Model Checking Combined Z and Statechart Specifications

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik

Robert Büssow

Promotionsausschuss

Vorsitzender:	Prof. Dr.-Ing. Günter Hommel
Berichter:	Prof. Dr.-Ing. Stefan Jähnichen
Berichter:	Prof. Dr. Peter Pepper

Vorgelegt von Diplom Informatiker Robert Büssow
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
Tag der wissenschaftlichen Aussprache: 18.11.2003
genehmigte Dissertation

Berlin 2003
D 83

Contents

Zusammenfassung

Eine der bedeutendsten Herausforderungen der Softwareentwicklung besteht darin, einen Entwicklungsprozess zu garantieren, der Fehlerfreiheit nicht nur gewährleistet sondern auch nachweisbar macht. Beides ist von besonderer Bedeutung, wenn sicherheitskritische Systeme entwickelt werden, etwa in den Bereichen der Medizin, der Produktionssteuerung oder der Verkehrstechnik. Softwarefehler können hier lebensbedrohlich sein. Aus diesem Grund ist es meist auch notwendig, die Fehlerfreiheit der Software einem Dritten nachzuweisen. Die Steuerung einer Verkehrsampel muss beispielsweise nicht nur fehlerfrei funktionieren, sondern auch vom TÜV abgenommen werden.

Der Einsatz *formaler Methoden* stellt einen vielversprechenden Ansatz dar, diese Probleme zu lösen. Formale Sprachen haben gegenüber den üblichen, nicht-formalen Methoden (umgangssprachliche Spezifikationsdokumente oder Spezifikationssprachen ohne eindeutige Semantik) den Vorteil einer eindeutigen Semantik. Damit können Anforderungen an ein System eindeutig beschrieben und seine Eigenschaften mathematisch bewiesen werden. In der Praxis haben sich diese Methoden allerdings bisher noch nicht durchgesetzt. Zwei herausragende Ursachen hierfür sind:

1. Die formalen Spezifikationssprachen orientieren sich meist mehr an mathematischer Eleganz als an einfachen und intuitiven Sprachmitteln. Das stellt eine große Hürde für den praktischen Einsatz dar. Die Spezifikationssprache $\mu\mathcal{SZ}$ [11] versucht dieses Problem zu lösen. Sie verbindet die von Harel [26] entwickelte und in der Industrie akzeptierte grafische Sprache Statecharts mit der formalen Sprache Z. Damit liegt eine intuitive Sprache vor, die den Anforderungen einer formalen Sprache genügt.

2. Formale Spezifikationen haben zwar eine präzise Semantik, sie lassen aber dem Spezifikateur immer noch die Freiheit, inkonsistente oder fehlerhafte (nicht den tatsächlichen Anforderungen entsprechende) Spezifikationen zu erstellen. Andererseits ermöglichen sie es, Konsistenz und Eigenschaften formal zu beweisen und so zu einer fehlerfreien Spezifikation zu gelangen. Werden solche Beweise nicht geführt, ist gegenüber einer nicht-formalen Spezifikation wenig gewonnen. Um die aufwändige Beweisführung praktikabel zu machen, ist eine möglichst weitgehende Automatisierung unverzichtbar.

Der Nachweis der Konsistenz sowie der Eigenschaften einer $\mu\mathcal{SZ}$ Spezifikation ist Ziel der vorliegenden Arbeit. Hierfür werden Model Checking Techniken eingesetzt. Um dies zu ermöglichen, wird die $\mu\mathcal{SZ}$ Spezifikation in drei Schritten übersetzt:

1. Übersetzung des Statechartanteils einer $\mu\mathcal{SZ}$-Spezifikation nach Z. Damit werden zusätzlich die Semantik der Statecharts und die Semantik die Integration von Statecharts und Z definiert. Außerdem erlaubt diese Vorgehensweise andere, reine Z-Werkzeuge für die Analyse zu benutzen.

2. Vereinfachung der Z-Spezifikation in ein vereinfachtes Z (*Simple Z*), das vom Sprachumfang der Eingabesprache eines Model Checkers entspricht. Dieser Schritt erlaubt es, sowohl $\mu\mathcal{SZ}$-Spezifikationen wie auch reine Z Spezifikationen für das Model Checking vorzubereiten. Das vereinfachte Z kann leicht in die Eingabesprache eines Model Checkers übersetzt werden.

3. Übersetzung von Simple Z in die Eingabesprache des SMV Model Checkers von McMillan [38]. Der Model Checker kann dann Konsistenz und Eigenschaften der Spezifikation beweisen.

Chapter 1

Introduction

1.1 Safety Critical Systems

One of today's major problems in software engineering is to achieve a high and comprehensive quality standard for the software development process, in order to maintain a reliable high quality for the resulting products. This holds particularly true for safety critical systems, where failure of the software may have life-threatening consequences. Here, not only the quality of the software itself is important, but also the ability to convince a third party of this very quality.

The usage of *formal methods* is one promising approach to achieve these goals. Roughly speaking, formal methods introduce mathematical precision to the development process. They do so by using formalisms with well defined semantics, and so stipulate formal proofs to verify development steps.

This approach is all too well feasible in theory. In practice, however, one will encounter various problems that impede a consequent usage of formal methods:

- The formal character of the proposed languages and the need to use them for every aspect of the described system makes them too bulky. The reason for this is that they often times concentrate more on the mathematical elegance of their underlying semantics than on comfortable and intuitive usage.
- As adequate tool support is often missing, implementation of the formal proof obligations becomes practically impossible, because without any ools, these proofs are quite complicated, and their development takes a lot of time.

To tackle these problems was one of the objectives of the Espress project.

1.2 The Espress Project

The Espress project was a joint project of the Daimler Benz AG, the Robert Bosch AG, Fraunhofer ISST, GMD First, and the Technical University of Berlin. The project aimed at a software development method for safety-critical embedded systems. The development

method was supposed to support the development process from specification to testing, being applicable throughout today's practice of software engineering.

In the course of the project, the specification formalism $\mu\mathcal{SZ}$ [11] was developed. $\mu\mathcal{SZ}$ uses techniques that are well-established within the scientific community and in industrial practice. Harel's Statecharts [26] on the one hand, and the Z formalism on the other hand, were combined to a powerful specification language. In that, Statecharts serve the comprehensive graphical and operational description of the specification's behavioral aspects, while the Z formalism (see Spivey [48]) is used for the axiomatic specification of data, data invariants, and transformation. With this combination, the advantages of the formalisms in their fields (operational behavior and axiomatic data specification, respectively) can be exploited. The usability of the formalism and its advantage over other formalisms in its application domain was shown in various case studies which were conducted throughout the project [10, 8, 13, 9, 12].

1.3 The Zeta System

An important motivation of combining existing notations instead of designing a new language from scratch, is to reuse existing tools. This has two major advantages. Users can continue to work with the tools they are acquainted with (in the context of Espress, this is the Statemate tool, widely used within the industry for embedded systems). Also, integrating existing tools is a lot less expensive than implementing newly designed tools. Therefore, tool support for $\mu\mathcal{SZ}$ was built integrating existing tools.

In the course of the Espress project, the Zeta system was developed to provide tool support for the $\mu\mathcal{SZ}$ language.[1] Zeta offers an open framework for the integration of existing and newly written tools. These tools directly support the notations which are semantically integrated as *plugins* into the $\mu\mathcal{SZ}$ notation. Zeta also supports generic tools based on the Z or $\mu\mathcal{SZ}$ language (type checkers, provers, model checkers, compilers), which can process the integrated notations by their mapping to Z and/or by proprietary mappings, and other tools, e. g. tools for generating documentation.

The Zeta system provides a general framework for the integration of different tools and languages, so called *adaptors*. The system itself is independent from the supported languages. Only the adaptors are language specific.

The Zeta system handles different specification languages, represented in different formats (e. g. abstract syntax, Postscript, or LaTeX), the *content*. Adaptors create, translate, and integrate the content. For example, a Zeta adaptor can create content in reading and parsing a file from the file system or fetching a model from the Statemate tool. Other adaptors translate Z abstract syntax to type-checked abstract syntax or SMV. The Zeta system is responsible for providing the input content of an adaptor by calling other adaptors.

The Zeta system is responsible for the following tasks:

- Upon request of some content, Zeta builds up a tool chain from the registered adaptors

[1] Zeta is available for download at http://uebb.cs.tu-berlin.de/zeta

that compute the requested content. Once computed, the content is cached by the system and recomputed if its sources have changed. This functionality is quite similar to the well known *make*-command. However, writing *makefiles* is not needed, since the configuration can be done automatically. The tool chain that is built up for model checking is depicted in Figure 1.2 on page 14.

- Zeta provides a uniform user interface. The adaptors register the commands they can perform and options that can be configured. The system builds a user interface for that. The user can chose between three different user interfaces: a textual, a Swing based graphical one, and an integration with the *Xemacs* editor. The system also performs the error reporting.

The Zeta system is implemented in Java. For the adaptors, a Java API is provided. The adaptors may either implement their translation themselves or integrate an external tool via JNI, RMI, CORBA, or program execution.[2]

The adaptors exchange their content as Java objects, in the respective content format. The Zirp format (*Z intermediate representation*) that is used to exchange $\mu\mathcal{SZ}$ abstract syntax, carries particular relevance in this context.

The most important adaptors are:

- The LaTeX adaptor, parser and type checker. It is responsible for reading Z/$\mu\mathcal{SZ}$ files, parsing and type checking them. It produces Zirp content with type annotations. The parser is also in charge of integrating specifications. For example, if the specification contains a Statemate Statechart, the LaTeX adaptor will replace the reference to the Statemate model by its Zirp representation. The Zirp representation for the Statemate model is provided by the Statemate adaptor.
- The Statemate adaptor (together with the StmToZirp adaptor) fetches Statemate models from the Statemate tool. It can retrieve models in Zirp and for documentation in Postscript format. The Statemate adaptor asks Statemate to write the model to file via the dataport. The StmToZirp adaptor then parses the file.
- The model checker adaptor translates a Zirp model into the input language of a model checker and performs model checking by using this very checker.
- The Z execution adaptor Zap (see Grieskamp [25]) rewrites and executes Z specifications. This adaptor is not used for model checking.
- The Isabelle adaptor connects Zeta to Isabelle/Holz (see Kolyang, Santen and Wolff [35]). This adaptor is not used for model checking.
- The LaTeX documentation adaptor formats the specifications to printable documents. This adaptor includes a special LaTeX style that does the $\mu\mathcal{SZ}$ and Z formating.

[2] For example the Statemate adaptor uses the Statemate dataport library to get Statemate models. The library is provided as shared object on Solaris and DLL on Windows NT. JNI is used to integrate the dataport library on Solaris. On Windows NT, it is not possible to dynamically link the library to a Java VM. Therefore, it runs in a separated process and is connected via Windows RPC. There is no dataport library for Linux. Thus, a special server, running on Solaris, is connected via RMI.

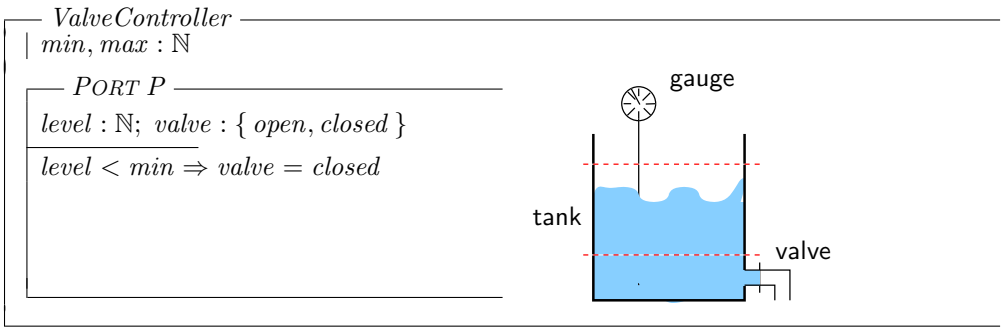


Figure 1.1: Specification of a Valve Controller

The Zeta system is implemented in Java. It runs under Linux, Solaris, and Windows NT.

1.4 Consistency of the Specification

With $\mu\mathcal{SZ}$, one can choose whether a requirement is specified operationally with Statecharts, axiomatically with Z or, redundantly, using both formalisms. This allows for the specification of requirements on different levels of abstraction and from different perspectives. This way, $\mu\mathcal{SZ}$ offers the possibility to formulate requirements as they are, rather than reformulating them—a vital capacity for any specification method.

The specification of a valve-controller, depicted in Figure 1.1, may serve as an example: The controller has to keep the water level in a tank below a predefined limit. In order to do so, it measures the water level and controls a valve to evacuate water from the tank. For that, the valve must stay closed whenever the water level is beneath a certain threshold. An additional requirement may apply, such as: "Do not open and close the valve too often". The specification also contains the actual control algorithm for the valve. The algorithm might be quite complex, since it has to fulfill all requirements. Because of its complexity, it can be difficult to establish whether the algorithm really fulfills all requirements or not. The advantage of the $\mu\mathcal{SZ}$ method is that a single requirement can be specified without reformulating it or considering the other requirements. The specification can be built up gradually, collecting requirements and adding them to the specification. In the end, the actual control algorithm is specified, then including all requirements.

It is not guaranteed, however, that the control algorithm fulfills all properties. Deficiencies may occur because the requirements might be contradictory by themselves or because a single requirement was overlooked. Such a specification is called *inconsistent*.

The problem of inconsistency is common among all axiomatic specification formalisms. From the semantic point of view, there is no program that fulfills an inconsistent specification. It is thus impossible to implement such a specification. However, finding all inconsistencies will often be just as impossible, as every single property of the specifi-

cation would have to be checked against any other property. An engineer who has to implement the controller for the above example will probably consider only the specification of the control algorithm, because it is the most concrete one, and ignore the rest. He will therefore not notice the inconsistency—the deficiency so moves onto the next development stage. The later such a failure is detected, the more costly or even dangerous it may become. Avoiding it at an early stage is therefore essential.

Moreover, suppose that the minimal water level is an important safety constraint and that there is a third party who reviews the specification. It appears easy to convince the reviewer that the property is being considered. However, it is also necessary to prove that the property is not violated somewhere else in the specification, i. e. that the specification is consistent. Without tool support, this proof is very difficult, since the entire specification has to be considered.

The reason for inconsistencies is often a deep-seeded misconception of the specified system. Suppose the second requirement of the presented example were more restrictive: "There must be a minimal delay between opening and closing". Now it can happen that the water level drops beneath the threshold, but the delay has not yet elapsed. In this situation, it is not possible to fulfill both requirements. The two requirements are thus contradictory and the specification is inconsistent. This disagreement comes from the original requirements themselves. Inconsistencies of such origin need to be detected in the earliest stage possible.

Axiomatic specifications and redundancies may lead to inconsistent specifications. However, an inconsistent specification is better than a consistent one with missing requirements. With an inconsistent specification, there it at least the chance that the error is detected. Inconsistencies are often not a problem of the used specification technique but of inconsistent requirements. It is an advantage, if such inconsistencies become visible in the specification. Detecting inconsistencies in the specification manually can prove to be extremely complicated.

The objective of this work, therefore, is to provide a technique that is able to check an $\mu\mathcal{SZ}$ specification for consistency automatically. This technique provides the following advantages:

- Misconceptions and contradictory requirements are found at the beginning of the development process.
- High-level specifications of properties can be used to convince a third party that the property is fulfilled.
- If changes are made to the specification, it can be checked immediately whether important properties are still fulfilled.
- The developer is forced to provide a complete and consistent specification. He also gets an impression of the behavioral characteristics of his specification. Thereby, using $\mu\mathcal{SZ}$ will increase the quality of the specification.

In general, the development can be performed more efficiently, since failures can be detected in the early development stages and the quality of the specification is increased.

1.5 Model Checking MSZ Specifications

For verification, we use the technique of model checking (see Chapter 4 on page 35). Model checking verifies a property by enumerating all possible states of a system. For this, the set of states has to be finite and "small". Its size determines the feasibility of model checking. This restricts the applicability of model checking and hereby the applicability of the presented approach. However, model checking carries the important advantage that proofs are conducted without user interaction. Model checking is well suited for control problems, as they are entailed by the utilization of Statecharts. In addition, Statechart models are finite. Therefore, model checking is well suited for proving properties of Statechart models.

Z models, on the other hand, are usually infinite or at least allow for no easy enumeration of their states. In order to handle these problems, a subset of Z was defined (see sections 5.1 on page 55 and 7.4 on page 103). In $\mu\mathcal{SZ}$, Z is used to specify the data aspects of control problems. For these applications, the proposed subset of Z is sufficient.

There are a number of model checking tools which can be utilized for the actual model checking task. For this work, McMillan's SMV model checker [38] was used. Most of the presented solution is independent of the actual model checker used. It would be easy to use some other tool. The SMV specific parts are presented in Chapter 8 on page 137.

In order to use an existing model checker, the model and the relevant properties have to be translated into the input language of the model checker. In $\mu\mathcal{SZ}$ the model is specified with Statecharts and Z whereas its properties are described with Z, enriched by temporal logic.

