.

Effectively, we now have a mapping from an ARMC transition number to a PEA edge identifier, from which we can infer the actual PEA edge, in each source PEA. This means we can translate the ARMC counterexample trace back to a series of parallel PEA transitions. The second step in the translation involves relating these PEA edges back to the original Syspect specification.

5.2 Linking Phase Event Automata to High-Level Specifications

Given a set of PEA edges, we now want to obtain high-level information about which part of the specification an edge belongs to, and what high-level event caused the PEA edge to fire.

Taking a glance at figure 5.1 again, we see that every PEA belongs to exactly one part of the specification. We also know that we have exactly n PEA edges, where n is the number of PEAs, and we know which part of the specification each PEA belongs to.

Now at one transition it might be the case that most Phase Event Automata are simply taking stuttering edges, while only one or two automata engage in a meaningful activity. We need to find out which PEAs engage in useful activity, and what those activities are.

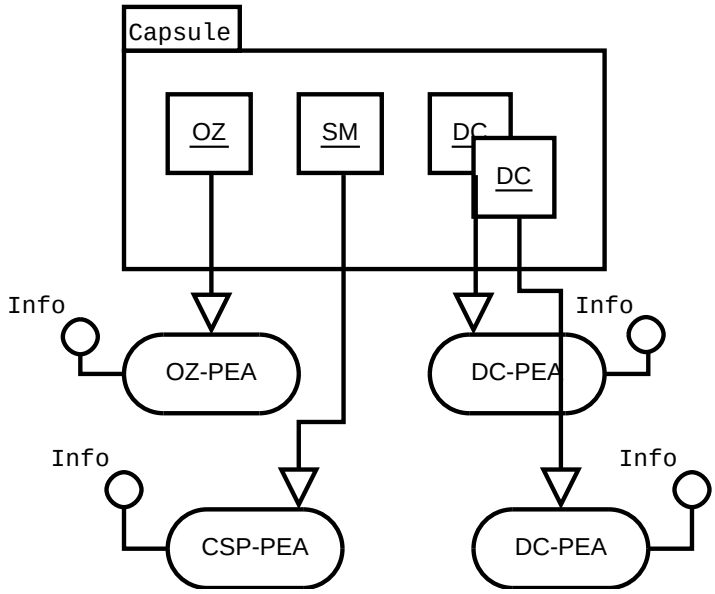

Figure 5.2: Translation of a capsule into Phase Event Automata

Figure 5.2 illustrates the translation of a Syspect capsule into Phase Event Automata. With every generated PEA we associate a `PEAInfo` object that will hold information about what PEA edges relate to which high-level entities in the specification.

The `PEAInfo` object has to encapsulate this information, because the actual converter objects will only live for the duration of the translation. The `PEAInfo` objects, however, are needed *after* model-checking, when we know the transitions in the ARMC counterexample trace—if there is one.

A `PEAInfo` object provides two methods: `getName()` and `getInfoForTransition(t)`, both of which return a textual string. `getName()` returns the name of the `PEAInfo` object, usually the type and name of the high-level specification element, such as "OZ: Capsule" or "SM: Train", where *OZ* would refer to an Object-Z specification and *SM* to a state machine. Other types are *DC* for Duration Calculus elements and *TF* for test formulae. `getName()` is informative, because just by looking at a generic `PEAInfo` object the user

(or a Java program) might not know what kind of `PEAInfo` object it is, a State Machine one, a Duration Calculus one, or something else.

`getInfoForTransition(t)` takes a PEA edge as a parameter and returns information about the part of the high-level specification that is related to the PEA edge. Often this method will return only an empty string, such as when the edge is a stuttering edge. Yet if any actual high-level activity occurred, it is returned in a form understandable by the user. The contents or even the kind of information returned by `getInfoForTransition()` is PEA-dependent and will be described below.

In principle `PEAInfo` objects are not bound by any structure, as long as they implement both methods. That is to say, there could be multiple `PEAInfo`s for each kind of Phase Event Automaton, if the user is interested in different kinds of information about it. However, we think that our provided `PEAInfo` implementations are a good first step in illustrating ARMC's model-checking results.

The following subsections describe what information the respective `PEAInfo` objects contain and how they are constructed during the translation process.

5.2.1 Linking Object-Z to Phase Event Automata

In section 4.2.2 we have seen that the Object-Z part of a capsule translates into a very simple Phase Event Automaton. The only actions it really takes are initializing, i.e. executing the *init* schema, and performing individual methods.

With such a simple automaton, the most useful information we can present to the user is the name of the operation (or "initialize"), and the operation's *enable* schema and *effect* schema, such as for instance "initialize PEA: $n = 0$" or "adjust: $speed' = speed - 2$."

The only information the `ObjectZPEAInfo` class needs is a map from PEA edges to Object-Z operations, which we generate during the Object-Z translation. For each method in the Object-Z definition we make an entry in the map linking the method's PEA edge with the operation information. For the initialization, which is not represented by a Syspect operation object, but by a field in the CSP-OZ-DC specification object, we generate a pseudo-operation that contains the *init* schema's effect. This uniform representation allows us to easily look up operations for each PEA edge in the `PEAInfo` object.

5.2.2 Linking State Machines to Phase Event Automata

Unlike Object-Z declarations, state machines are not quite so simply translated. A state machine is first converted into a CSP representation, an equation system, and then further processed into a Phase Event Automaton. Furthermore, the three layers—state machines, CSP processes, and Phase Event Automata—have differing semantics in regard to parallelism.

Phase Event Automata are not able to express any parallelism themselves. Although multiple PEAs can be composed to run in parallel, this merely makes use of a product construction: instead of having the PEA express parallelism, its state set has to embody all possible parallel behavior.

State Machines in Syspect support simple fork/join parallelism (similar to the Unix model of `fork()` and `wait()`), but do not allow arbitrary recursion: all states—or regions—have to be explicit; there are no placeholders. This contrasts with CSP, which features both concurrent processes and recursion by the use of process variables.