1.5.1 Translating the Model

In brief, model checkers handle models that are defined by a set of Boolean variables, a predicate defining a state transition relation, and a predicate defining the initial states. The predicates are defined by Boolean operators ($\wedge, \vee, \Rightarrow \Leftrightarrow, \neg$) over Boolean variables. In case of the state transition relation, two sets of the variables are used—one for the pre-state and one for the post-state.

For this reason, Statecharts have to be translated into a state transition relation. In $\mu\mathcal{SZ}$, Statecharts contain Z expressions in the labels of the Statechart state transitions. Hence, for the Statechart translation into a state transition relation, Z expressions have to be translated as well. In order to reduce the complexity of the Statechart translation, Statecharts are first translated into a state transition relation, defined by a Z predicate. The Z expressions in the labels are not changed. In a second step, the Z predicate is translated into the input language of a model checker. Besides simplifying the translation, this approach makes the translation of the Statechart into a state transition relation independent of the model checker used.

Büssow and Grieskamp [14] proposed a general scheme to integrate formalisms for the description of operational behavior into $\mu\mathcal{SZ}$. The approach allows to integrate any

formalism, if a translation into a Z state transition relation can be given. The translation defines the semantics of the integration. Z becomes the base formalism for the integration of various formalisms: The chosen approach is well supported by the $\mu\mathcal{SZ}$ language design and the Zeta system. There are three more benefits to this approach:

- Tools that can handle only Z such as a type checker or a theorem prover (e. g. Isabelle/Holz) can be used to analyze $\mu\mathcal{SZ}$ specifications with Statecharts without adaptations.
- If an automatic translation into a Z transition relation is provided, other languages for the description of operational behavior (e. g. message sequence charts) can be model checked.
- Pure Z specification, where the state transition relation is defined directly by a Z predicate, can be model checked, too.

The logic supported by a model checker is rather restricted. For common $\mu\mathcal{SZ}$ specifications, this is not sufficient. Therefore, more sophisticated Z expressions and data types are rewritten to simple ones that can be translated directly into the model checker input language.

The model checking support for $\mu\mathcal{SZ}$ comprises three tasks:

1. Translate Statecharts into a state transition relation, including the handling of racing and *assignment* semantics. This is described in Chapter 6 on page 71.

2. Rewrite Z expressions that are not directly handled by the model checker. This is described in Chapter 7 on page 97.

3. Translate the Z model into the model checker input language and perform model checking. This is a quite simple, syntactic translation. Thus, the model checker can be changed easily. The translation is described for the SMV model checker in Chapter 7 on page 142.

These steps are reflected by the architecture of the Zeta model checker adaptor. Figure 1.2 on the next page shows the complete tool chain as it is applied for model checking in the Zeta system.

1.5.2 Temporal Properties

Büssow and Grieskamp [13] developed the temporal logic DZ (*Dynamic Z*), for the abstract description of properties of an $\mu\mathcal{SZ}$ model. However, DZ is not supported by model checkers. As described in Chapter 4 on page 35, the model checking algorithms implemented by model checkers are specialized on a specific temporal logic. Moreover, there are no efficient model checking algorithms for every temporal logic. Therefore, a temporal logic that is implemented by a model checker is used instead of DZ.

Most model checkers, including SMV, support Computation Tree Logic (CTL, see 4.2 on page 37), proposed by Clarke and Emerson [17]. In section 4.3 on page 40

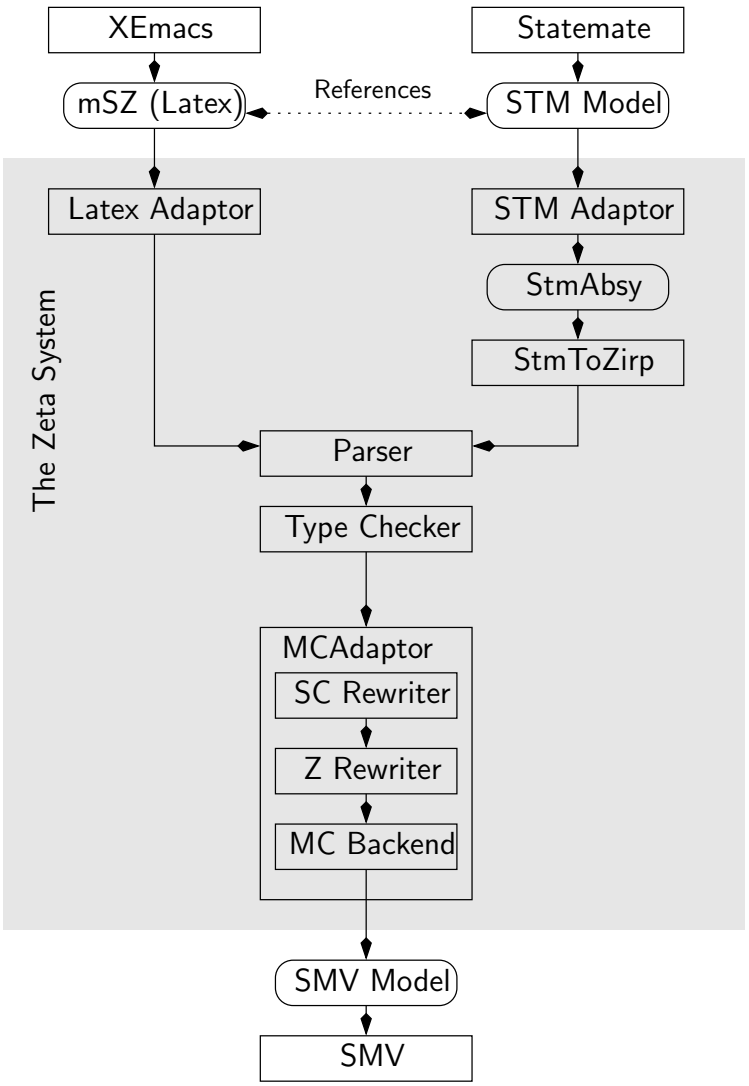


Figure 1.2: The complete model checking tool chain. The rounded boxes describe data types of the exchanged data. Zirp is exchanged, if rounded boxes are missing between two tools. The tools inside the Zeta system are adaptors.

it is shown how CTL can be defined for $\mu\mathcal{SZ}$. That means, CTL operators are defined by Z operators, the atomic (non-temporal) properties are specified with Z, and the underlying semantics is defined by the $\mu\mathcal{SZ}$ trace semantics.

1.6 Model Checking Statemate Statecharts

The *Statemate Extractor* translates not only Statemate Statecharts to $\mu\mathcal{SZ}$, but expressions and variable declarations as well. That way, it is possible to specify a model using Statemate only—without using $\mu\mathcal{SZ}$. Statemate enjoys a relevant acceptance in the industry, so already existing Statemate specifications can be checked with the presented approach as well as new ones. Thus, this work provides both: a model checker for $\mu\mathcal{SZ}$ and for Statemate.

Model checking is a quite promising technology for the verification of properties of control programs given by a finite model. Statemate is usually used for the specification of such control programs, and its models are usually finite. It is exactly this characteristic that makes Statemate models a well suited target for model checking.

Model checking Statecharts has been an issue for quite some time. The first approaches for automatic verification of properties, such as absence of racing, were built into the Statemate tool itself. However, this feature was abandoned later by i-Logix, since i-Logix was not able to keep up with the technological progress in the area of software verification and to provide the required quality for such a tool.

As one of the first ones, Day [21] translated Statemate Statecharts into the input language of the HOL-Voss tool and used the symbolic model checker of this tool for the verification of CTL properties.

Kelb [34] studied model checking of Statecharts and abstraction techniques. He also used symbolic model checking, but did not provide an automatic translation of Statecharts.

Mikk et. al. [41, 42] used the model checkers Spin/Promela and SMV for model checking Statecharts. They provided an abstract representation of Statemate Statecharts [39] from which models are translated into the respective input languages of Spin and SMV. Spin's input language Promela is a process algebraic, asynchronous language. This makes the translation of Statemate models rather complicated, since the synchronization of parallel Statecharts has to be done explicitly, as Mikk et. al. pointed out in [40]. Their approach was adopted by Hiemer [28] for the translation into CSP and verification with the FDR model checker. Hiemer verified the correctness of the translation into the process algebra.

The performance of Mikk's SMV translation was compared with the presented approach. As the target specification, a Statemate model of the production cell [34] was used. The model had 30 events, 92 Statechart states (including 30 super states). 158 CTL properties were shown. The model had $3\,10^{25}$ states, $3\,10^{6}$ of which were reachable. The results of the comparison are depicted in Table 1.1 on the next page. They show that the SMV model created by the approach presented here, verifies the specification twice as fast as Mikk's approach. It is assumed that the reason for this is the hierarchical translation scheme for Statecharts presented in section 6.3 on page 86, since this is the only major

	time/s	memory/MB	BDDs (total)	BDDs (transition relation)
Mikk	295	3.0	126,000	44,000
Büssow	136	2.3	88,000	11,000

The tests were performed on a 233 MHz Intel Pentium II with 96 MB RAM running Linux 2.0. They were executed with the following command line options: `smv -inc -f -i pc.ord pc.smv`, where `pc.smv` is a model, generated by one of two approaches and `pc.ord` is the respective variable order file.

Table 1.1: Comparing Model Checking Performance

difference to Mikk's approach.

As Mikk [42] points out, translation into a process algebra and verification with Spin is much slower than the SMV approach. Verification of a single property only took more than 1:30 hours.

Brockmeyer and Wittich [6, 5] use the Siemens symbolic model checker for the verification of Statemate models. They focus particularly on the verification of time requirements. For this, they apply a straightforward solution for Statemate timeouts that is also applied in the approach presented here, and a timed CTL for the specification of properties. *Timed CTL* is an extension of CTL by discrete time.

1.7 Model Checking Z

The results of this work can also be used to check pure Z specifications. However, the Z language provides no means to describe a model with reactive behavior. Therefore, reactive behavior of a Z specification cannot be verified. Z users usually work around this deficiency by informally declaring a predicate to be the state transition relation. If $\mu\mathcal{SZ}$ were used to declare the transition relation formally, the specification could also be model checked (if the model is finite).

An alternative approach of checking Z specifications via state enumeration is presented by Jackson, Jha, and Damon in [33]. Their Nitpick system checks specifications in the relational calculus NP, which is, roughly, a subset of Z.

- The Nitpick system automatically applies finite bounds to free types. Hereby, it is possible to check "infinte" specifications with free types, as well. The approach presented here, in contrary, assumes that finite bounds are applied to free types by hand. For example a declaration $x : \mathbb{Z}$ has to be rewritten to $x : 0 \,..\, \mathrm{MAXVAL}$. The Nitpick approach would be an interesting enhancement of the approach presented here.
- The Nitpick system uses its own model checking technique. It is based on removing isomorphs in relations. Isomorphs are combinations of variable assignments that all lead to the same valuation of the properties to show. Jackson, Jha and Damon report that they can search state spaces of 10^{12} in "less than a minute". As stated in the previous section, BDD based model checking can cope with much larger models. However, this is not a thorough comparison, since model checking techniques do not only depend on the size of the model.

- The major drawback of the Nitpick system is that it supports only relations and neither scalars nor sets. In particular, it does not support numbers. Thus, the approach presented here supports a much greater subset of Z.
- Nitpick language has no step semantics. Therefore, it is not compatible with Statechart semantics, and it is thus not suitable for a combination. Nor is it possible within Nitpick to use temporal logic to specify certain properties in an abstract way.

1.8 Main Features

The main features of the presented work are:

- Support of constants from axiomatic definitions. For example, if the model contains a constant $limit < 255$, the model is checked for all possible values of *limit* (This complies to the Z semantics of such constants).
- Support of invariants from declarations of data invariants. For example in the declaration of data variables a, b, c, a can be defined to be always b or c: $[a, b, c : 0 \,..\, 255 \mid a \in \{ b, c \}]$
- Support of timeouts. Timeouts that trigger firing of a transition are supported.
- An efficient and fast translation scheme for Statecharts is used.
- Declarations of data variables, data transitions and properties can be described with a rich (yet finite) subset of Z.
- Static analysis for racing is performed, and racing is supported with minimal overhead.
- Computation Tree Logic is defined for Z.
- It is integrated in the Zeta developing system and takes advantage of formating, type checking, Statemate Extraction.
- Derived, port, and data variables are supported.
- The overall architecture allows to exchange the model checker easily.
- The translators are implemented in 100% pure Java. Only the Statemate Extractor is platform dependend. The system has been tested under Linux, Windows NT and Solaris.
- Other formalisms for the description of behavioral aspects are supported, provided that they are translated into a Z state transition relation.

1.9 Notation

The mathematical theory presented in this work uses the Z notation. New features, of the recently accepted Z standard [49, 45] were used. These features are in particular: (1) Sections **section** that introduce separated name spaces to Z. (2) Schema expressions can be ordinary expressions. The theory was type checked using the Zeta tools.

1.10 Acknowledgements

I like to thank the entire Espress team for the often controversial but always fruitful discussions. In particular I thank the group at the TU Berlin: Hartmut Ehrig, Robert Geisler, Wolfgang Grieskamp, Stephan Herrmann, Stefan Jähnichen, Marcus Klar, Peter Pepper, and Matthias Weber. Most valuable were the discussions and close cooperation with Wolfgang during the project and later. I like to thank Stefan for supervising this work and waiting so long until I finally got it done and Peter for being the second supervisor. I like to thank Erich Mikk for the deep discussions on model checking Statecharts.

Special thanks go to Sonja Bonin and Jens-Christian Pastille for proof reading this work and many valuable hours.

Chapter 2

The Intelligent Cruise Control (ICC)

section *icc*

As one application of safety-critical systems, the Espress project focused on traffic engineernig. The Intelligent Cruise Control (ICC) was one of the two reference case studies in the Espress project. The ICC was independently developed by Daimler Benz during the project. It is now sold under the product name "Distronic". The ICC is presented here as an introductory example.

In addition to normal features of a cruise control, the intelligent cruise control has sensors to measure the distance to the traffic ahead, in order to adopt the speed accordingly. The cruise control knows different modes of operation:

- activated by the driver or not and
- adopting the speed requested by the driver or by the traffic ahead.

The latter depends on whether the distance sensor detects a car ahead that drives slower than the requested speed. Here, the *mode control* is defined. It is a part of the intelligent cruise control (ICC). The mode control keeps track of the ICC's current mode of operation (e. g. if it is activated or not) and computes the nominal speed for the ICC.

2.1 Definitions

In order to be able to model check the specification, *SPEED*, *ACCELERATION* and *LENGTH* are defined as bounded integers and the constants *stepSpeed*, *minSpeed*, *maxSpeed* and *safeDistance* are assigned concrete values. The original specification was done without having model checking in mind, and these were the only adaptions needed for model checking.

ICC

$MAXINT == 255$
$stepSpeed == 2$
$minSpeed == 30$
$maxSpeed == 150$
$safeDistance == 100$

$SPEED == 0 .. MAXINT$
$ACCELERATION == 0 .. MAXINT$
$LENGTH == 0 .. MAXINT$

2.2 Interfaces

The ICC has several interfaces to communicate with its environment. It reads the current position of the lever (*Lever*) and the brake (*Pedal*), the current speed and acceleration (*Movement*), and the distance and speed of a car ahead (*Ahead*). If there is no traffic ahead, *distAhead* is zero. The only output of the system is the nominal speed of the car (*NominalSpeed*) that has to be computed.

The input values (*Lever*, *Pedal*, *Movement*, and *Ahead*) are provided by the environment of the controller. This environment remains beyond the scope of the specification. Therefore, concrete values for the input variables are unknown. The model checker handles such input variables in verifying the properties for all possible combinations of input variables.

$\mu\mathcal{SZ}$, however, allows to specify invariants for the input variables. This is done in the schema *Ahead*: $distAhead > 0 \Leftrightarrow accelAhead > 0$. The translation scheme presented here takes such invariants into account. Many properties only hold if the environment fulfills certain requirements. You cannot verify these properties if you cannot specify reasonable assumptions about the environment.

ICC

$LEVERPOS ::= increase \mid decrease \mid resume \mid off \mid idle$

PORT Lever

$lever : LEVERPOS$

PORT Pedal

$brake$: signal

PORT Movement

$curSpeed : SPEED$
$curAccel : ACCELERATION$

PORT Ahead

$distAhead : LENGTH$
$accelAhead : ACCELERATION$

$distAhead > 0 \Leftrightarrow accelAhead > 0$

PORT NominalSpeed

$nominalSpeed : SPEED$

INIT NominalSpeedInit

$NominalSpeed$

$nominalSpeed = 0$

2.3 Internal Data

Internally, the ICC keeps the speed requested by the driver. *nominalSpeed*, the output of the ICC, is set to this speed, if the ICC is activated and no car is driving ahead.