Fortunately, our CSP processes are constrained by the specifications expressed as State Machines, i.e. our goal was not to infer the structure of a State Machine from just any CSP process, but to link a State Machine to CSP processes that it generated. Because of the very structured fork/join concurrency present in State Machines, there are only certain transition patterns that occur in CSP, which makes the linking process manageable.

As described in section 4.2.1, every CSP process translates into exactly one PEA phase, which we take advantage of in our `PEAInfo` implementation: a Java map object registers all these translations, so that we can easily infer the CSP process for each PEA phases.

Unlike with the Object-Z part of the specification, we are not interested in what operation a transition performs (which in the CSP case would be just an event), but rather which process transitions into which other process. Since every PEA edge (which we can infer for every ARMC transition) has both a target and source phase, we can look up both CSP processes. By registering the CSP event for every transition as well, we can also infer it later for each PEA edge. In the CSP-to-PEA translator in Syspect we therefore integrated three map objects for a PEA transition's CSP source, CSP target, and event, respectively. Figure 5.3 illustrates these mappings from a transition to a source process, a target process, and an event.

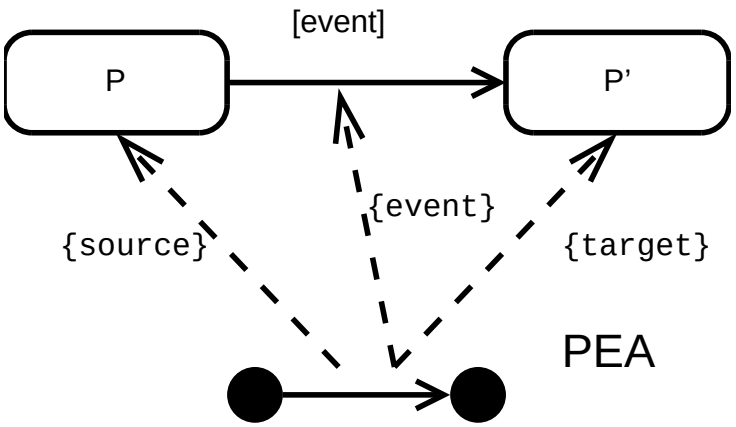

Figure 5.3: Mapping PEA transitions to CSP processes

As with the Object-Z translation, we do not register stuttering edges, so when we try to look up a stuttering PEA edge later, we will not find any CSP processes for it. For a stuttering edge there simply is no meaningful high-level information, because there is no equivalent action in the specification, so we do not need to return anything.

The harder problem is relating CSP processes to State Machine configurations, as a State Machine can be in more than one state at a time. If it is inside a *complex state*, it is both in that state and in one substate for each region inside it. So in a state with two regions, the State Machine could be in three states (if it is inside the complex state).

For this reason we introduce the *State Machine Configuration* (represented by a Java class with the same name), which comes in two variants: a `State` object packaging one State Machine state, or a `RegionSet` object packaging one complex state (which contains the regions) and a list of occupied states inside (i.e. one for each region inside the complex state). Our goal is to associate one State Machine Configuration with each CSP process, so that we can identify at any given moment which configuration we are in.

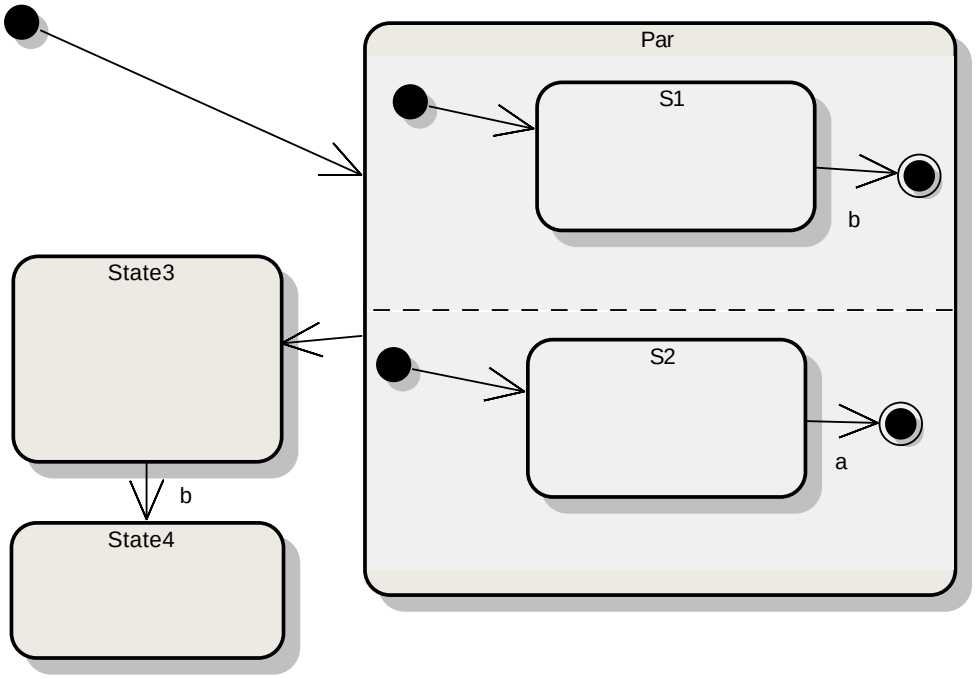

Figure 5.4: A State Machine with regions

Figure 5.4, for instance, shows a State Machine with one complex state containing two regions. The regions' initial states are called $Init1$ and $Init2$ and their final states are named $F1$ and $F2$.

Starting out in configuration $Init$ (the initial state), the State Machine will transition into the configuration $Par(Init1, Init2)$, from where it can transition further into $Par(S1, S2)$. Taking either event a or event b, it can transition into $Par(F1, S2)$ or $Par(S1, F2)$. After taking both events, it can pass from configuration $Par(F1, F2)$ to $State3$, and so on. Because the CSP equation system is optimized internally, not all of these states or configurations need be present in the Phase Event Automaton (and thus in the Transition Constraint System), but every transition in the lower-level systems should be attributable to a definite source and target State Machine Configuration.