ICC

DATA RequestedSpeed

$requestedSpeed : SPEED$

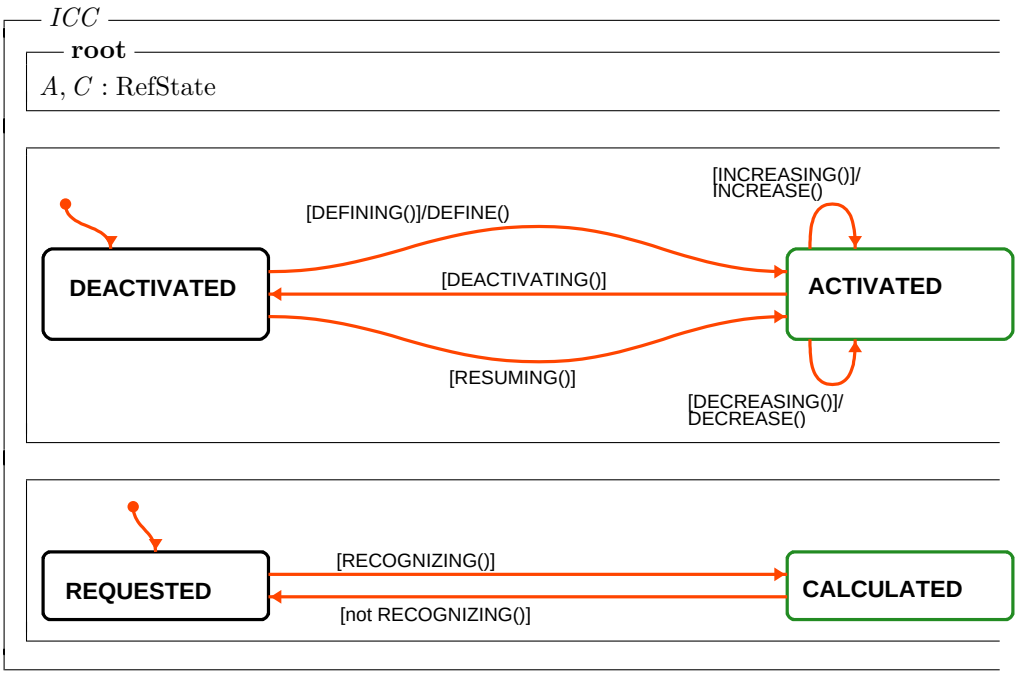
INIT RequestedSpeedInit

$RequestedSpeed$

$requestedSpeed = 0$

2.4 Behavior

The behavior of the ICC is defined by two parallel Statecharts A and C. The charts are references to external Statemate models.



2.5 Guards and Operations

The guards and operations of the Statecharts are defined in Z as follows.

ICC

GUARD DEFINING

$Lever;\ Pedal;\ Movement$

$lever = increase \lor lever = decrease$
$\neg\ \mathsf{df}\ brake$
$minSpeed \leq curSpeed \leq maxSpeed$

OP DEFINE

$Movement;\ \Delta RequestedSpeed$

$requestedSpeed' = curSpeed$

GUARD INCREASING

$Lever;\ Pedal;\ RequestedSpeed$

$lever = increase \land \neg\ \mathsf{df}\ brake$

OP INCREASE

$\Delta RequestedSpeed$

$requestedSpeed' =$
if $requestedSpeed + stepSpeed \leq maxSpeed$
then $requestedSpeed + stepSpeed$ **else** $requestedSpeed$

GUARD DECREASING

$Lever;\ Pedal;\ RequestedSpeed$

$lever = decrease \wedge \neg\ \mathsf{df}\ brake$

OP DECREASE

$\Delta RequestedSpeed$

$requestedSpeed' =$
if $requestedSpeed - stepSpeed > minSpeed$
then $requestedSpeed - stepSpeed$
else $requestedSpeed$

GUARD RESUMING

$Lever;\ Pedal;\ Movement;\ RequestedSpeed$

$lever = resume$
$requestedSpeed > 0$
$\neg\ \mathsf{df}\ brake$
$minSpeed \leq curSpeed \leq maxSpeed$

GUARD DEACTIVATING

$Lever;\ Pedal;\ Movement;\ RequestedSpeed$

$lever = off$
$\vee\ \mathsf{df}\ brake$
$\vee\ (curSpeed < minSpeed \wedge$
$requestedSpeed > curSpeed)$

GUARD RECOGNIZING

$RequestedSpeed;\ Ahead;\ Movement$

$0 < distAhead \wedge distAhead < safeDistance$

The nominal speed is defined by an invariant, depending on the Statechart states and the speed requested by the driver.

ICC

DATA SetSpeed

$Lever;\ Pedal;\ NominalSpeed;\ RequestedSpeed;$ Control

$\mathsf{instate}\ A{:}ACTIVATED \wedge \mathsf{instate}\ C{:}REQUESTED \Rightarrow$
$nominalSpeed = requestedSpeed$
$\mathsf{instate}\ A{:}ACTIVATED \wedge \mathsf{instate}\ C{:}CALCULATED \Rightarrow$
$minSpeed \leq nominalSpeed \leq maxSpeed$
$\mathsf{instate}\ A{:}DEACTIVATED \Rightarrow nominalSpeed = 0$

2.6 Safety

The schema *Safety* specifies the security properties for the above specification (*Safety*). The first affirms that the nominal speed is in the range the ICC is allowed to take control of ($minSpeed \leq nominalSpeed \leq maxSpeed$). The second states that the ICC may only take control after it was activated by the driver. Note that in order to make this property true, we have a fairness (*Fairness*) constraint, ensuring that the ICC is activated eventually. The third property states that if the driver pushes the brake (df *brake*) the cruise control will switch itself off immediately.

ICC

Fairness

Lever

$lever = increase$

Regulating

DATA

$nominalSpeed > 0$

Safety

INIT
CTL[DATA]

$\mathcal{AG}([\text{DATA} \mid minSpeed \leq nominalSpeed \leq maxSpeed \vee nominalSpeed = 0])$
$\mathcal{AG}((\neg\ Regulating) \Rightarrow$
$\quad \mathcal{A}(\neg\ Regulating)\mathcal{U}$
$\quad\quad [\text{DATA} \mid lever = increase \vee lever = decrease \vee lever = resume])$
$\mathcal{AG}([\text{DATA} \mid \mathsf{df}\ brake] \Rightarrow \mathcal{AX}(\neg\ Regulating))$

2.7 Model Checking the ICC

Figure 2.1 on the next page shows a Zeta session which model checks the ICC specification. The Zeta system performs the necessary translation for model checking. For this, it executes a tool-chain consisting of the type-checker, the Statemate adaptor and the SMV translator. The output of the translation can be seen in the upper part of the window. In the lower part, the SMV model checker is executed. The box shows the actual SMV output. The SMV model checker needs approximately 1.3 seconds (on an Intel PII 233 MHz) to verify the three properties.

2.8 Annotated SMV Listing

In order to give an impression of what the translation presented in the following chapter, does, the generated SMV program is presented. The code is roughly explained. The SMV syntax is explained in Chapter 8 on page 137.

Variables to store the active Statechart states (the *configuration*) are declared. One variable is declared for each xor-state. The representation of the Statechart configuration is discussed in section 6.1.5 on page 76.

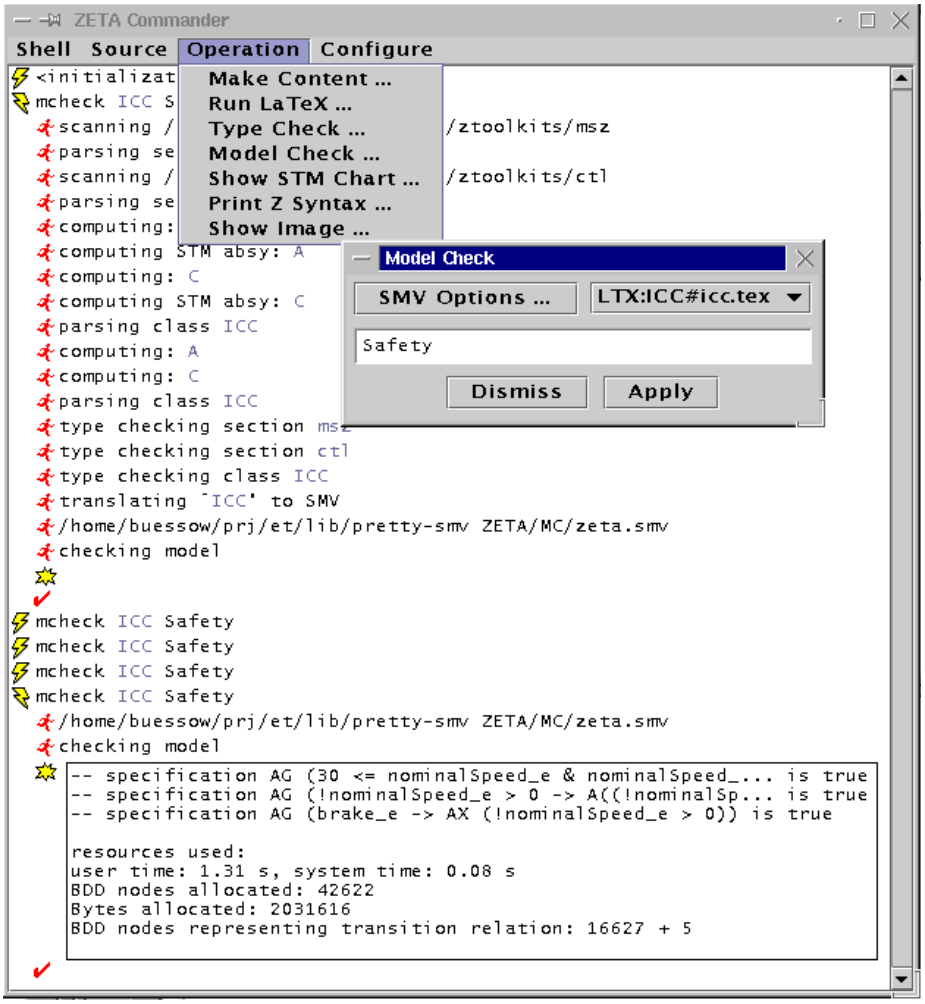


Figure 2.1: Model checking the ICC with the Zeta toolkit

```
4   VAR
5       A_a : { A_ACTIVATED_s,  A_DEACTIVATED_s };
6       C_a : { C_CALCULATED_s,  C_REQUESTED_s };
```

The data variables are declared straightforwardly.

```
11  VAR
12      curAccel_e : 0..255;
13      requestedSpeed_e : 0..255;
14      nominalSpeed_e : 0..255;
15      curSpeed_e : 0..255;
16      distAhead_e : 0..255;
17      accelAhead_e : 0..255;
18      lever_e : { increase,  decrease,  resume,  off,  idle };
19      brake_e : boolean;
```

In an operational specification language such as Statecharts, one would expect that variables keep their value until they are assigned a new one (*assignment semantics/persistency of variables*). For the translation into a state transition relation, extra meaures have to be taken, in order to ensure this. For the variable *requestedSpeed*, an auxiliary variable is introduced. It is zero in a step, if *requestedSpeed* is not written and should preserve its value. The mechanism for handling persistent variables and racing, which is used here, is presented in section 6.2.3 on page 79.

```
22  VAR
23    requestedSpeed_l : 0..1;
```

The SMV init predicate consists of the predicates of the init schemata and the invariants of the data and port schemata. The invariants have to hold in the initial state, too, and this has to be ensured explicitly. Therefore, the invariants are added to the init-predicate. The presented predicates origin form the schemata: *Ahead*, *NominalSpeed*, and *RequestedSpeedInit*. *SetSpeed* is omitted.

```
25  INIT
26    (((distAhead_e > 0) <-> (accelAhead_e > 0)) &
27    nominalSpeed_e=0 &
28    requestedSpeed_e=0 &
29    ...
```

The initial configuration of the Statechart:

```
41  INIT
42    (A_a=A_DEACTIVATED_s & C_a=C_REQUESTED_s)
```

In order to increase readability, the translator generates abbreviations for the Statechart translation. For each transition, three abbreviations are generated: `grd_`i, `lck_`i, and `act_`i. The transition's guard is represented by `grd_`i and the action by `act_`i. They correspond to the function *transGuard* and *trFullAct* introduced in section 6.3 on page 86. `lck_`i corresponds to *protectOtherPlaces* also introduced in section 6.3 on page 86. The transitions are numbered subsequently. Transition 1 represents *DEFINE*/*DEFINING*, 2 *INCREASING*/*INCREASE*, etc.

```
46  DEFINE
47    grd_1 := (A_a=A_DEACTIVATED_s &
48              (lever_e=resume & (requestedSpeed_e > 0) & !(brake_e) &
49              ((30 <= curSpeed_e) & (curSpeed_e <= 150))));
50    lck_1 := !(next(requestedSpeed_l)=1);
51    act_1 := (grd_1 & (lck_1 & next(A_a)=A_ACTIVATED_s));
52    grd_2 := (A_a=A_ACTIVATED_s & (lever_e=increase & !(brake_e)));
53    lck_2 := 1;
54    act_2 := (grd_2 &
55              (lck_2 &
56              (next(A_a)=A_ACTIVATED_s &
57              (!(next(requestedSpeed_l)=0) &
58              (next(requestedSpeed_l)=1 ->
59              (next(requestedSpeed_l)=1 &
60              next(requestedSpeed_e)=case
61                                      ((requestedSpeed_e + 2) <= 150) :
62                                        (requestedSpeed_e + 2);
63                                      1 : requestedSpeed_e;
64                                    esac))))));
```

For each state, a transition relation is defined as an abbreviation. For an xor-state the transition relation executes a transition if one of the state's guards is true. Otherwise, it remains in the same state and does not write any variable. The definition of the transition relation corresponds to the function *stateTrans*.

```
93    trans_A := case
94                 (grd_1 | grd_2 | grd_3 | grd_4 | grd_5) :
95                   (act_1 | act_2 | act_3 | act_4 | act_5);
96                 1 :
97                   (next(A_a)=A_a &
98                   case
99                     1 : !(next(requestedSpeed_l)=1);
100                  esac);
101              esac;
```

A transition relation is defined for the root state similar to the one of state A. Since the root state is an and-state, its transition relation is the conjunction of its sub-states.

```
122     trans_root := (trans_A & trans_C);
```

The root state's transition relation is added to the program's transition relation.

```
126   TRANS
127     trans_root
```

This predicate ascertains that the persistent variable *requestedSpeed* keeps its value as long as it is not written.

```
130   TRANS
131     (next(requestedSpeed_l)=0 -> next(requestedSpeed_e)=requestedSpeed_e)
```

In addition to the init predicate, the invariants of the schemata *Ahead* and *SetSpeed* are also added to the transition relation.

```
134   TRANS
135     (((next(distAhead_e) > 0) <-> (next(accelAhead_e) > 0)) &
136     ((next(A_a)=A_ACTIVATED_s & next(C_a)=C_REQUESTED_s) ->
137     next(nominalSpeed_e)=next(requestedSpeed_e)) &
138     ...
```

The fairness schema is added as a fairness constraint.

```
149   FAIRNESS
150     lever_e=increase
```

Finally the properties to be shown are added.

```
152   SPEC
153     AG((((30 <= nominalSpeed_e) & (nominalSpeed_e <= 150)) | nominalSpeed_e=0))
154
155   SPEC
156     AG((!((nominalSpeed_e > 0)) ->
157        A[!((nominalSpeed_e > 0)) U ((lever_e=increase | lever_e=decrease) |
158                                     lever_e=resume)]))
159
160   SPEC
161     AG((brake_e -> AX(!((nominalSpeed_e > 0)))))
```

2.9 Complete SMV Listing

The following presents the complete SMV program as it is produced by the translator. The pretty printing is done by the Zeta system's pretty printer.

```
1   MODULE
2   main
3
4   VAR
5       A_a :
6         { A_ACTIVATED_s,  A_DEACTIVATED_s };
7       C_a :
8         { C_CALCULATED_s,  C_REQUESTED_s };
9
10  -- VARIABLES
11  VAR
12      curAccel_e : 0..255;
13      requestedSpeed_e : 0..255;
14      nominalSpeed_e : 0..255;
15      curSpeed_e : 0..255;
16      distAhead_e : 0..255;
17      accelAhead_e : 0..255;
18      lever_e : { increase,  decrease,  resume,  off,  idle };
19      brake_e : boolean;
20
21  -- LOCK VARIABLES
22  VAR
23    requestedSpeed_l : 0..1;
24
25  INIT
26    (((distAhead_e > 0) <-> (accelAhead_e > 0)) & nominalSpeed_e=0 &
27    requestedSpeed_e=0 &
28    ((A_a=A_ACTIVATED_s & C_a=C_REQUESTED_s) -> nominalSpeed_e=requestedSpeed_e
29    ) &
30    ((A_a=A_ACTIVATED_s & C_a=C_CALCULATED_s) ->
31    ((30 <= nominalSpeed_e) & (nominalSpeed_e <= 150))) &
32    (A_a=A_DEACTIVATED_s -> nominalSpeed_e=0) &
33    ((distAhead_e > 0) <-> (accelAhead_e > 0)) &
34    ((A_a=A_ACTIVATED_s & C_a=C_REQUESTED_s) -> nominalSpeed_e=requestedSpeed_e
35    ) &
36    ((A_a=A_ACTIVATED_s & C_a=C_CALCULATED_s) ->
37    ((30 <= nominalSpeed_e) & (nominalSpeed_e <= 150))) &
38    (A_a=A_DEACTIVATED_s -> nominalSpeed_e=0))
39
40  -- INITIAL CONFIGURATION
41  INIT
42    (A_a=A_DEACTIVATED_s & C_a=C_REQUESTED_s)
43
44  -- DEFINITIONS FOR STATE A
45  DEFINE
46    grd_1 := (A_a=A_DEACTIVATED_s &
47              (lever_e=resume & (requestedSpeed_e > 0) & !(brake_e) &
48              ((30 <= curSpeed_e) & (curSpeed_e <= 150))));
49    lck_1 := !(next(requestedSpeed_l)=1);
50    act_1 := (grd_1 & (lck_1 & next(A_a)=A_ACTIVATED_s));
51    grd_2 := (A_a=A_ACTIVATED_s & (lever_e=increase & !(brake_e)));
52    lck_2 := 1;
53    act_2 := (grd_2 &
54              (lck_2 &
55              (next(A_a)=A_ACTIVATED_s &
56              (!(next(requestedSpeed_l)=0) &
```

```
57            (next(requestedSpeed_l)=1 ->
58            (next(requestedSpeed_l)=1 &
59            next(requestedSpeed_e)=case
60                                  ((requestedSpeed_e + 2) <= 150) :
61                                    (requestedSpeed_e + 2);
62                                  1 : requestedSpeed_e;
63                                esac))))));
64    grd_3 := (A_a=A_DEACTIVATED_s &
65            ((lever_e=increase | lever_e=decrease) & !(brake_e) &
66            ((30 <= curSpeed_e) & (curSpeed_e <= 150))));
67    lck_3 := 1;
68    act_3 := (grd_3 &
69            (lck_3 &
70            (next(A_a)=A_ACTIVATED_s &
71            (!(next(requestedSpeed_l)=0) &
72            (next(requestedSpeed_l)=1 ->
73            (next(requestedSpeed_l)=1 & next(requestedSpeed_e)=curSpeed_e)))))
74            );
75    grd_4 := (A_a=A_ACTIVATED_s &
76            ((lever_e=off | brake_e) |
77            ((curSpeed_e < 30) & (requestedSpeed_e > curSpeed_e))));
78    lck_4 := !(next(requestedSpeed_l)=1);
79    act_4 := (grd_4 & (lck_4 & next(A_a)=A_DEACTIVATED_s));
80    grd_5 := (A_a=A_ACTIVATED_s & (lever_e=decrease & !(brake_e)));
81    lck_5 := 1;
82    act_5 := (grd_5 &
83            (lck_5 &
84            (next(A_a)=A_ACTIVATED_s &
85            (!(next(requestedSpeed_l)=0) &
86            (next(requestedSpeed_l)=1 ->
87            (next(requestedSpeed_l)=1 &
88            next(requestedSpeed_e)=case
89                                  ((requestedSpeed_e - 2) > 30) :
90                                    (requestedSpeed_e - 2);
91                                  1 : requestedSpeed_e;
92                                esac))))));
93    trans_A := case
94                 (grd_1 | grd_2 | grd_3 | grd_4 | grd_5) :
95                   (act_1 | act_2 | act_3 | act_4 | act_5);
96                 1 :
97                   (next(A_a)=A_a &
98                   case
99                     1 : !(next(requestedSpeed_l)=1);
100                  esac);
101             esac;
102
103   -- DEFINITIONS FOR STATE C
104   DEFINE
105     grd_6 := (C_a=C_REQUESTED_s & ((0 < distAhead_e) & (distAhead_e < 100)));
106     lck_6 := 1;
107     act_6 := (grd_6 & (lck_6 & next(C_a)=C_CALCULATED_s));
108     grd_7 := (C_a=C_CALCULATED_s & !(((0 < distAhead_e) & (distAhead_e < 100)))
109             );
110     lck_7 := 1;
111     act_7 := (grd_7 & (lck_7 & next(C_a)=C_REQUESTED_s));
112     trans_C := case
113                  (grd_6 | grd_7) : (act_6 | act_7);
114                  1 :
115                    (next(C_a)=C_a &
116                    case
117                      1 : 1;
```

```
118                 esac);
119             esac;
120
121 -- DEFINITIONS FOR STATE \root
122   trans_root := (trans_A & trans_C);
123
124
125 -- THE TRANSITION RELATION
126 TRANS
127 trans_root
128
129 -- DEFAULT BEHAVIOR
130 TRANS
131   (next(requestedSpeed_l)=0 -> next(requestedSpeed_e)=requestedSpeed_e)
132
133 -- INVARIANTS
134 TRANS
135   (((next(distAhead_e) > 0) <-> (next(accelAhead_e) > 0)) &
136   ((next(A_a)=A_ACTIVATED_s & next(C_a)=C_REQUESTED_s) ->
137   next(nominalSpeed_e)=next(requestedSpeed_e)) &
138   ((next(A_a)=A_ACTIVATED_s & next(C_a)=C_CALCULATED_s) ->
139   ((30 <= next(nominalSpeed_e)) & (next(nominalSpeed_e) <= 150))) &
140   (next(A_a)=A_DEACTIVATED_s -> next(nominalSpeed_e)=0) &
141   ((next(distAhead_e) > 0) <-> (next(accelAhead_e) > 0)) &
142   ((next(A_a)=A_ACTIVATED_s & next(C_a)=C_REQUESTED_s) ->
143   next(nominalSpeed_e)=next(requestedSpeed_e)) &
144   ((next(A_a)=A_ACTIVATED_s & next(C_a)=C_CALCULATED_s) ->
145   ((30 <= next(nominalSpeed_e)) & (next(nominalSpeed_e) <= 150))) &
146   (next(A_a)=A_DEACTIVATED_s -> next(nominalSpeed_e)=0))
147
148 -- FAIRNESS CONSTRAINTS
149 FAIRNESS
150   lever_e=increase
151
152 SPEC
153   AG((((30 <= nominalSpeed_e) & (nominalSpeed_e <= 150)) | nominalSpeed_e=0))
154
155 SPEC
156   AG((!((nominalSpeed_e > 0)) ->
157      A[!((nominalSpeed_e > 0)) U ((lever_e=increase | lever_e=decrease) |
158                                   lever_e=resume)]))
159
160 SPEC
161   AG((brake_e -> AX(!((nominalSpeed_e > 0)))))
```

Chapter 3

Mathematical Definitions

section *Aux*

Z and the Z mathematical toolkit are used for the mathematical parts of this work. The toolkit is extended by some more definitions, which are given in this chapter.

3.1 Functions and Sequences

Functions for "currying" functions:

$$[X, Y, Z]$$

$$curryFstOf2 : (X \times Y \leftrightarrow Z) \to (X \leftrightarrow (Y \leftrightarrow Z))$$
$$currySndOf2 : (X \times Y \leftrightarrow Z) \to (Y \leftrightarrow (X \leftrightarrow Z))$$

$$\begin{aligned}&\forall f : X \times Y \leftrightarrow Z;\ x : X;\ y : Y \bullet \\ &\quad curryFstOf2\, f\, x = ((\{ x \} \times Y) \lhd f) \circ (second^{\sim}) \land \\ &\quad currySndOf2\, f\, y = ((X \times \{ y \}) \lhd f) \circ (first^{\sim})\end{aligned}$$

The zip function combines two sequences of equal length: $\mathsf{zip}(\langle a, b \rangle, \langle b, c \rangle) = \langle (a, b), (b, c) \rangle$.

$$[X, Y]$$

$$\mathsf{zip} : \operatorname{seq} X \times \operatorname{seq} Y \pfun \operatorname{seq}(X \times Y)$$

$$\mathsf{zip} = \lambda\, xs : \operatorname{seq} X;\ ys : \operatorname{seq} Y \mid \operatorname{dom} xs = \operatorname{dom} ys \bullet (\lambda\, i : \operatorname{dom} xs \bullet xs\ i \mapsto ys\ i)$$

reduce reduces a sequence from left to right with a given functions: $\mathsf{reduce}\,(\_+\_)\,\langle 1, 3, 5 \rangle = (1 + 3) + 5$

$$[X]$$

$$\mathsf{reduce} : (X \times X \to X) \to \operatorname{seq}_1 X \to X$$

$$\begin{aligned}&\forall f : X \times X \to X;\ xs : \operatorname{seq} X;\ x : X \bullet \\ &\quad \mathsf{reduce}\, f\, \langle x \rangle = x \land \\ &\quad \mathsf{reduce}\, f\, (xs \frown \langle x \rangle) = f(x, \mathsf{reduce}\, f\, xs)\end{aligned}$$

setreduce reduces a set in an arbitrary order.

$$[X]$$

$$\mathsf{setreduce} : (X \times X \to X) \to \mathbb{P}_1\, X \to X$$

$$\begin{aligned}
&\forall f : X \times X \to X;\ xp : \mathbb{P}_1\, X;\ x : X \mid x \notin xp \bullet \\
&\quad \mathsf{setreduce}\, f\, \{\, x \,\} = x \land \\
&\quad \mathsf{setreduce}\, f\, (\{\, x \,\} \cup xp) = f(x, \mathsf{setreduce}\, f\, xp)
\end{aligned}$$

3.2 Implode and Explode

It is often necessary to translate a sequence of sets into a set of sequences and vice versa. Consider for example the following problem. Let $\vec{\tau}$ be a sequence of types ($\vec{\tau} = \langle \tau_1, \ldots, \tau_n \rangle$) and $values : Type \to \mathbb{P}\ Value$ a function, mapping types to their possible values ($values = \{\, \tau_1 \mapsto \{\, a, b, c \,\}, \ldots, \tau_n \mapsto \{\, x, y, z \,\} \,\}$. Now, all sequences of possible values are searched: $\{\, \langle a, \ldots, x \rangle, \langle a, \ldots, y \rangle, \ldots \,\}$ Formally: $\{\, vs : \mathrm{seq}\ Value \mid \mathsf{dom}\ vs = \mathsf{dom}\ \vec{\tau} \land \forall\, i : \mathsf{dom}\ vs \bullet vs\ i \in values(\vec{\tau}\ i) \,\}$.

This problem can be expressed by explode. $values \circ \vec{\tau}$ is the sequence of possible values, and explode($values \circ \vec{\tau}$) translates this sequence into a set of sequences of values. A sequence of two sets $\{\, a, b \,\}$ and $\{\, b, c \,\}$ is translated as follows:

$$\mathsf{explode} \langle \{\, a, b \,\}, \{\, b, c \,\} \rangle = \{\, \langle\, a, b \,\rangle, \langle\, a, c \,\rangle, \langle\, b, b \,\rangle, \langle\, b, c \,\rangle \,\}$$

explode is defined over an arbitrary index set X and a range set Y. The range sets must not be empty ($\mathbb{P}_1\ Y$).

The reverse function of explode is implode. Note that implode is not a total function. It translates only sets of functions with equal domains, e. g. implode$\{\, \langle\, a, b \,\rangle, \langle\, b, c, d \,\rangle$ is not defined.

explode and implode are usually used in conjunction with sequences and function mappings ($\circ$).

$$[X, Y]$$

$$\begin{aligned}
&\mathsf{explode} : (X \pfun \mathbb{P}_1\ Y) \rightarrowtail \mathbb{P}(X \pfun Y) \\
&\mathsf{implode} : \mathbb{P}(X \pfun Y) \pfun (X \pfun \mathbb{P}_1\ Y)
\end{aligned}$$

$$\begin{aligned}
&\forall f : X \pfun \mathbb{P}_1\ Y \bullet \\
&\quad \mathsf{explode}\, f = \{\, g : X \pfun Y \mid \mathsf{dom}\, f = \mathsf{dom}\, g \land (\forall\, x : \mathsf{dom}\, f \bullet g\ x \in f\ x\,)\} \\
&\mathsf{implode} = \mathsf{explode}^{\sim}
\end{aligned}$$

3.3 Fixed Points

Z provides least fixed point declarations with its free types. The free type construction has the disadvantage that it can only be used to introduce an new type, not to define a subset of an existing type. Therefore, a new operator is defined. Firstly, some fix point definitions are provided (according to Davey and Priestley [20]).

function 200 leftassoc ($\_ \sqsubseteq \_$)

A relation $(\_ \sqsubseteq \_)$ is called a *partial order*, if it is reflexive and antisymmetric ($(\_ \sqsubseteq \_) \cap (\_ \sqsubseteq \_)^\sim = \mathsf{id}[X]$ and transitive $(\_ \sqsubseteq \_)^+ \subseteq (\_ \sqsubseteq \_)$ }. For some set X, *porder* is the set of all partial orders.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
porder : \mathbb{P}(X \leftrightarrow X) \\
\hline
porder = \{ \_ \sqsubseteq \_ : X \leftrightarrow X \mid ((\_ \sqsubseteq \_) \cap (\_ \sqsubseteq \_)^\sim) = \mathsf{id}[X] \land (\_ \sqsubseteq \_)^+ \subseteq (\_ \sqsubseteq \_) \} \\
\hline
\end{array}
$$

Fix-points, least fix-points, and greatest fix-points.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
\mathsf{fix} : (X \to X) \to \mathbb{P}\, X \\
\mathsf{lfix}, \mathsf{gfix} : porder[X] \to (X \to X) \nrightarrow X \\
\hline
\forall f : X \to X \bullet \mathsf{fix} f = \{ x : X \mid x = f\, x \} \\
\forall \_ \sqsubseteq \_ : porder[X];\ f : X \to X \bullet \\
\quad (\mathsf{lfix}\, (\_ \sqsubseteq \_)\, f) \in \mathsf{fix} f \land \\
\quad (\forall x : \mathsf{fix} f \bullet ((\mathsf{lfix}\, (\_ \sqsubseteq \_)\, f), x) \in (\_ \sqsubseteq \_)) \\
\forall \_ \sqsubseteq \_ : porder[X];\ f : X \to X \bullet \\
\quad (\mathsf{gfix}\, (\_ \sqsubseteq \_)\, f) \in \mathsf{fix} f \land \\
\quad (\forall x : \mathsf{fix} f \bullet (x, (\mathsf{gfix}\, (\_ \sqsubseteq \_)\, f)) \in (\_ \sqsubseteq \_)) \\
\hline
\end{array}
$$

The fix-point operators are used to construct smallest sets. Consider for example the definition of the set *Conj* that represents only conjunctions of equality predicates. The predicate type *Pred* is already declared. Assume that there are, among others, constructors for equality $(\_ = \_)$ and conjunction $(\_ \land \_)$.

Conj is defined to be the smallest set the following axioms hold for:

1. $\forall e_1, e_2 : Expr \bullet (e_1 = e_2) \in Conj$ (or $(\_ = \_)(\!| Expr \times Expr |\!) \subseteq Conj$)
2. $\forall p_1, p_2 : Conj \bullet (p_1 \land p_2) \in Conj$ (or $(\_ \land \_)(\!| Conj \times Conj |\!) \subseteq Conj$)

This can be expressed as the least fix-point (one has to warrant that the lambda abstraction is continuous):

$$
\begin{aligned}
Conj = \mathsf{lfix}(\_ \subseteq \_), (\lambda X : \mathbb{P}\, Pred \bullet\ & (\_ = \_)(\!| Expr \times Expr |\!) \cup \\
& (\_ \land \_)(\!| Conj \times Conj |\!))
\end{aligned}
$$

For more convenience, the `\fixmax` macro is introduced. The term

$$
\begin{array}{l}
X_c \overset{X}{\leftarrow} E_1 \\
\quad \| \ldots \\
\quad \| E_n
\end{array}
$$

is translated to $X_c = \mathsf{lfix}(\_ \subseteq \_)(\lambda X_c : \mathbb{P}\, X \bullet E_1 \cup \ldots \cup E_n)$. Using this macro, the definition can be rewritten:

$$
\begin{array}{l}
Conj \overset{Pred}{\leftarrow} (\_ = \_)(\!| Expr \times Expr |\!) \\
\quad \| \ (\_ \land \_)(\!| Conj \times Conj |\!)
\end{array}
$$

In order to define two sets that are mutually dependent, a subset relation for tuples of sets is defined:

$$
\begin{array}{|l}
\hline
[X, Y] \\
\subseteq_2 : ((\mathbb{P}\, X) \times \mathbb{P}\, Y) \leftrightarrow ((\mathbb{P}\, X) \times \mathbb{P}\, Y) \\
\hline
\forall\, X_1, X_2 : \mathbb{P}\, X;\ Y_1, Y_2 : \mathbb{P}\, Y \bullet \\
\quad (X_1, Y_1) \mapsto (X_2, Y_2) \in \subseteq_2 \Leftrightarrow X_1 \subseteq X_2 \land Y_1 \subseteq Y_2 \\
\subseteq_2 \in \mathit{porder} \\
\hline
\end{array}
$$

The term $X_c \leftarrow_X E_{X1} \parallel \ldots \parallel E_{Xn} \oplus Y_c \leftarrow_Y E_{Y1} \parallel \ldots \parallel E_{Ym}$ is translated to $(X_c, Y_c) = \mathsf{lfix}(\_ \subseteq_2 \_)(\lambda\, X_c : \mathbb{P}\, X;\ Y_c : \mathbb{P}\, Y \bullet E_{X1} \cup \ldots \cup E_{Xn} \mapsto E_{Y1} \cup \ldots \cup E_{Y_m})$.