As already described in sections 3.5.1 and 4.2.1, Syspect State Machines are translated into CSP and then into PEA phases. However, while generating CSP from State Machines, we do not yet convert the whole reachable process space. Instead only basic states (non-complex states) are translated into basic CSP processes, and complex states (containing basic states within their regions) are translated into interleaving compositions of those basic processes.

For instance, in figure 5.4, the second region's final state $F2$ is translated into $SKIP$,

$S2$ is translated into $a \rightarrow F2$, and $Init2$ is translated into $S2$. Par as a complex state translates into $(Init1 \;|||\; Init2)\;;\; State3$, i.e., the interleaving of both regions' processes followed by its own successor state.

As a product of simpler processes, an interleaved process has many transitions it can take, but these transitions (unlike the transitions of its constituting processes) are not generated explicitly at that point. Instead, the interleaved process will return a list of possible successor processes on demand, combined with the events that trigger them. Again, in our example, all individual states are translated, but for instance no translation has generated the process $(S1 \;|||\; F2);State3$ yet, although the State Machine's CSP translation can reduce to exactly this process. Only when we explicitly request the successors to $(Init1 \;|||\; Init2)\;;\; State3$ will they be generated. However, there are still no State Machine Configurations that these processes map to, so if we encounter a reduction to $(S1 \;|||\; F2);State3$ in an ARMC counterexample trace, we would not know which State Machine Configuration that process represents.

We considered recursively registering all successors of an interleaved process with the respective State Machine Configurations (right after creating the interleaved process in the translation phase), so that later we could look up every non-basic CSP process in the map, but some problems with equivalence surfaced.

In Java, equivalence is implemented by the `equals()` method, which is easy to write for data types with known semantics, such as CSP processes. Equivalent CSP processes are often used in different contexts, though, so multiple map entries would try to map a CSP process to *different* State Machine Configurations. For instance, a process *SKIP* can be used in different contexts, so they may not be considered equivalent. Likewise, a reference to a process P can be called in different places. By linking any occurrence of P with one calling state, we make *all* call sites refer to this state.

Therefore we chose not to make use of process equivalence, but to keep Java's built-in equivalence relation that checks for identity (i.e. the default implementation of `equals()`). This way one process does not have to map to multiple states (which would not work, since we need a mapping *function*), but every instance of an equivalent process can map to a distinct configuration. Instead of generating the whole process space and later using the map structure to look up equivalent processes, we chose to only generate map entries on demand.

Our implementation now uses two data structures to map CSP processes to State Machine Configurations: a `processes` map that maps basic processes to states (like explained above) and a `configs` map that maps both simple and complex processes to State Machine Configurations. The latter is what interests us, as it is the map we need to infer which configuration belongs to which process. However, as we only generate map entries for basic processes and for a limited number of complex processes, namely initial configurations for complex states such as $(Init1 \;|||\; Init2)\;;\; State3$ above, there will still be many processes that do not have a configuration assigned to them, like $(S1 \;|||\; F2);State3$.

The CSP to PEA translation phase recursively generates CSP successor processes exactly once, by extracting each process' successor set. Effectively, the CSP to PEA translation builds the whole process space that is reachable in the CSP equation system. At this point all processes are registered with their respective PEA phases, i.e., *every* PEA phase will have a CSP process, basic or complex, assigned to it.

For this reason, the `StateMachinePEAInfo` object operates on-demand. Each basic process and each interleaving of a complex state's initial states is already part of the map, as explained above, so we can look up what event caused the transition and if the transition's target process is also already registered with the map. For basic processes, it is, but for complex processes it is not. We now analyze which process in the interleaving construct took a transition, and what its successor was. For example, in the above example *Par*'s initial process $(Init1 \mid\mid\mid Init2)\,;\,State3$ is in the map, but its successors are not.

Remember that all basic processes can be looked up in the `processes` map, and that we can find out their successors: for instance $Init2$ maps to the state $S2$ and the instance of the *SKIP* process that is used by the process equation representing $F2$ maps to the state $F2$. When we have found the successor for the basic subprocess of the complex process that took a transition, we generate a new complex State Machine Configuration from the old one, and insert it into the `configs` map, so that the next time the configuration is already there. Effectively, we simulate a transition from an interleaved process (or a State Machine Configuration) ourselves and remember it, because other transitions may start from there.

The end result is that for every two CSP processes and event trigger we have two State Machine Configurations describing the transition's source and target.

Taking figure 5.4 as an example, state *Par* would actually be taken as the initial state (after optimizations), so the State Machine would start in $Par(S1, S2)$. By taking a simple test formula that can be fulfilled (such as $(\updownarrow a\,;\,\lceil true\rceil\,;\,\updownarrow b\,;\,\lceil true\rceil\,;\,\updownarrow b)$), we can trigger a series of transitions. The first CSP transition to take place goes from process $(Init2 \mid\mid\mid Init1)\,;\,State3$ to $(S2 \mid\mid\mid Init1)\,;\,State3$). As an initial configuration (for complex state *Par*), the former already is in the `configs` table and maps to $Par(Init2, Init1)$, while the latter one maps to `null`, as it is not registered in the table yet. The event in question is τ.

We find the first region to be responsible for the transition, specifically $Init2 \rightarrow S2$ using τ. We build a new State Machine Configuration and set its first component (the subprocess taking the transition) by looking up its mapping in the `processes` table. Owing to optimizations, process $S2$ maps to state $Init2$, so the new configuration is the same as the old one, unfortunately. Nonetheless, we register a map entry, mapping the target process $(S2 \mid\mid\mid Init1)\,;\,State3$ to the new configuration. After all, the next CSP transition must start from this configuration.

The second CSP transition works completely likewise, only that this time $Init1$ transitions to $S1$, but again, optimization prevents the high-level state from visibly changing.