The fix point functions are used and typeset with the following macros:

```
%%macro \fixmore 0 \cup
%%macro \fixmac 3  (#1 = \lfix(\_\subseteq\_)(\lambda #1 : \power #2 @ #3))
%%macro \fixmacc 6
  ((#1,#3) = \lfix(\subseteqtuple)
             (\lambda #1 : \power #2; #3 : \power #4 @ (#3) \mapsto (#6)))
```

3.4 Sum and Product

The well known sum (Σ) and product (Π) are defined for sequences of numbers.

$$
\begin{array}{|l}
\hline
\Sigma, \Pi : \mathrm{seq}\, \mathbb{Z} \to \mathbb{Z} \\
\hline
\Sigma = \mathsf{reduce}(\_ + \_) \oplus \{\, \varnothing \mapsto 0 \,\} \\
\Pi = \mathsf{reduce}(\_ * \_) \oplus \{\, \varnothing \mapsto 1 \,\} \\
\hline
\end{array}
$$

3.5 Macros for Type and Rewriting Rules

For better presentation, the type and rewriting rules are formatted with special `\rule` macro. The macro is defined as follows:

```
%%macro \rule 3 \forall #3 @ #1 \implies #2
```

As a TEX macro `\rule` produces the following:

$$\forall\, \#3 \bullet \frac{\#1}{\#2}$$

Chapter 4

Model Checking

This section describes the basic concepts and theories used for model checking and, in particular, symbolic model checking. It describes

- *Kripke structures* for the representation of the model to be checked,
- *Computation Tree Logic*, for the formulation of properties of the model,
- the model checking algorithm that computes the validity of a formula for a given model,
- the usage of *ordered Binary Decision Diagrams BDDs*, to represent sets of states and compute Boolean operators efficiently,
- how the presented concepts and theories can be used to check properties of $\mu\mathcal{SZ}$ classes, and
- other model checking techniques.

4.1 Kripke Structures

section *ModelChecking* **parents** *Aux*

The technique of *model checking* for the verification of properties of finite models was proposed by Emerson and Clarke [17] and independently by Quielle and Sifakis [47]. In this work model checking is used for the verification of properties of a $\mu\mathcal{SZ}$ or Statemate specification. The theory presented in this Chapter is implemented by the model checker used (i. e. McMillan's [38] SMV symbolic model checker).

The basic principle of model checking is that properties of a system, defined by a finite model, are proven by complete enumeration of the system's states (its *state space*) and the possible traces. The model represents the system by a Kripke structure.[1] A Kripke structure is a triple (S, R, L), where S is a set of states, $R \subseteq S \times S$, is the state transition

[1]The term *Kripke structure* is taken from modal logic. The use of these structures, for defining the validity of modal logic formulae, derives from the work of Saul Kripke (see [31]). The essence of this thesis is, in fact, to translate an $\mu\mathcal{SZ}$ (or Statemate) specification into a Kripke structure, using a representation accepted by the model checker.

relation, and L is a valuation function. The valuation function defines in which states a primitive (i. e. non-modal) property is true.

For some system, for example, a state may consist of the values of the system's variables and its program counter. For such a system, the valuation function L would be defined in such a way that a property, $P \equiv a = 4$ is true for all states s of the system (i. e. $P \in L(s)$) that assign the value 4 to the variable a.

For model checking, the state space S has to be finite. Classic model checking proves that a model fulfills a specification given in a temporal logic. Other approaches, e. g. approaches based on automata theory, use finite state machines to represent the system model and the specification. Verification is done by showing that the externally visible behaviors of the model and the specification are compatible. In this work, temporal logic, i. e. computation tree logic is used.

For an arbitrary domain of states State and properties Prop, a finite Kripke structure with a total transition relation is defined as:[2]

Model[State, Prop]

$$S : \mathbb{F}_1\ \mathsf{State}$$
$$R : \mathsf{State} \leftrightarrow \mathsf{State}$$
$$L : \mathsf{State} \nrightarrow \mathbb{P}\ \mathsf{Prop}$$

$$R \subseteq S \times S$$
$$\mathsf{dom}\ R = \mathsf{dom}\ L = S$$

For a model, the set of *paths* are defined. A path of a model is an infinite sequence of states, where each subsequent pair of states is member of R.

Path[State, Prop]

$$\mathit{Model}[\mathsf{State}, \mathsf{Prop}]$$
$$\mathit{paths} : \mathbb{P}(\mathbb{N} \rightarrow \mathsf{State})$$

$$\mathit{paths} = \{\, \vec{s} : \mathbb{N} \rightarrow S \mid (\forall\, i : \mathbb{N} \bullet (\vec{s}\, i, \vec{s}(i+1)) \in R) \,\}$$

Note that while the state space is finite, the set of paths (or traces) is infinite. Both their number and their length are infinite. Properties verified by model checking consider paths rather then states. Thus, the model checking tools have to deal with the infiniteness of the paths. Therefore, it is imprecise to say a model checkers simply enumerates the finite state space. More precisely, model checkers use the fact that because of the finiteness of the state space any reachable state, can be reached after a finite number of states.

[2] The totality of the transition relation ($\mathsf{dom}\ R = S$) is required for the definition of the CTL semantics in the literature [22], [38]. This restriction is not mandatory. CTL and the model checking algorithms could also be defined without it. The difference is that in order to support non-total relations, finite paths (where the last state has no successor state) have to be considered, too. Model checkers such as the SMV rely on a total transition relation, however. Therefore, only total relations are considered here.

4.2 Computation Tree Logic

Model checking can prove the reachability or non-reachability of certain states or fulfillment of temporal logic properties. In the latter case, the model checking algorithms are specialized in a certain temporal logic. Still, not for every temporal logic there is an efficient model checking algorithm. Thus, the choice of the temporal logic is very important for model checking.

Clarke and Emerson [17], propose *Computation Tree Logic* (CTL) for the specification of properties of the model. According to the characterization of Emerson [22], CTL is a propositional, global (non-compositional), branching time, point-based, discrete temporal logic:

- *Branching time:* In distinction to *linear time*, in branching time logics, it is assumed that each state can have different futures. For this, branching time logics have operators to express whether a formula has to hold for *all* ($\mathcal{A}$) futures or for *some* ($\mathcal{E}$). This implies some differences as to how the semantics can be formulated: Roughly spoken, in branching time temporal logic, a formula holds for a state of a model, whereas in linear time, a formula holds for a state in a trace.
  Therefore, the fulfillment relation $\vdash$ is defined differently: A branching time formula T_{branch}, holds for a state s in a model M if and only if:

  $$(M, s) \vdash_b T_{branch}$$

  A linear time formula T_{lin},holds for the i-th state of a trace σ, if and only if:

  $$(\sigma, i) \vdash_l T_{lin}$$

  A branching time formula holds for a model, if it holds for all its (initial) states. A linear time formula holds for a model if it holds for (the first state of all) its traces.
  Model checkers use branching time logic to take advantage of the fact that formulae hold for single states. They recursively perform the verification by computing the sets of states that hold for each sub-formula. Since the state space is finite, these sets are finite, too, and can be represented in a computer. A set of traces, in contrary to states, may be infinite and is thus much harder to represent.
  Manna and Pnueli [37] present a linear time temporal logic, whereas CTL is a branching time logic.
  Branching time and linear time temporal logics also differ in their expressiveness. Consider the branching time formula (see Table 4.1 on the next page for a description of the CTL operators) $(\mathcal{AG}P) \Rightarrow (\mathcal{EF}Q)$ ("if P holds for all states of *all paths*, then *there is a path* with a state satisfying Q"). And the similar linear time formula $(\Box P) \Rightarrow (\diamond Q)$ ("*for any path*, if P holds for all states of the path, then Q holds for some state of this very path"). The difference is subtle, but important. If for a path of some model P, does not hold in all states, $\mathcal{AG}P$ is false and the model satisfies the

$\mathcal{EX}\,T$	exist next	there exists some next state fulfilling T
$\mathcal{AX}\,T$	always next	all next states fulfill T
$\mathcal{EG}\,T$	exist globally	their exists a path such that all states fulfill T
$\mathcal{AG}\,T$	always globally	all paths fulfill T globally
$\mathcal{EF}\,T$	exist finally	their exists a path with a state that fulfills T
$\mathcal{AF}\,T$	always finally	all paths have a state that fulfills T
$\mathcal{E}\,T\,\mathcal{U}\,T'$	exist until	their exists a path such that T holds until T' holds
$\mathcal{A}\,T\,\mathcal{U}\,T'$	always until	T holds until T' holds in all paths.

Table 4.1: CTL Operators

branching time formula for all paths. For the linear time formula, in contrary, there can be paths that are not fulfilled and therefore the model does not fulfill the formula.

- *Point Based:* Formulae are true or false at *points in time* rather then *time intervals*. In contrary to CTL, The interval logics, presented by Chaochen, Hoare and Ravn [16] or Büssow and Grieskamp [13], are not point based.
- *Discrete:* A temporal logic is discrete if each state has a next state. The time domain is thus the natural numbers. *Continuous* (or dense) logics, in contrary, interprete over the real (or rational) numbers.

CTL is defined for some set of properties Prop. Most of the operators can be derived. The names of the operators are shown in table 4.1.

function 200 leftassoc $(\_ \wedge \_)$

function 200 leftassoc $(\_ \vee \_)$

function 200 leftassoc $(\_ \Rightarrow \_)$

$$
\begin{aligned}
&\mathrm{CTL}[\mathsf{Prop}] ::= \mathit{prop}\langle\!\langle \mathsf{Prop} \rangle\!\rangle \\
&\qquad\qquad \mid \mathit{true} \\
&\qquad\qquad \mid (\_ \vee \_)\langle\!\langle \mathrm{CTL}[\mathsf{Prop}] \times \mathrm{CTL}[\mathsf{Prop}] \rangle\!\rangle \\
&\qquad\qquad \mid \neg\, \langle\!\langle \mathrm{CTL}[\mathsf{Prop}] \rangle\!\rangle \\
&\qquad\qquad \mid \mathcal{EX}\langle\!\langle \mathrm{CTL}[\mathsf{Prop}] \rangle\!\rangle \\
&\qquad\qquad \mid (\mathcal{E}\_\,\mathcal{U}\,\_)\langle\!\langle \mathrm{CTL}[\mathsf{Prop}] \times \mathrm{CTL}[\mathsf{Prop}] \rangle\!\rangle \\
&\qquad\qquad \mid (\mathcal{A}\_\,\mathcal{U}\,\_)\langle\!\langle \mathrm{CTL}[\mathsf{Prop}] \times \mathrm{CTL}[\mathsf{Prop}] \rangle\!\rangle \\
&\mathit{false} \quad == \neg\, \mathit{true} \\
&(\_ \wedge \_)[\mathsf{Prop}] == \lambda\, T, T' : \mathrm{CTL}[\mathsf{Prop}] \bullet \neg\, (\neg\, T \vee \neg\, T') \\
&(\_ \Rightarrow \_)[\mathsf{Prop}] == \lambda\, T, T' : \mathrm{CTL}[\mathsf{Prop}] \bullet \neg\, T \vee T' \\
&\mathcal{AX} \quad == \lambda\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet \neg\, (\mathcal{EX}(\neg\ T)) \\
&\mathcal{EG} \quad == \lambda\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet \neg\, (\mathcal{AF}(\neg\ T)) \\
&\mathcal{AG} \quad == \lambda\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet \neg\, (\mathcal{EF}(\neg\ T)) \\
&\mathcal{EF} \quad == \lambda\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet \mathcal{E}\ \mathit{true}\ \mathcal{U}\, T \\
&\mathcal{AF} \quad == \lambda\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet \mathcal{A}\ \mathit{true}\ \mathcal{U}\, T
\end{aligned}
$$

CTL formulae are evaluated for states of a model. The satisfaction relation $\vdash$ defines for which states of a given model a formula holds: $(M, s) \vdash T$ if the state s of the model

M satisfies T, for short: *s satisfies T in M.*

relation $(\_ \vdash \_)$

$$[\mathsf{State}, \mathsf{Prop}]$$

$$\_ \vdash \_ : (Model[\mathsf{State}, \mathsf{Prop}] \times \mathsf{State}) \leftrightarrow \mathrm{CTL}[\mathsf{Prop}]$$

$$\begin{aligned}
&\forall\, Path[\mathsf{State}, \mathsf{Prop}] \bullet \\
&\forall\, M == \theta Model;\ s : S;\ P : \mathsf{Prop};\ T, T' : \mathrm{CTL}[\mathsf{Prop}] \bullet \\
&\quad ((M, s) \vdash true) \land \\
&\quad ((M, s) \vdash prop\ P \Leftrightarrow P \in L(s)) \land \\
&\quad ((M, s) \vdash T \lor T' \Leftrightarrow ((M, s) \vdash T) \lor ((M, s) \vdash T')) \land \\
&\quad ((M, s) \vdash \neg\ T \Leftrightarrow \neg\ ((M, s) \vdash T)) \land \\
&\quad ((M, s) \vdash \mathcal{EX}\ T \Leftrightarrow (\exists\, s' : S \bullet (s, s') \in R \land (M, s') \vdash T)) \land \\
&\quad ((M, s) \vdash \mathcal{A}\, T\ \mathcal{U}\ T' \Leftrightarrow \\
&\qquad (\forall\, \vec{s} : paths \mid head\ \vec{s} = s \bullet \\
&\qquad\quad \exists\, k : \mathsf{dom}\ \vec{s} \bullet (M, \vec{s}\, k) \vdash T' \land (\forall\, i : 0 \mathinner{..} k - 1 \bullet (M, \vec{s}\, i) \vdash T))) \land \\
&\quad ((M, s) \vdash \mathcal{E}\, T\ \mathcal{U}\ T' \Leftrightarrow \\
&\qquad (\exists\, \vec{s} : paths \mid head\ \vec{s} = s \bullet \\
&\qquad\quad \exists\, k : \mathsf{dom}\ \vec{s} \bullet (M, \vec{s}\, k) \vdash T' \land (\forall\, i : 0 \mathinner{..} k - 1 \bullet (M, \vec{s}\, i) \vdash T)))
\end{aligned}$$

Model checkers usually add a set of initial states I to the Kripke structure and proof satisfaction of formulae for the set of initial states. So the model checker shows that a model $M = (S, R, L)$ with a set of initial states $I \subseteq S$ satisfies a formula P: $M, I \models P$.

relation $(\_ \models \_)$

$$[\mathsf{State}, \mathsf{Prop}]$$

$$\_ \models \_ : (Model[\mathsf{State}, \mathsf{Prop}] \times \mathbb{P}\, \mathsf{State}) \leftrightarrow \mathrm{CTL}[\mathsf{Prop}]$$

$$\begin{aligned}
&\forall\, M : Model[\mathsf{State}, \mathsf{Prop}];\ I : \mathbb{P}\, \mathsf{State};\ T : \mathrm{CTL}[\mathsf{Prop}] \mid I \subseteq M.S \bullet \\
&\quad (M, I) \models T \Leftrightarrow (\forall\, s : I \bullet (M, s) \vdash T)
\end{aligned}$$

section *CTLTransform* **parents** *ModelChecking*

The function map_{CTL} maps a property-transformation function to CTL formulae made of such properties. It will be used in the following chapters as an auxiliary function to describe CTL transformations.

$$[X, Y]$$

$$\mathsf{map}_{CTL} : (X \nrightarrow Y) \to \mathrm{CTL}[X] \nrightarrow \mathrm{CTL}[Y]$$

$$\begin{aligned}
&\forall\, f : X \nrightarrow Y;\ T, T' : \mathrm{CTL}[X];\ x : X \bullet \\
&\quad \mathsf{dom}(\mathsf{map}_{CTL}\, f) = \mathrm{CTL}[\mathsf{dom}\, f] \land \\
&\quad (\mathsf{map}_{CTL}\, f)(true) = true \land \\
&\quad (\mathsf{map}_{CTL}\, f)(prop\ x) = prop(f\ x) \land \\
&\quad (\mathsf{map}_{CTL}\, f)(\neg\ T) = \neg\ ((\mathsf{map}_{CTL}\, f)\ T) \land \\
&\quad (\mathsf{map}_{CTL}\, f)(T \lor T') = (\mathsf{map}_{CTL}\, f\ T) \lor (\mathsf{map}_{CTL}\, f\ T') \land \\
&\quad (\mathsf{map}_{CTL}\, f)(\mathcal{AX}\ T) = \mathcal{AX}((\mathsf{map}_{CTL}\, f)\ T) \land \\
&\quad (\mathsf{map}_{CTL}\, f)(\mathcal{A}\, T\ \mathcal{U}\ T') = \mathcal{A}(\mathsf{map}_{CTL}\, f)\ T\ \mathcal{U}\ (\mathsf{map}_{CTL}\, f)\ T' \land \\
&\quad (\mathsf{map}_{CTL}\, f)(\mathcal{E}\, T\ \mathcal{U}\ T') = \mathcal{E}(\mathsf{map}_{CTL}\, f)\ T\ \mathcal{U}\ (\mathsf{map}_{CTL}\, f)\ T'
\end{aligned}$$

4.3 The Model Checking Algorithm for CTL Formulae

The model checking algorithm computes, for some given formula T, the set of states satisfying the formula: $\{ s : S \mid (M, s) \vdash T \}$. This set is computed inductively by the function $sat : S \to \mathbb{P}\,\mathsf{State}$ (the rules for $\mathcal{A}\_\mathcal{U}\_$ are omitted).

$Sat_0[\mathsf{State}, \mathsf{Prop}]$

$Model[\mathsf{State}, \mathsf{Prop}]$
$SatExUntil[\mathsf{State}, \mathsf{Prop}]$
$sat : \mathrm{CTL}[\mathsf{Prop}] \to \mathbb{P}\,\mathsf{State}$

$$
\begin{aligned}
&\forall\, T, T' : \mathrm{CTL}[\mathsf{Prop}];\ P : \mathsf{Prop} \bullet \\
&\quad sat(true) = S \land \\
&\quad sat(prop\ P) = \{ s : S \mid P \in L(s) \} \land \\
&\quad sat(\neg\ T) = S \setminus sat(T) \land \\
&\quad sat(T \land T') = sat(T) \cap sat(T') \land \\
&\quad sat(\mathcal{EX}\ T) = \{ s : S \mid (\exists\, s' : sat(T) \bullet (s, s') \in R) \} = R^{\sim}(\!|sat(T)|\!) \land \\
&\quad sat(\mathcal{E}\ T\ \mathcal{U}\ T') = sat\_exuntil(sat(T), sat(T'))
\end{aligned}
$$

$$Sat[\mathsf{State}, \mathsf{Prop}] == Sat_0[\mathsf{State}, \mathsf{Prop}] \setminus (sat\_exuntil)$$

In order to compute the states fulfilling an *until*-formula, i. e. $T_0 = \mathcal{E}\ T\ \mathcal{U}\ T'$, the graph of the state transition relation is considered (see Figure 4.1 on page 42). A path in the Kripke structure corresponds to a path in the graph. Finding the states which fulfill an *until*-formula can thus be regarded as a graph traversal problem. A state that fulfills T_0 either fulfills T', or there is a path from this state to a T'-state, where all intermediate states of the path are T-states. The computation is done backwards. First all states fulfilling T' are computed. Then all T-states that are predecessors of these states are added. This is repeated until no more additional states are added. Since the set of states is finite, the computation terminates after a finite number of steps, that is at most $\#S$: It can be shown that if there is an infinite path to some state, their is also a finite one: Because the number of states is finite, in an infinite path, states have to occur several times, i. e. the path has loops. If the loops are removed, a valid path with respect to the transition relation remains. If a state is reachable in a path after an infinite number of states, loops before the occurrence of that state can be removed, and the resulting path is still valid. If all loops are removed, each state appears not more than once in the path and hence is a finite path leading to the state. The resulting set of states is the set of states fulfilling T_0.

This computation is done by the function *sat_exuntil* for some given model. As input the function gets the invariant $\vec{T}$ that has to hold for all states in the path and a set of already collected states $\tilde{S}$. $\tilde{S}_n$ is the set of states added in the step, that is the set of states that are reachable from $\tilde{S}$: $(R^{\sim})(\!|\tilde{S}|\!)$ and that are a member of $\vec{T}$. As the computation is done backwards, the reverse $R^{\sim}$ of the transition relation R is used to compute the

reachable states.. The computation terminates if no more states are added: $\tilde{S}_n \cup \tilde{S} = \tilde{S}$ (or $\tilde{S}_n \subseteq \tilde{S}$).

SatExUntil[State, Prop]

Model[State, Prop]

$sat\_exuntil : \mathbb{P}\,\mathsf{State} \times \mathbb{P}\,\mathsf{State} \pfun \mathbb{P}\,\mathsf{State}$

$\mathsf{dom}\ sat\_exuntil = \mathbb{P}\,S \times \mathbb{P}\,S$

$\forall\, T_S, \tilde{S} : \mathbb{P}\,S \bullet$

$\quad (\exists\, \tilde{S}_n == (T_S \cap (R^\sim)(\!|\tilde{S}|\!)) \bullet$

$\quad\quad sat\_exuntil(T_S, \tilde{S}) = (\mathbf{if}(\tilde{S}_n \cup \tilde{S} = \tilde{S})\mathbf{then}\ \tilde{S}\ \mathbf{else}\ sat\_exuntil(T_S, \tilde{S} \cup \tilde{S}_n)))$

The algorithm for checking a CTL formula T runs in $\mathcal{O}(length(T)\,\#S^2)$ time, where $length(T)$ denotes the number of CTL operators in T. Note, however, that the computation critically depends on the representation of the set of states and the computation of its union and intersection. So does the cost of the application of the state transition relation. An efficient solution for this problem is the usage of BDDs (see section 4.5 on the following page).

4.4 Fixed-Point Definition of the Model Checking Algorithm

The computation of the satisfying states for an *until*-operator can also be expressed by a fixed-point definition. Consider the following axiom that holds for *until*:

[State, Prop]

Model[State, Prop]

$\forall\, T, T' : \mathrm{CTL}[\mathsf{Prop}] \bullet \mathcal{E}\,T\ \mathcal{U}\ T' = T' \vee T \wedge \mathcal{E}\mathcal{X}(\mathcal{E}\,T\ \mathcal{U}\ T')$

With this, the set of states, fulfilling a formula $T(\mathcal{E}\_\ \mathcal{U}\ \_)\,T'$ can be defined as the least fixed-point (under set inclusion) of the function $\lambda\, Y \bullet T' \vee (T \wedge \mathcal{E}\mathcal{X}\, Y)$. The exact definition is:

[State, Prop]

Model[State, Prop]

$\forall\, T, T' : \mathrm{CTL}[\mathsf{Prop}];\ Sat[\mathsf{State}, \mathsf{Prop}] \bullet$

$\quad sat(\mathcal{E}\,T\ \mathcal{U}\ T') = \mathsf{lfix}(\_ \subseteq \_)(\lambda\, Y : \mathbb{P}\,S \bullet sat(T') \cup (sat(T) \cap R^\sim(\!|\,Y\,|\!)))$

For the proof see Clarke, Grumberg, and Peled [18].

The computation of the satisfying states defined in *Sat* has to be compatible with the CTL semantics, i. e.:

$$\forall\, Sat[\mathsf{State}, \mathsf{Prop}] \bullet \forall\, T : \mathrm{CTL}[\mathsf{Prop}] \bullet sat\ T = \{\, s : S \mid (\theta Model, s) \vdash T \,\}$$

This property can be shown by structural induction over the CTL syntax. Hereby, the induction step for *until* is shown by induction over the length of the interval, which is k in the definition of $(M, S) \vdash \mathcal{E}\,T\ \mathcal{U}\ T'$.

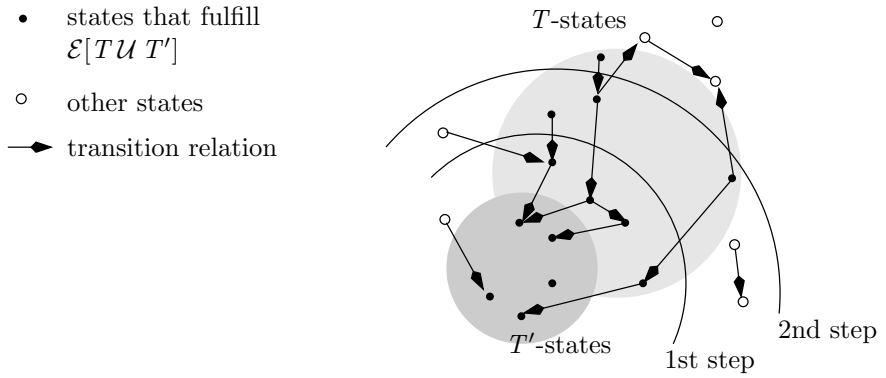


Figure 4.1: The CTL model checking algorithm for a formula $\mathcal{E}[T\,\mathcal{U}\,T']$. *The figure displays the graph of the transition relation R. First, the states that fulfill T' are computed (dark circle). From these states, the transition relation is traversed backwards, adding states that fulfill T. So the algorithm "radiates" outward from the states that fulfill T'. This is repeated, until no more states are added.*

Remark on complexity: Clarke and Emerson [17] show that it is not possible to find a polynomial time model checking algorithm for any temporal logic. If the CTL language is extended by path quantifiers that prefix an arbitrary assertion (i. e. $\mathcal{E}(\mathcal{F}P_1 \wedge \ldots \wedge \mathcal{F}P_n \wedge \mathcal{G}Q_1 \wedge \ldots \wedge \mathcal{G}Q_m)$), it can be shown that the model checking problem for this logic also solves the Hamiltonian Path problem and is thus NP-hard. This shows how important the choice of the temporal logic is for model checking.

4.5 Symbolic Model Checking

The states of a Kripke structure are usually built by a set of variables and some kind of program counter (here the program counter is the Statechart configuration, i. e. the set of active or entered states). Thus, a state is a valuation function of the variables plus the current value of the program counter. The properties are atomic properties over the variables (e. g. $x = 4$, $x \leq y$). The interpretation function (L) is then defined as assigning each state (variable valuation) the atomic properties that are true under this very valuation. Consider for example a variable valuation function $V \in \mathit{Val}$ and an evaluation function $[\![\_]\!] : \mathit{Prop} \times \mathit{Val} \to \{\, \mathit{true}, \mathit{false} \,\}$, then $L(V) = \{\, P : \mathit{Prop} \mid [\![P]\!]_V = \mathit{true} \,\}$.

How efficiently the presented model checking algorithm can be executed depends on:

- the representation of sets of states (The state space grows exponentially with the number of variables. Dealing with this *state explosion problem* is crucial for model checking),
- the computation of equality of two sets of states wich is needed for the termination condition,
- the computation of the union and intersection of two sets of states,
- the representation of the state transition relation, and
- the application of the state transition relation to a set of states.

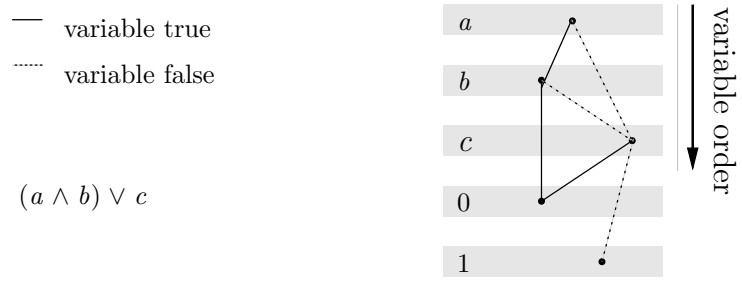


Figure 4.2: Representing Formulae as Ordered Binary Decision Diagrams.*OBDDs represent formulae over Boolean variables as a directed acyclic graph. The formula is normalized in an if-then-else form, e. g.* $(a \wedge b) \vee c$ *is normalized to:*

if a **then if** (b **then** *true* **else** **if** c **then** *true* **else** *false*)

else **if** c **then** *true* **else** *false*

The variable ordering (here a-b-c) defines at which depth the variables occur in the term. In order to avoid exponential growth of the normalized formula, common sub-trees are shared. Sharing is in fact the most important technique used for BDDs to avoid exponential growth. In the example, this can be done for the gray underlayed sub-term.
A node in a BDD represents a (sub-)formula, i. e. one if-then-else term for a specific variable. Thus, each node is a variable assigned. The nodes can be seen to be in layers, where each layer holds all nodes of a specific variable. The node has two child nodes, one for the then-case and one for the else-case. A BDD has exactly two leaves, one for true and one for false.
With the decision diagrams, one can test whether a formula is true for a specific variable valuation. For that, the graph is traversed from the node, representing the formula. The routing is done with respect to the variable valuation. If the "true"-leave is reached, the formula holds for the valuation, otherwise it does not hold.

McMillan [38] used propositional formulae to represent sets of states. So the set of all states, where the variable x has the value 4, is represented by the formula $x = 4$. The state transition relation is represented by formulae over two pairs of these variables, one for the pre-state and one for the post-state. Z uses the same technique to represent operations. The variables of the post-state are displayed primed. Incrementing x by one is for example represented by $x' = x + 1$. Unions and intersections are computed by conjunction and disjunction, respectively.

McMillan called his approach *symbolic* model checking, since the sets of states are not represented explicitly, but by formulae. For McMillan's approach it is still necessary that the value domain of the variables is known and finite, since he represents them with binary decision diagrams. This is an important difference to other *symbolic* techniques, such as axiomatic theorem provers.

For the computation of the next operator, the state transition relation has to be applied backwards. This backward application can be done by existential quantification. That means, for a system with state variables $x_1, \ldots, x_n$ and a set of states defined by the predicate $\tilde{S}$,

$$\exists\, x_1', \ldots, x_n' \bullet \tilde{S}[x_1'/x_1, \ldots, x_n'/x_n] \wedge R$$

defines the set of states that have a next state in $\tilde{S}$.

4.6 Binary Decision Diagrams (BDDs)

For the representation of the formulae, McMillan used Bryant's [7] *Binary Decision Diagrams* (BDD). BDDs provide a size-efficient, canonical representation of predicates over Boolean variables. The size is reduced by sharing common subtrees (see Figure 4.2 on the page before. Bryant also provides fast algorithms (quadratic in the size of the BDDs) to compute the conjunction or disjunction of two formulae. The restriction to Boolean variables does not cause a problem, since the value domains of the state variables are finite. Non-Boolean state variables can be represented by a set of Boolean variables in the same way as such variables are represented by sets of bits in computer hardware. The same holds for addition, subtraction, etc. These operations have to be realized as logical operations over the "bits", representing the numbers. Existential quantification needed for the application of the state transition relation can be translated into a disjunction, using the following property:

$$\exists\, x : \{\, \mathit{true}, \mathit{false} \,\} \bullet P \quad \equiv \quad P[\mathit{true}/x] \vee P[\mathit{false}/x]$$

The cost of the BDD computation depends on the size of the BDD. Due to the "symbolic" representation of the state space and the transition relation, the size of the BDDs does not directly depend on the size of the represented set. However, the size of the set defines an upper bound of the size of BDD (n Boolean variables spawn a state space of 2^n states. With no sharing at all, which is the worst case, the BDD is a binary tree of depth n,

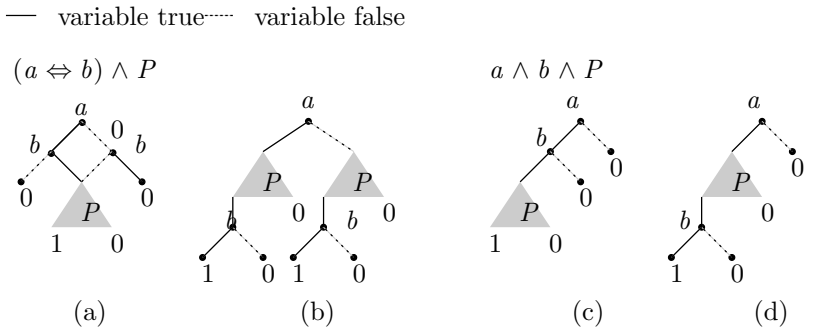


Figure 4.3: BDD Complexity and Variable Ordering. *Only in (b) the BDD, representing P, appears twice in the resulting BDD. The variable ordering in (a) avoids this. In (c) and (d) the variables are not as closely related. Their ordering is therefore not important.*

---

having $2^n - 1$ nodes.) The actual size depends on the "complexity" of the formula and on the variable ordering. Roughly speaking, variables that are closely related, should also be close in the variable ordering. Whether this can be achieved depends on the "complexity" of the formula.

For example the BDD representing the set of all states where the Boolean variable a is true needs one BDD node, no matter how big the state space is.

Another example is shown in Figure 4.3. Consider the formula $(a \Leftrightarrow b) \wedge P$, where a and b are Boolean variables and P is s sub-formula with no occurences of a or b. The BDD representing this formula depends on ordering of the a, b, and the variables occuring in P. If all variables from P are in between a and b (Figure 4.3(b)), the resulting BDD will have twice the size of P's BDD, whereas if a and b are above all variables from P (Figure 4.3(a)), the BDD grows only by two nodes. Thus, this formula depends on the variable ordering. For the formula $a \wedge b \wedge P$, in contrary, the BDD would grow only by two in both cases. For these reasons it is hard to tell the size of a BDD representing a specific model. Therefore, it is also difficult to find out, how big a model may be to be feasible with symbolic model checking.