The third CSP transition takes the event a to transition $(F2 \mid\mid\mid S1)\,;\,State3$, and this time we see $S2$'s afterstate $F2$ mapping to a different high-level state: $F2$, making the new configuration $Par(F2, Init1)$. This is illustrated in figure 5.5. Arrow [1] represents the source process' `configs` map entry. $S2$ is the process that changes to $F2$ using event a, so we look it up in the `processes` map and discover that its target process maps to the State Machine Configuration $F2$ ([3]). Consequently, we build a new State Machine Configuration from the one in [1] and install an entry of the current target CSP process $((F2 \mid\mid\mid S1)\,;\,State3)$ to it in `configs` ([4]). The output of the `StateMachinePEAInfo` object for this PEA edge would be "Par (Init2, Init1) –a→ Par (F2, Init1)".

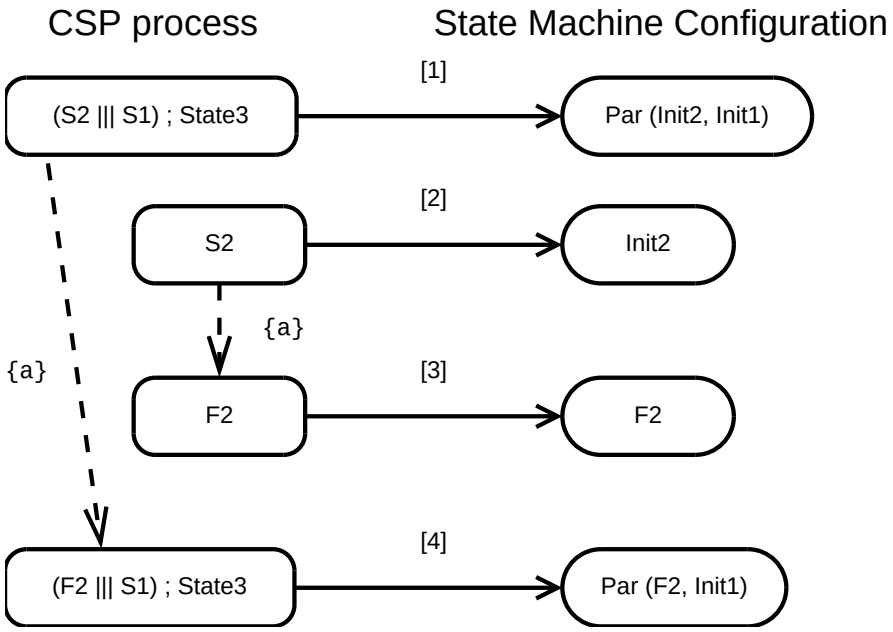

Figure 5.5: Resolving a transition in CSP

Event b takes the State Machine to configuration $Par(F2, F1)$. The next event is again b, but it transitions from CSP process $(F2 \mid\mid\mid F1)$; $State3$ to $State4$, without bothering with the states in-between (again, due to optimization). This time both processes are found in the `configs` map. As we already know, the first one maps to $Par(F2, F1)$, and the second one maps to $State4$, which is already registered in the map, because it is a simple state.

In a way for each process transition we generate the successor State Machine configuration on demand and remember it for the next transition which often starts from there. Inductively, all transitions will find their source configuration in the map, because a predecessor process already created it. Because all transitions in a complex state have to start from the state's initial configuration (with all regions' initial states), which is entered into the map at the beginning, the induction is sound. Because the number of states in finite, and because cycles to already processed states are guaranteed to be found in the map (as the targets must have been registered already), the induction also terminates.

5.2.3 Linking Duration Calculus to Phase Event Automata

When model-checking a specification constrained by Duration Calculus counterexamples, we want to know to what extent such a formula is already fulfilled. Because the formula *constrains* actual behavior, that is the best information we can get—it will never be 100% fulfilled, but a prefix of the formula might be.

In section 4.2.3 we mentioned that a Duration Calculus PEA represents in each of its states a *set* of possible counterexample phases it can be in. For phases with a lower time bound, the phase may not be fully active, i.e. it is *waiting*. This is also the information we are interested in.

The existing Duration Calculus translation in Syspect parses the Unicode syntax into a countertrace object that contains a list of DC phases. For our purposes, this list maps an integer index to an object describing the DC phase numbered by that index.

The DC countertrace gets further translated by the `Trace2PEACompiler` that implements the translation algorithm described in [Hoe06]. We did not need to change the algorithm in any way, only to extract information about what DC phase set maps to which PEA phase. Wherever a new PEA transition is generated, we therefore map its destination state to the respective DC phase set.

For any Phase Event Automaton edge we encounter, we can thus provide information about what DC phases are satisfiable at the current state. An example output of the `DCPEAInfo` look like the following. Note how a phase's entry events are displayed at the beginning of the phase, before the phase's duration part.

In Phases:
$\lceil TRUE \rceil$
$a \,;\, \lceil TRUE \rceil \wedge \ell \leq 3$
$\lceil !n \leq 2 \rceil$

5.2.4 Linking Test Formulae to Phase Event Automata

While Test Automata are not part of CSP-OZ-DC and per se are not related to the specification whose behavior we want to observe, we decided that it is helpful to the user to not only see the specification's behavior, but also in what way a trace fulfills a test formula.

Test Formulae use the same infrastructure as Duration Calculus countertraces, with the difference that they translate to PEA Test Automata which are characterized by having final states. Where a DC PEA constrains behavior, i.e. can never violate the counterexample, a Test Automaton has a final state that is reachable, should the formula be fulfilled.

By displaying the same information for Test Formulae as for DC specifications, we allow the user to observe how a specification relates to test formulae, which test formula phases the specification can reach at a given moment.

While a Duration Calculus formula is only one formula and translates to exactly one Phase Event Automaton, a Test Formula may translate to more than one automaton, namely in the event that it conjuncts multiple Test Formula traces. Since the test formula compiler can deliver us multiple automata, it also has to return us multiple structures mapping transitions to DC phase sets. Internally the test formula compiler uses the `Trace2PEACompiler` for its translation, so we could get the data just like we did in the Duration Calculus case.

Chapter 6

Implementation and Visualization

In this chapter we will describe the visualization of counterexample traces in Syspect and give a short overview of the Syspect application's structure and of our contributions. Many parts of Syspect and many algorithms used in it have already been mentioned previously in the text, but because Syspect is quite a complex application, it is appropriate to outline the changes that were necessary to implement our objective.