The largest Statemate model that was translated with the technique described in this thesis had a state space of 10^{34} states, where 10^{25} states were reachable. The BDD representing the transition relation had 53,000 BDD nodes. Verification of about 40 reachability formulae took 2 minutes on a 233 MHz Intel Pentium II. The model was only feasible with a suitable variable ordering that had to be computed in several steps. This was done using the *dynamic variable reordering* feature of the model checker. This computation took several hours. The largest models that are feasible today, have up to 10^{120} states ([18]).

Note that the size of the state transition relation is the power of two of the size of the state space (since $R \subseteq S \times S$). Therefore, it is usually the size of the BDD representing the state transition relation that is critical, not the representation of the sets of states.

Experience with symbolic model checking has shown that control problems, by contrast to data driven problems, can be handled quite efficiently with symbolic model checking

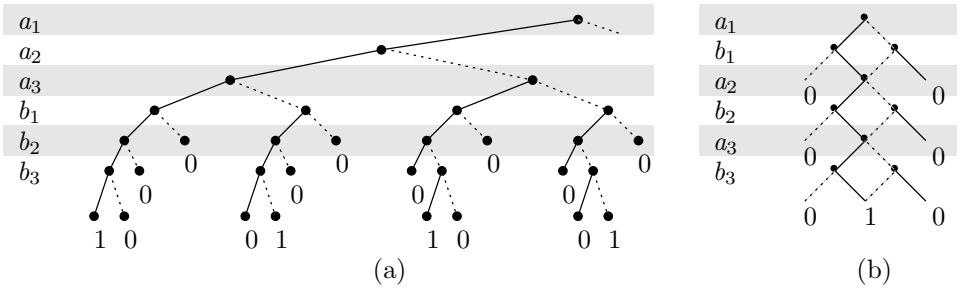


Figure 4.4: Exponential BDD Growth when Comparing two Integer Variables. *The depicted BDDs test equality for two three-bit integer variables. The size of the BDD depends on the variable ordering. In (a) the BDD grows exponentially with the number of bits, whereas in (b) a better ordering causes linear growth only.*

(the state explosion problem omitted). Normal algebra already causes problems: efficiency of addition and subtraction depends on the variable ordering, as shown in Figure 4.4. Multiplication cannot be represented efficiently at all, i. e. the size of the BDD grows exponentially with the number of bits needed to represent the multiplied variables.

4.7 Other Model Checking Techniques

Besides CTL model checking and the application of BDDs to model checking, a number of other techniques are presented by Clarke, Grumberg, and Peled [18]. These techniques either extend the expressiveness of the input language (μ-calculus and real time model checking) or try to tackle the state explosion problem with more sophisticated approaches (partial order reduction and abstraction). The most important among these techniques are presented in this section.

4.7.1 Mu-Calculus Model Checking

The propositional μ-calculus is an alternative to CTL for the specification of properties to be checked. The μ-calculus is a temporal logic as well. It is more expressive than CTL, and many temporal and program logics (including CTL) can be translated into the μ-calculus. Instead of temporal operators such as *always*, *sometimes*, or *until* ($\mathcal{AG}, \mathcal{EG}, \mathcal{AF}, \mathcal{EF}, (\mathcal{A}\_\mathcal{U}\ \_), (\mathcal{E}\_\mathcal{U}\ \_)$ in CTL), the μ-calculus offers fixed-point operators. The temporal operators can be constructed with these operators, using the fixed-point equations that are also used by the CTL model checking algorithms. E. g. for the always operator the following rule holds: $\mathcal{EG} f = f \wedge \mathcal{EX}\ \mathcal{EG}\ f$. With this rule, the satisfaction function *sat* that computes the set of states fulfilling a CTL formula, can be defined for the always operator by a fixed-point definition:

$$sat(\mathcal{EG} f) = \mathsf{gfix}((\_ \subseteq \_), \lambda\, Y \bullet sat(f \wedge \mathcal{EX}\ Y))$$

In μ-calculus syntax, this is expressed by: $\nu\, Y.\ f \wedge \langle a \rangle\, Y$, where a is the state transition relation.

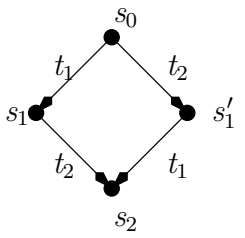


Figure 4.5: Independent Transitions

The fixed-point operators make the μ-calculus very flexible, and so it is well suited as a core formalism for model checkers that can be reused for different input logics. On the other hand, it is rather impractical to use the μ-calculus directly as specification formalism, because the formlae are not very intuitive.

μ-calculus formulae evaluate to sets of states the same way CTL formulae do. Symbolic (BDD based) technique can also be applied for μ-calculus-model-checking and model checkers for this language exist. For model checking, the μ-calculus is thus a good target language of other temporal languages.

4.7.2 Partial Order Reduction

With Partial order reduction or *model checking through representatives*, the search performed for the verification can be reduced. Partial order reduction does not decrease the actual number of states as *abstraction* does. It reduces the number of traces to be searched. Partial order reduction uses the fact that often times, the order of execution of some transitions is of no relevance. In Figure 4.5, for example, it is shown that for the transitions t_1 and t_2 the order of application does not matter ($t_1 \circ t_2 = t_2 \circ t_1$). In such cases, it is sufficient to consider only one possible execution order and reduce the number of paths to be searched. This may also lead to a reduction of the number of states, since, in the example, one of the states s_1' or s_1 is not needed anymore, after partial order reduction is applied.

Independent transitions occur mostly in asynchronous systems, where parallel execution is expressed by interleaving. With interleaving, parallel transitions are applied sub-sequentially in an undetermined order. The model checker has to consider all execution orders. If the result of the application does not depend on the execution order, only one order has to be considered. This is done by partial order reduction.

In fact, the independent transitions are introduced by the interleaving semantics. Since Statecharts do not have an interleaving semantics, partial order reduction is not useful for model checking their properties.

Partial order reduction is built into the Spin model checker presented by Holzmann and Peled [30].

original	**abstract**
$x : \mathbb{Z}$	$\bar{x} : \{ pos, zero, neg \}$
$x' = x + 1$	$\bar{x}' \in \begin{cases} \{ pos \} & \text{iff. } \bar{x} \in \{ pos, zero \} \\ \{ neg, zero \} & \text{iff. } \bar{x} \in \{ neg \} \end{cases}$
$x > 0$	$\bar{x} = pos$
$x < 0$	$\bar{x} = neg$

Table 4.2: Data Abstraction for an Integer Variable

4.7.3 Abstraction

The most important technique for the reduction of the number of states in the state space and thereby avoiding the state explosion problem, is abstraction. They are two approaches for abstraction.

The first technique, the *cone of influence reduction*, reduces the variables that span the state space. For a property to be shown, the set of variables is reduced to those variables that are of interest for the property (i. e. are used in the formula, defining the property) plus all variables these variables depend on. It is easy to apply the cone of influence reduction automatically. However, it depends on the model and the property, whether an actual reduction takes place. In many cases, there is no *cone* that is smaller than the state space, so there is no reduction in the end.

Verification of data-driven models is one the of the greatest challenges of model checking, since *data* quickly lead to large data space. The largest models that are tackled by symbolic model checking today have a state space of up to 10^{120} states. This is about 400 boolean variables. With a state space of this size, it is not possible to encode more than twelve 32-bit integers, which will be insufficient even for medium sized systems. Moreover, typical operations such as multiplication cannot be executed efficiently with the proposed techniques.

The second technique, the *data abstraction*, can be used to tackle these problems. Data abstraction takes into consideration that, for a lot of properties to be shown, the actual data values are not important. Therefore, it is sufficient to consider only the important aspects of a date, i. e. the aspects that influence the property and the behavior of the model. An integer variable, for example, could be abstracted to three states, (1) less than zero, (2) equals zero, and (3) greater than zero. Such an abstraction is shown in Table 4.2

The problem of data abstraction is that there is no reasonable automatic way to find a good abstraction. Hence, data abstraction must be done by the modeler. You can do this by considering the abstraction within the model itself, or by providing an abstraction function and thus performing the abstraction semi-automatically.

4.7.4 Real-Time

Checking real-time constraints (i. e. verifying required response times) is hardly possible with CTL or μ-calculus model checking. It is possible to state that some event e has to happen in the future ($\mathcal{AG}\,e$). However, for model checking real time properties, one has to state *when* e happens. For *discrete real-time* this can be done by using the *next* operator, so the property that e happens within the next three steps can be specified by $\mathcal{EX}(e \vee \mathcal{EX}(e \vee \mathcal{EX}\,e))$. For such properties, *real-time CTL* (RTCTL) was introduced by Emerson et. al. [23]. RTCTL introduces the *bounded until* operator that is annotated with a time interval in which the property must be true. In $\mathcal{E}\,T\,\mathcal{U}_{[a,b]}\,T'$ the sub-formula T' has to be true after a steps and before b steps (the above example can be specified by $\mathcal{E}\ true\,\mathcal{U}_{[1,3]}\,e$). RTCTL can be model checked with only slightly modified CTL model checking algorithms. For checking $\mathcal{EG}_{[a,b]}\,e$, the first a steps are not considered in the fixed-point computation, and the computation is stopped, after b stopped, rather then after no additional states are found.

With discrete real-time, events can only happen at integer time values. This approach is sufficient for synchronous systems, where all subsystems are synchronized by a global clock and events happen simultaneously. For asynchronous systems, it is not sufficient. Continuous model checking can be done for *Timed Automata*. These automata differentiate between "discrete" steps (*actions*) that change values of variables and "continuous" (time) steps that cause time to elapse. Change of variable values takes place only during discrete actions.

Continuous real-time models are inherently infinite. Thus, in order to model check them, a finite representation has to be found. Alur [1] tackles these problems in introducing *clock zone* and *difference bound matrices*.

4.8 Model Checking MSZ

4.8.1 Instantiating Kripke Structures

The presented theory has to be instantiated for $\mu\mathcal{SZ}$, in order to check properties of $\mu\mathcal{SZ}$ classes. For this, $\mu\mathcal{SZ}$ classes have to be translated into *Kripke structures* and CTL has to be defined for the classes. The definitions of the previous sections will be used for this.

As described by Büssow et. al. [11], there are implicitly defined schemata called DATA and INIT for each $\mu\mathcal{SZ}$ class. The first is defined as the schema conjunction of all data and port schemata of the class, hence it defines the data space of the class. The second is defined as the conjunction of all init schemata, therefore it defines the set of initial states of the class.

In order to provide an interface for the integration of formalisms that specify the dynamic behavior of the class, Büssow and Grieskamp [14] have proposed a schema TRANS, defining the state transition relation of the class in a similar way. Formalisms can be integrated in providing a translation into such a state transition relation. This is done for Statecharts in section 6.2 on page 78.

With these schemata, the Kripke state space and the transition relation can be defined immediately by the DATA schema and the TRANS schema respectively. A state s is a Z binding of the variables declared in DATA, which means $s \in \mathrm{DATA}$. Thus, for the Kripke structure, DATA can be used as the state space. The state transition relation has to be a subset of DATA $\times$ DATA. However, TRANS is a subset of ΔDATA. Therefore, TRANS has to be transformed, using some Z tricks, to form a relation:

$$R = \{\, \mathrm{TRANS} \bullet (\theta\mathrm{DATA}) \mapsto (\theta(\mathrm{DATA})') \,\}$$

Properties are defined as schemata over the data state space, they are therefore sets of bindings ($P \in \mathbb{P}\,\mathrm{DATA}$), too. The satisfaction function L is defined such that a state $s \in \mathrm{DATA}$ satisfies a property P if and only if $s \in P$. In class *Example*, shown in Figure 4.6 on the next page, the property *xGreater* is defined. A state (binding) $s = \langle x == 3, y == 2 \rangle$ fulfills *xGreater*, since $s \in \mathit{xGreater}$.

CTL

MSZModel

$\mathit{Model}[\mathrm{DATA}, \mathbb{P}\,\mathrm{DATA}]$

$S = \mathrm{DATA}$

$R = \{\, \mathrm{TRANS} \bullet (\theta\mathrm{DATA}) \mapsto (\theta(\mathrm{DATA})') \,\}$

$L = \lambda\, s : \mathrm{DATA} \bullet \{\, P : \mathbb{P}\,\mathrm{DATA} \mid s \in P \,\}$

For arbitrary $\mu\mathcal{SZ}$ classes, *MSZModel* defines the Kripke structure representing the class. Note, however, that the state space has to be finite and the transition relation has to be total. With this, CTL (i. e. $\mathit{CTL}[\mathbb{P}\,\mathrm{DATA}]$) and the semantics of model checking are defined for $\mu\mathcal{SZ}$ classes. *MSZModel* is imported into other classes via the $\mu\mathcal{SZ}$'s *class enrichment*.

A class fulfills a CTL property $T \in \mathit{CTL}[\mathbb{P}\,\mathrm{DATA}]$ if and only if

$$(\mathit{MSZModel}, \mathrm{INIT}) \models T$$

Model checkers, such as the SMV model checker, prove a given CTL formula for a Kripke structure in checking the above given property. A special fulfillment operator $\Vdash$ for this property is defined for $\mu\mathcal{SZ}$ classes. The model checker is able to verify exactly this fulfillment relation.

CTL

relation $(\Vdash\_)$

CTLProperty

MSZModel

$\Vdash\_ : \mathbb{P}\,\mathrm{CTL}[\mathbb{P}\,\mathrm{DATA}]$

$\forall\, T : \mathrm{CTL}[\mathbb{P}\,\mathrm{DATA}] \bullet$

$\quad \Vdash T \Leftrightarrow (\theta\mathit{Model}, \mathrm{INIT}) \models T$

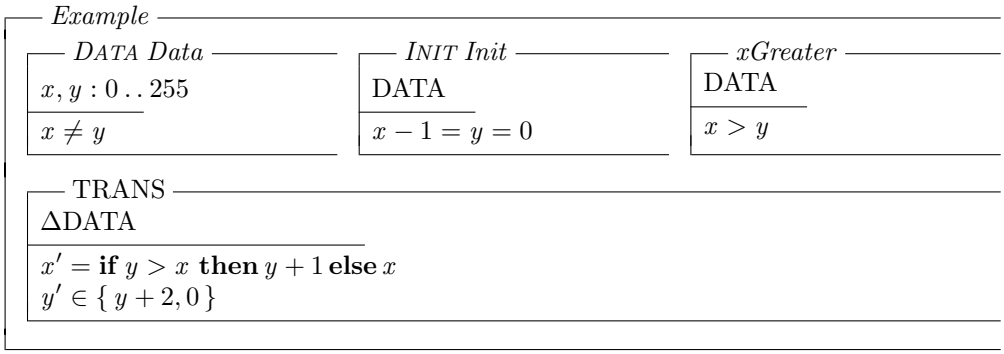


Figure 4.6: Data space, transition relation and property schemata of a class

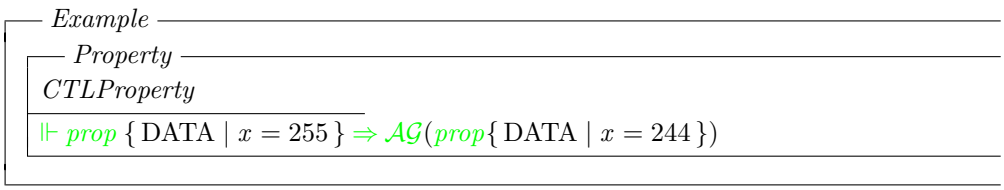


Figure 4.7: CTL property for the example class. *If in any state of any path x equals 255, it will keep this value in all following states.*

The usage of CTL and its fulfillment operator is shown in Figure 4.7.

4.8.2 Different Kinds of Variable Declarations

The previous section explained how Kripke structures can be instantiated for $\mu\mathcal{SZ}$ specifications. This instantiation raises some technical problems, which are discussed now.

Consider the different kinds of variable declarations occurring in an $\mu\mathcal{SZ}$ specification:

- *Axiomatic Definitions:* Variables can be declared in axiomatic definitions. These variables do not belong to the data space of any class. They never change their value during execution. However, there value may not be defined definitly. Consider for example the declaration of the variable *limit*:

  $$\begin{array}{|l} limit : NUMBER \\ \hline limit \leq 255 \end{array}$$

  A specification containing such declarations is considered to be a *loose specification*. A loose specification can also be seen as a family of specifications. In the above example, one specification for each value *limit* can take.

  The concept of a loose specification does not exist in Kripke structures. In order to support loose specifications, the variables causing the specification to be loose have to

be added to the state space. Their initial value is not defined, but restricted by the property. The variables are constant, i. e. they never change their value: for *limit* the following equation is added to the transition relation: $limit' = limit$. This variable must not be written by any action.