6.1 Syspect

Syspect is an application built on the Eclipse rich client platform.[1] It is structured as a collection of *plugins*, i.e., simple components that work together via well-defined interfaces.

The plugins we had to work with to implement the objectives of this thesis were `SyspectModel`, `SyspectOZDCUtil`, `SyspectProperties`, `SyspectCSPOZDCExport`, and `SyspectPEAExport`, as well as the PEA toolkit plugin `PEA-Tool`. Additionally, we contributed the `SyspectARMCExport` plugin.

SyspectModel

This plugin exports the main Java packages containing the Syspect model interfaces and implementation classes, i.e., the classes holding the information the Syspect user is editing, like State Machines, Duration Calculus tags, or class diagrams.

We extended the `CapsuleImpl` class and the `ICapsule` interface to include a list of Test Formulae.

SyspectOZDCUtil

`SyspectOZDCUtil` provides classes for parsing Object-Z and Duration Calculus formulae and a graphical editor for creating formulae in both notations. We added a parser for test formulae in the `de.syspect.ozdcutil.parser.testformula` package.

[1] `http://wiki.eclipse.org/index.php/Rich_Client_Platform`

SyspectProperties

The `SyspectProperties` plugin is responsible for the "Properties" view in Syspect. We extended the properties view for capsules to allow editing of Test Formulae (in conjunction with the Test Formula parser from the `SyspectOZDCUtil` plugin).

SyspectCSPOZDCExport

This plugin harbors the model classes for CSP and Object-Z, algorithms to generate both from Syspect's UML models, and export wizards to export the user's specification to XML or LaTeX. The class `SyspectSM2CSPConverter` is the place where we create the `processes` and `configs` maps to link CSP processes to State Machine Configurations.

SyspectPEAExport

The `SyspectPEAExport` plugin contains another export wizard (to export a Syspect specification as a PEA encoded in XML) and converters to transform the CSP, Object-Z, and Duration Calculus parts of the specification into PEAs.

We augmented the plugin to generate mapping structures during translation, i.e. we made the converters in the Java package `de.syspect.export.pea.converter` apply the linkage mechanisms outlined in chapter 5 and construct appropriate `PEAInfo` instances. All `PEAInfo` implementations are also part of this plugin, in package `de.syspect.export.pea.info`.

It would have been preferable to have the `PEAInfo` classes reside in the `SyspectARMCExport` plugin, as they are only used from there, but they have to be visible to the converter classes, so we chose to put the `PEAInfo` classes into the `SyspectPEAExport` plugin as well.

PEA-Tool

The PEA toolkit plugin provides lots of algorithms to generate Phase Event Automata, to combine them and to export them to model-checkable formats like Transition Constraint Systems.

This plugin proved to be very stable, so we did not have to perform many changes, only some small adjustments to make the ARMC output correctly translate Z predicates, and some functions to extract information from the countertrace-to-PEA compiler for use in the `DCPEAInfo` class. The `ZDecision` class, which represents Z predicates inside Constraint Decision Diagram (CDD) trees (introduced in [Hoe06]), was extended with some simple string analysis functions to generate whole CDD trees instead of single `ZDecision`s, where appropriate.

As the functionality in `ZDecision` is rather rudimentary, it would be desirable to employ advanced parsing algorithms to allow the user to express complex Z expressions more freely. Unfortunately, the author's attempts to use the *Community Z Tools*[2] (in

[2] `http://czt.sourceforge.net/`

particular the visitor[3] classes for analyzing abstract Z syntax trees) for this purpose did not turn out successful, which is why the current mechanism was put in place.

SyspectARMCExport

The ARMC export plugin provides the user interface elements needed to interact with the ARMC export and verification algorithms and all necessary helper classes for this task. We developed two wizards for exporting and verifying a specification with ARMC, shown in figures 6.1 and 6.2.

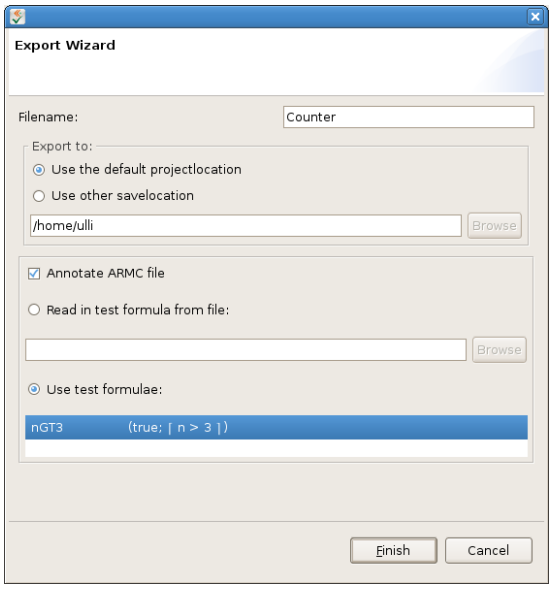

Figure 6.1: ARMC Export Wizard

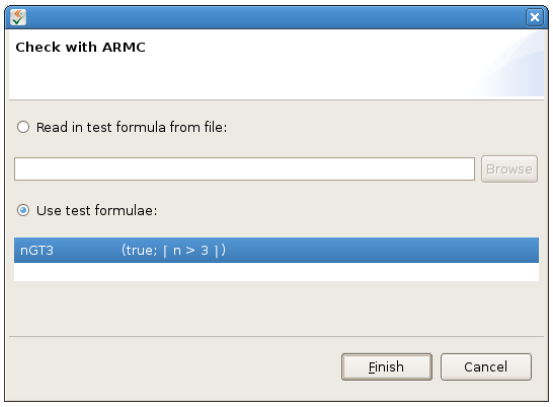

Figure 6.2: ARMC Verification Wizard

When exporting a specification as an ARMC program, the user has the option to turn off annotations, which will decrease the size of the output file and hopefully improve model-checking efficiency. When verifying, the output is always annotated, so that we can later find out which ARMC transitions relate to which Phase Event Automata edges.

The following sections explain in more detail the output rendering and the translation process in the `SyspectARMCExport` plugin.

6.2 Visualization of Counterexample Traces

If the model-checker verifies a program as correct, we pop up a small dialog announcing success. If during model-checking a test formula can be fulfilled, the model-checker ARMC will provide us with a counterexample trace listing all transitions in the Transition Constraint System.

We map these ARMC transitions to edges in the combined Phase Event Automaton, from which we infer the edges in the individual PEAs. These edges are then used to

[3]The *visitor pattern* is a design pattern (see [GHJV95]) used to define an algorithm operating on a variant data type separately from that data type, in order to improve modularity.

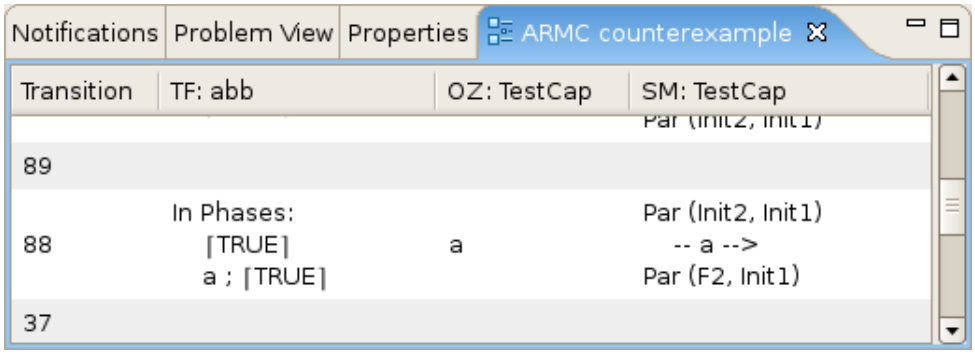

Figure 6.3: ARMC Counterexample view

extract data from the appropriate `PEAInfo` objects of the respective CSP, Object-Z, or Duration Calculus part of the specification (and—less importantly—the test formula as well).

The result is that for every transition ARMC made we have information describing the actions of every part of the specification. We found the most concise and readable rendering of this to be a table, listing transitions on one axis and the various specification parts and test formulae on the other.

To implement the visualization, we decided to create a new *view* in the Eclipse RCP framework.[4] This means that the model-checking results will be displayed in a tab next to the properties, and that the user will be able to review the results while also regarding a class diagram or state machine in Syspect's editor area.[5]

An example view is shown in figure 6.3. The reason every other line is blank is that in order to translate Phase Event Automata into Transition Constraint Systems, every other transition accounts for the changes in the system clocks, i.e. transitions take turns adjusting the clock and performing actual work. This is necessary, because ARMC has no built-in notion of time. For further details we refer the reader to [Hoe06].

6.3 The Verification Process in Detail

Figure 6.4 sketches the data flow in Syspect, from the specification model to the Transition Constraint System, and from ARMC's output back to a visualization of the counterexample in Syspect. The following sections describe the implementation in more detail.

6.3.1 Export to ARMC

The ARMC generation process is implemented in the class `ARMCExporter` in package `de.syspect.export.armc`. As an input the class needs an output file for the ARMC

[4] Alternatively, Eclipse allows the creation of an *editor*, but with the model-checking output being read-only and not related to an editable document, we decided to go with the view.

[5] Eclipse RCP assigns to editors and views separate areas of the application window, so all editors share one large area, while various views can be positioned around the editor area, as convenient.

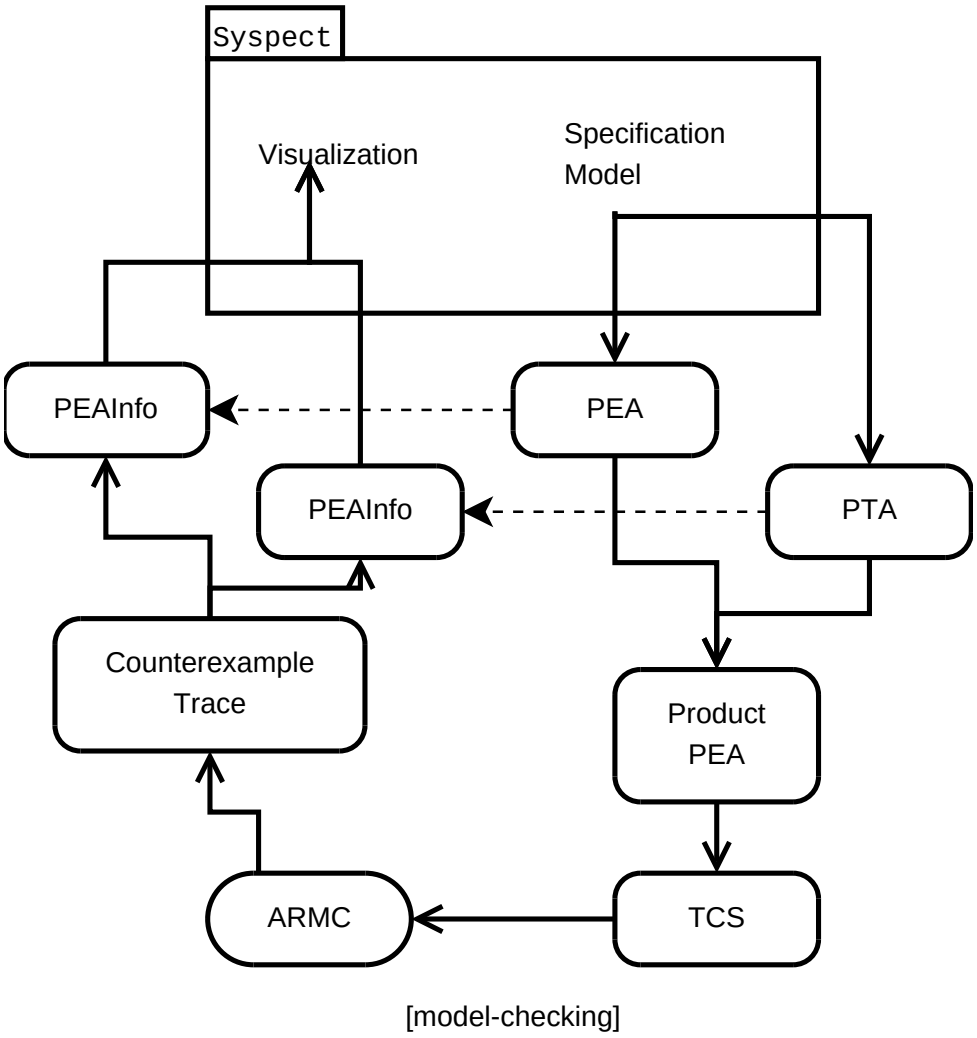

Figure 6.4: An overview of the data flow involved in model-checking

program, a list of test formulae to check (the files contain the XML-encoded test formulae, but the list of formulae is needed to get their names, as these are not incorporated in the files), a list of capsules, projects, or both to process, and a progress monitor, to give visual feedback how the export is progressing.

First, the method `generatePEAs()` constructs a list of Phase Event Automata (and PEA Test Automata, which form a subclass of PEAs): all test formulae are converted into PTAs, and information is extracted to construct according `DCPEAInfo` objects (as test formulae are very similar to Duration Calculus formulae in their translation, we let one class display information for both).

Next, we iterate over the list of capsules and projects and convert each capsule (including all capsules that are part of a project) into a CSP-OZ-DC class using the `UML2CSPOZDCJConverter` provided by Syspect. For each such class we first compile a list of variables used in the Object-Z part of the class, because all variables used must be declared for ARMC. Then we invoke various translation algorithms to convert the constituents of the class into PEAs: Object-Z, CSP, and all Duration Calculus counterex-

ample formulae. From each translator object we then extract a `PEAInfo` object containing the linkage structures to be able to relate PEA edges back to high-level information later.

The second—usually much more time-intensive—part is implemented by the `export()` method, which can be parametrized whether to produce ARMC annotations or not. Every PEA that has been produced so far is rewritten: each edge is checked for variables that are not being assigned, and assignments of the form $x' = x$ are generated for all these variables. Additionally, we generate annotations if desired, as explained in section 5.1. Finally, we expand all terms including $=$ or $\neq$ operations to use combinations of $<$ and $\leq$ operations. This way, even if building the disjunctive normal form of a PEA edge inverts an operation, there will be no $\neq$ operations at the end that would be untranslatable into ARMC Transition Constraint Systems.

All rewritten PEAs are then combined into one parallel automaton, all bad states are merged into one single bad state, and all events are removed from the final automaton. This is done because in the combined automaton all synchronizations through events have already been performed—each edge in the automaton performs all operations that would have been performed at the same time in the source automata. Since ARMC knows no events, they are removed, so just the actual operations remain. All these operations manipulating PEAs are performed by classes in the PEA toolkit.

At the end, we call the PEA toolkit to write the combined Phase Event Automaton into the designated output file as a Transition Constraint System.

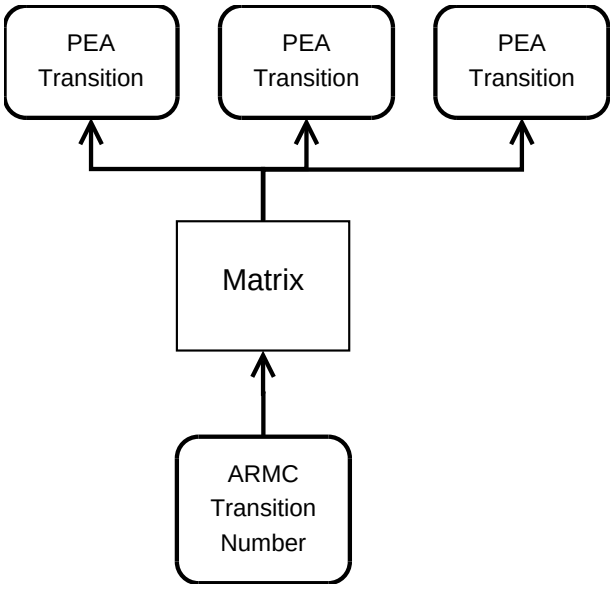

Figure 6.5: Matrix mapping ARMC to PEA edges

6.3.2 Model-Checking

Also implemented in the `ARMCExporter`—because it already has all the data needed—are the methods `runARMC()` and `showResults()`.

The former uses `ARMCParser` to parse the generated ARMC file to read in the list of available transitions. From the list of transitions we build a matrix that maps ARMC transition numbers and PEA numbers to PEA transitions (see figure 6.5).

Subsequently we call the ARMC model-checker and analyze its output for either a correctness result or a counterexample trace including a list of transition numbers. The model-checking result—success, error, or a counterexample—is returned to the caller.

The `showResults()` method finally displays either a success message, an error dialog, or opens the `CounterExampleView` to display high-level information for all Phase Event Automata, passing it the matrix, the list of ARMC transitions, and the list of `PEAInfo` objects.

6.4 Example Rehash

Recall the examples from section 3.6, *Counter* and *CounterDC*, and the test formulae in section 4.1.1. Now that we have discussed the internals of our implementation, it might be interesting to see the results of model-checking the examples with their test formulae.

For the *Counter* example we wanted to know if the counter can ever attain a value of $n > 3$. Model-checking the example returns the message that the "program is correct," i.e., the counter can never reach a value of $n > 3$.

For *CounterDC* we mainly imposed the restriction that only one increment operation was to happen in more than three seconds, or that the duration between two such operations should never be less or equal to three. The test formula was a test if the counter could go from 0 to more than 1 in less or equal to seven seconds.

Figure 6.6 shows the counterexample trace output for *CounterDC*. The second-left column contains the test formula's phases, the middle one the Object-Z part, and the right one the Duration Calculus formula's phases.

We can observe that two increment operations occur, which is the only way to meet the test formula's condition. The decrement operation at the end is insignificant. In its last transition, the PEA test automaton representing the test formula already is in the final state, so the other PEAs can take arbitrary transitions without changing the result in any way.

The DC part oscillates between two states: one with no specific phase, and one where the automaton could be in a post-increment phase (the "increment" is an entry event, so firing the event increment could have put the DC automaton in that phase), followed by a duration of less or equal to three. As soon as that duration is exceeded, we see that the automaton can no longer stay in that phase—it has to leave and thus return into its only previous phase. Note that the automaton can always be in a *set* of phases (so it could "decide" to be in the $\lceil TRUE \rceil$ phase, not in the other one), yet in some phase sets certain activities (like firing the increment event) are forbidden, because the automaton *could* be in a phase where they are strictly disallowed. In our case, the automaton can not partake in increment events, because one of its possible phases disallows this.

Notifications | Problem View | Properties | ARMC counterexample

Transition	TF: nGT3	OZ: CounterDC	DC: ThreeSecsPerInc
2			
56			
52	In Phases: ⌈TRUE⌉ ⌈n≤0⌉ ⌈TRUE⌉ ∧ ℓ ≤ 7	initialize PEA: n = 0	In Phase: ⌈TRUE⌉
69			
58	In Phases: ⌈TRUE⌉ ⌈TRUE⌉ ∧ ℓ ≤ 7	increment: n′ = n + 1	In Phases: ⌈TRUE⌉ increment ; ⌈TRUE⌉ ∧ ℓ ≤ 3
181			
167	In Phases: ⌈TRUE⌉ ⌈TRUE⌉ ∧ ℓ ≤ 7		In Phase: ⌈TRUE⌉
156			
148	In Phases: ⌈TRUE⌉ ⌈TRUE⌉ ∧ ℓ ≤ 7 ⌈!n≤1⌉	increment: n′ = n + 1	In Phases: ⌈TRUE⌉ increment ; ⌈TRUE⌉ ∧ ℓ ≤ 3
137			
133		decrement: n′ = n − 1	In Phase: ⌈TRUE⌉

Figure 6.6: Counterexample trace for *CounterDC*

Chapter 7

Conclusion

7.1 Summary

In this thesis we laid out a mechanism to infer source Phase Event Automata edges from a Transition Constraint System and to link those PEA edges to high-level specification elements.

We described how the first part is performed through annotations, and the second part by building maps during the compilation process within Syspect.

While our implementation works well, it would profit from improvement. The following section discusses some topics for future research and development.

7.2 Future Work

The various parsers present in Syspect could use some makeover. Further areas of interest include the translation of more Z expressions and optimization of ARMC programs for faster model-checking. The accuracy of the State Machine translation could also be improved.

7.2.1 Parsers

The Z parser is part of the `ZDecision` class in the PEA toolkit right now, but it only supports basic operator precedence. Furthermore, being integrated only into a back-end plugin, there is no possibility to provide meaningful error messages to the user. Usability would benefit from a more powerful parser with a well-documented grammar, integrated with the Syspect graphical user interface.

Test formulae suffer from a very rigid syntax, as explained in section 4.1. An LR parser would probably ameliorate the situation and allow more liberal placement of parentheses. Error handling is usually more complex in LR parsers, but with a more flexible grammar, not many error cases would need to be handled.

7.2.2 Translating Z

Currently, only arithmetic operations and comparisons can be translated to ARMC. Research could devise ways to model-check other Z expressions as well.

7.2.3 Translating State Machines

With the translation as implemented now, State Machines do not allow the most accurate linking to ARMC transitions (see 5.2.2). Instead of converting State Machines to CSP and further into PEAs, it is possible to develop a direct translation from State Machines to PEAs. With both being automata, such a conversion would likely keep down automata size by permitting better optimization techniques and could improve linking PEA edges to high-level State Machine transitions.

7.2.4 Optimizing ARMC

Finally, there are ways to optimize ARMC output. We annotate ARMC files with PEA edge numbers and expand $=$ and $\neq$ operations, which results in rather big Transition Constraint Systems.

By careful analysis of what operations PEA conditions contain, expansion of $=$ and $\neq$ could be avoided completely or reserved only for cases where it is truly necessary.

Another feasible optimization is to omit annotations from the ARMC file used for model-checking. Since annotations are static, they are not required at run-time. It has to be ensured, though, that the number and order of transitions generated will be exactly the same with and without annotations. This includes both the algorithms used and Java classes such as `HashMap` that might behave differently on each run, or might produce different transitions when the input is different.

With this optimization, a first run could produce an ARMC file with annotations and scan it for PEA edge numbers, while a second ARMC file without any annotations could be used for quicker model-checking.

We manually exported the *CounterDC* example with and without annotations, and ARMC model-checking time decreased from 1.48 seconds to 1.30 seconds (averaged over five runs), which shows the feasability of this optimization. We suspect that for larger specifications using more PEAs the time difference will be more significant, due to the higher number of variables.

Bibliography

[Fre95] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, 1995.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[HM05] Jochen Hoenicke and Patrick Maier. Model-checking of specifications integrating processes, data and time. Technical Report 5, AVACS Project, 2005.

[HO02] Jochen Hoenicke and Ernst-Rüdiger Olderog. Combining specification techniques for processes, data and time. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer-Verlag, May 2002.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoe06] Jochen Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD Operating System*. Addison Wesley, 1989.

[Mey05] Roland Meyer. Model-checking von Phasen-Event-Automaten bezüglich Duration Calculus Formeln mittels Testautomaten. Diplomarbeit, University of Oldenburg, August 2005.

[MFR06] Roland Meyer, Johannes Faber, and Andrey Rybalchenko. Model-checking Duration Calculus: A practical approach. In *International Colloquium on Theoretical Aspects of Computing, ICTAC*, 2006.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[ML06] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform*. Addison Wesley, 2006.

[MORW06] Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim. Integrating a formal method into a software engineering process with UML and Java. Under consideration for publication in Formal Aspects of Computing, 2006.

[RP07] Andrey Rybalchenko and Andreas Podelski. ARMC: the logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages, PADL*, 2007.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[ZH04] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.

Index