DATA D

$limit : NUMBER$

INIT I

DATA

$limit \leq 255$

TRANS T

ΔDATA

$limit' = limit$

- *Plain Schemata:* Variables can be declared in ordinary Z schemata. These declarations affect the specification only if the are used somewhere. Therefore, declarations in plain schemata do not have to be considered.
- *Port- and Data-Schemata:* Declarations of port- and data-schemata are compiled to the DATA schema, as already mentioned. In Z, a schema can also be seen as a set of bindings. The DATA schema was used in the previous section to define the system's state space to be the set of bindings defined by the DATA schema. It is, however, not always easy to compute this set of bindings, since it can be restricted by arbitrary properties (invariants) in the port- and data-schemata. Consider the following example (x and y must not be equal and x is a prime number):

  DATA D

  $x, y : 0 \,..\, 255$

  $x \neq y \land (x \geq 100 \Rightarrow y \in \{ x, 4 \})$

  A model checker, such as the SMV, expects the state space to be given as a set of variable declarations, for the above example: `x:0..255; y:0..255`. Therefore, only the variable declarations are taken into consideration for the state space. In order to preserve the invariants, they are added to the state transition relation.

The formal definition of *MSZModel* and *CTLProperty* that takes the above presented concepts into consideration is given later.

4.9 Kripke Structure With Constants

The Kripke structure is the link between model checking theory and $\mu\mathcal{SZ}$ specifications. A Kripke structure is defined, considering constant variables.

Assuming that a schema *CONST* contains all constant variables and predicates over them, *MSZModelWithConstants* defines the Kripke structure that represents a $\mu\mathcal{SZ}$ class.

CTL

$DATACONST == \text{DATA} \land CONST$

$INITCONST == \text{INIT} \land CONST$

MSZModelWithConstants

$$Model[DATACONST, \mathbb{P}\, DATACONST]$$

$$\begin{aligned}
&S = DATACONST \\
&R = \{\, \text{TRANS};\ \Delta CONST \mid \theta CONST = \theta(CONST)' \bullet \\
&\qquad (\theta DATACONST) \mapsto (\theta(DATACONST)')\,\} \\
&L = (\lambda\, s : DATACONST \bullet \{\, P : \mathbb{P}\, DATACONST \mid s \in P\,\})
\end{aligned}$$

CTLPropertyWithConstants

$$MSZModelWithConstants$$

$$\Vdash\_ : \mathbb{P}\, \text{CTL}[\mathbb{P}\, \text{DATA}]$$

$$\forall\, T : \text{CTL}[\mathbb{P}\, DATACONST] \bullet \Vdash T \Leftrightarrow (\theta MSZModelWithConstants, INITCONST) \models T$$

The data space and transition relation is also augmented by the Statechart translation. For our purpose, it is assumed that this translation is already done and that the class therefore contains the necessary declarations and predicates.

The schemata defined in the class *CTL* provide means of using CTL in a $\mu\mathcal{SZ}$ class and define the semantic relation to model checking. The latter is done on a semantical basis, using the implicitly defined schemata DATA, INIT, TRANS, and *CONST*. The translation that is described in the following does not use these schemata. It translates the schemata of the respective roles directly. The transformation relation is constructed by translating the predicates of all schemata with role *TRANS* (see section 8.2.4 on page 141) rather then translating the schema *MSZModelWithConstants*.

4.10 Semantic Issues

The presented approach implies that Kripke structures are used as a semantic model for $\mu\mathcal{SZ}$ classes. The reactive semantic of a class is defined by the Kripke structure *MSZModel*. This is not necessarily compatible with other approaches to model the $\mu\mathcal{SZ}$ semantics, for instance the $\mu\mathcal{SZ}$ semantic of Geisler [24]. Moreover, the formalisms that are used to describe the reactive behavior are restricted to formalisms the semantics of which can be expressed by a state transition relation. For example, the semantics of Dynamic Z, proposed by Büssow and Grieskamp [13] for the formulation of properties, cannot be expressed by a state transition relation. It is not the aim here to present the semantics for $\mu\mathcal{SZ}$. The Kripke structures are introduced to $\mu\mathcal{SZ}$ and used only to define the semantic relation between $\mu\mathcal{SZ}$ specifications and CTL formulae.

Technically, the Kripke structures do not interfere with other underlying semantics. This would be the case only if the CTL semantics were defined on basis of another semantics, which is not the case.

Chapter 5

Syntax and Environment

In this Chapter an abstract Z syntax is introduced. This abstract syntax serves as the domain for the Z rewriting presented in Chapter 7 on page 97. Besides the syntax, an environment to store the context of a translation is introduced.

5.1 Reduced Z Syntax

The $\mu\mathcal{SZ}$ transformation and translation presented in this work is based on the abstract Z/$\mu\mathcal{SZ}$ syntax presented in this section. The syntax is similiar to the internal abstract syntax used in the Zeta tool. Yet, it does not reflect the complete Z language. It assumes some simplifications and does not include Z expressions that are not supported by the translation.

Like all mathematical definitions, the abstract syntax is defined using Z. In order to differentiate symbols of the actual Z (meta) language and symbols of the abstract syntax, the following notational conventions are used:

- Z expressions are typeset in black as usual: $\forall\, x : \mathbb{Z} \bullet x < y$
- Abstract syntax expressions are typeset in gray/red: $\forall\, var_{\varnothing}\ v : var_{\varnothing}\, \mathbb{Z} \bullet var_{\varnothing}\ x < var_{\varnothing}\ y$ is the abstract syntax representation of the above predicate. In Z, it is an expression of type *Pred*, which is subject to type checking.

5.1.1 Unsupported Expressions

In general, expressions that would demand extra (temporary) variables for the translation, are not supported, and expressions that merely blow up the theory without introducing new, interesting aspects, are omitted, too. Additionally, some syntactic simplifications are assumed to be already applied. The restrictions and assumptions are:

- Schema inclusion in declarations has been removed. The declaration and property part of the included schema is added to the declaration and property part of the including schema, respectively.

- Schema references in predicates have been replaced by the schema predicate and the predicates induced by the declaration.
- General set comprehension: $\{ d \mid p \bullet e \}$ has been simplified to $\{ x : X \mid (\exists d \mid p \bullet x = e) \}$, i. e. the "bullet" from set comprehensions has been removed.
- μ-Expressions: a predicate containing a μ-expression $p[\mu d \mid q \bullet E]$ has been simplified to $\exists_1 t \bullet (\exists d \mid q \bullet t = E) \land p[t]$. Note that the translation of quantors is very expensive, and that the usage of μ-expressions and general set displays is thus not very useful.

General set comprehension and μ-expressions can be removed in introducing existence quantors. Such quantors are to be translated to disjunctions: $\exists x : \{ x_1, \ldots, x_n \} \bullet p \equiv p[x_1/x] \lor \ldots \lor p[x_n/x]$. Thus, these constructs lead to an explosion of the size of the formulae.

To keep the translation simple, the parametrized given types are not supported. The branches of parametrized given types can carry values ($branch\langle\langle\mathbb{Z}\rangle\rangle$) of powerset types. The values can be finite or infinte. Parameterized type branches can be supported if the parameters of the branchs are finite. For example $branch\langle\langle\{ a, b, c \}\rangle\rangle$ can be rewritten to $branch\_a \mid branch\_b \mid branch\_c$.

5.1.2 Names

section *Name*

The identifiers that can be used in a Z specification are given by $NAME$. This includes names of given- and free-types, names of given-type branches, numbers, abbreviations, Statechart states, and variables.

$[NAME]$

A subset of $NAME$, $special$, is reserved for internal generated names, needed for the translation. It is assumed that these names do not appear in a specification.

The function $conc$ concatenates two variable names. It can be defined as: $conc(a, b) = \_\_a\_b$.

$prime$ computes the primed version of a variable: $prime(a) = a'$.

$BuiltinRel$ and $BuiltinFun$ denote built-in functions of the target model checker that do not need to be resolved.

$$
\begin{array}{|l}
prime : NAME \to NAME \\
conc : NAME \times NAME \rightarrowtail NAME \\
special, BuiltinRel, BuiltinFun : \mathbb{P}\, NAME \\
\hline
\mathsf{disjoint}\ \langle special, BuiltinRel, BuiltinFun, Number \rangle \\
\mathsf{ran}\ conc \subseteq special
\end{array}
$$

5.1.3 Numbers

The set of number identifiers *Number* $(0, 1, \ldots)$ is a subset of *NAME*. $\mathbb{Z}$ is the name of the given type, denoting the numbers. There is a bijection, *Num*, mapping numbers to number identifiers in the meta language.

$$Number : \mathbb{P}\, NAME$$
$$\mathbb{Z} : NAME$$

$$Num : \mathbb{N} \rightarrowtail\!\!\!\!\!\rightarrow Number$$

5.1.4 Ordering

An arbitrary ordering function is needed that transforms a set of names into a sequence. An actual tanslator can use an alphabetical ordering for instance.

$$order : \mathbb{F}\, NAME \rightarrow \mathrm{seq}\, NAME$$
$$order^{\sim} \subseteq \mathsf{ran} \land (\# \circ order \circ \#^{\sim}) \subseteq (\_ = \_)$$

The sequences must contain only elements from the original set and the number of sequence elements has to be equal to the number of set elements (duplications have to be avoided). Only finite sets of names can be mapped into sequences, since sequences are defined to be finite themselves, in the Z mathematical toolkit.

5.1.5 Expressions

section *Syntax* **parents** *zrm,Aux,Name,CTL*

The abstract syntax is build up using Z given types. The common Z identifiers and Z mixfix operators are used for the abstract syntax. With this, abstract syntax terms can be formulated in the same way as Z terms. In order to differentiate them from the meta-language, they are typeset in a different color. $\{x : var_{\varnothing}\ \mathbb{Z};\ y : var_{\varnothing}\ \mathbb{Z} \mid var_{\varnothing}\ x > var_{\varnothing}\ y\}$ is, for example, an abstract syntax term, corresponding to the Z expressions: $\{x, y : \mathbb{N} \mid x > y\}$. Only the Cartesian product and the function application cannot be presented in the same way as in Z, because the Cartesian product $(e_1 \times \ldots \times e_n)$ operator with arbitrary numbers of $\times$ cannot be expressed as a Z mixfix operator. For the "invisible operators", variable application and function application, operators have to be introduced in the abstract syntax. These are *var* v respectively $f\ \omega\ e$.

function $30(\{\_ \mid \_\})$
function $400((,))$
function 30 leftassoc $(\_\ \omega\ \_)$
function 30 leftassoc $(\_ == \_)$
function $30(\lambda\ \_ \mid \_ \bullet \_)$
function $30(\textbf{if}\ \_\ \textbf{then}\ \_\ \textbf{else}\ \_)$
function $30([\_ \mid \_])$
function 30 leftassoc $(\_\ .\ \_)$
function $1000(\{,\})$
function 30 leftassoc $(\_\ .\ \_)$
function $30([,])$

$$
\begin{array}{lll}
Expr ::= var\langle\!\langle NAME \times \mathrm{seq}\, Expr \rangle\!\rangle & & [\text{ (generic) variable application }] \\
\quad | \quad (\{\_ \mid \_\})\langle\!\langle Decl \times Pred \rangle\!\rangle & & [\text{ set comprehension }] \\
\quad | \quad (\{,,\})\langle\!\langle \mathrm{seq}_1\, Expr \rangle\!\rangle & & [\text{ set display }] \\
\quad | \quad \mathbb{P}\langle\!\langle Expr \rangle\!\rangle & & [\text{ powerset }] \\
\quad | \quad prod\langle\!\langle \mathrm{seq}_1\, Expr \rangle\!\rangle & & [\text{ Cartesian product }] \\
\quad | \quad ((,,))\langle\!\langle \mathrm{seq}_1\, Expr \rangle\!\rangle & & [\text{ tuple }] \\
\quad | \quad (\_ . \_)\langle\!\langle Expr \times \mathbb{N} \rangle\!\rangle & & [\text{ tuple selection }] \\
\quad | \quad (\_ . \_)\langle\!\langle Expr \times NAME \rangle\!\rangle & & [\text{ binding selection }] \\
\quad | \quad (\lambda \_ \mid \_ \bullet \_)\langle\!\langle Decl \times Pred \times Expr \rangle\!\rangle & & [\text{ lambda abstraction }] \\
\quad | \quad (\_\ \omega\ \_)\langle\!\langle Expr \times Expr \rangle\!\rangle & & [\text{ function application }] \\
\quad | \quad ([\_ \mid \_])\langle\!\langle Decl \times Pred \rangle\!\rangle & & [\text{ schema expression }] \\
\quad | \quad ([,,])\langle\!\langle \mathrm{seq}\, Bind \rangle\!\rangle & & [\text{ schema binding }] \\
\quad | \quad not_E\langle\!\langle Expr \rangle\!\rangle & & [\text{ schema calculus negation }] \\
\quad | \quad bin_E\langle\!\langle BinType \times Expr \times Expr \rangle\!\rangle & & [\text{ schema calculus (binary op.) }] \\
\quad | \quad (\mathbf{if}\ \_\ \mathbf{then}\ \_\ \mathbf{else}\ \_)\langle\!\langle Pred \times Expr \times Expr \rangle\!\rangle & & [\text{ conditional }]
\end{array}
$$

$$Bind ::= (\_ == \_)\langle\!\langle NAME \times Expr \rangle\!\rangle \quad [\text{ binding }]$$

The function b_{res} transforms a *valid* binding ($[a == 3, b == 4]$) into the variable assignment: $\{ a \mapsto 3, b \mapsto 4 \}$. The variable assignment is a partial function, mapping variable names to expressions. A binding is valid, if no variable occurs more than once.

$$
\begin{array}{l}
b_{res} : \mathsf{ran}([,,]) \pfun (NAME \pfun Expr) \\
b_{res}^{Inv} : (NAME \pfun Expr) \fun Expr \\
\hline
\forall\, bs : \mathrm{seq}\, Bind \mid \mathsf{ran}((\_ == \_)^{\sim} \circ bs) \in (NAME \pfun Expr) \bullet \\
\qquad b_{res}\ ((([,,])(bs)) = \mathsf{ran}((\_ == \_)^{\sim} \circ bs) \\
\forall\, as : NAME \pfun Expr \bullet \\
\qquad b_{res}^{Inv}\ as = ([,,])((\lambda\, n : \mathsf{dom}\ as \bullet n == as\ n) \circ order(\mathsf{dom}\ as))
\end{array}
$$

Variable application (var) carries, in addition to the name of the variable, the actualization of generic parameters. In the expression $\{1\} \cup \{2\}$, for example, application of $(\_ \cup \_)$ would be represented by $var(\_ \cup \_, \langle\, var(\mathbb{Z}, \varnothing)\, \rangle)$. The computation of the generic variable actualization is not discussed here. It is assumed that the actualization is already computed, by the Z type checker of the Zeta system. For non-generic variables (for example $\mathbb{Z}$), the actualization list is empty. This is abbreviated with the $var_{\varnothing}$ function:

$$var_{\varnothing} == \lambda\, v : NAME \bullet var(v, \varnothing)$$

5.1.6 Predicates

$$
\begin{array}{ll}
\mathbf{function}\ 100\ \mathrm{leftassoc}\ (\_ \in \_) & \mathbf{function}\ 30\ \mathrm{leftassoc}\ (\_ = \_) \\
\mathbf{function}\ 30(\mathcal{Q}\_\ \_ \mid \_ \bullet \_) & \mathbf{function}\ 20\ \mathrm{leftassoc}\ (\_ \wedge \_) \\
\mathbf{function}\ 21\ \mathrm{leftassoc}\ (\_ \vee \_) & \mathbf{function}\ 22\ \mathrm{leftassoc}\ (\_ \Rightarrow \_) \\
\mathbf{function}\ 23\ \mathrm{leftassoc}\ (\_ \Leftrightarrow \_) & \mathbf{function}\ 30(\forall \_ \mid \_ \bullet \_)
\end{array}
$$

$$\textbf{function}\ 30(\exists \_ \mid \_ \bullet \_)$$

$$
\begin{array}{lll}
\mathit{BinType} & ::= \mathit{and} \mid \mathit{or} \mid \mathit{iff} \mid \mathit{implies} & [\text{ Boolean operators }] \\
\mathit{QuantType} & ::= \mathit{forall} \mid \mathit{exists} & [\text{ quantor types }] \\
\mathit{Pred} & ::= (\_ = \_)\langle\!\langle \mathit{Expr} \times \mathit{Expr} \rangle\!\rangle & [\text{ equality }] \\
& \mid\ (\_ \in \_)\langle\!\langle \mathit{Expr} \times \mathit{Expr} \rangle\!\rangle & [\text{ element test }] \\
& \mid\ \neg \langle\!\langle \mathit{Pred} \rangle\!\rangle & [\text{ negation }] \\
& \mid\ \mathit{bin}\langle\!\langle \mathit{BinType} \times \mathit{Pred} \times \mathit{Pred} \rangle\!\rangle & [\text{ binary operator }] \\
& \mid\ (\mathcal{Q}\_\ \_ \mid \_ \bullet \_)\langle\!\langle \mathit{QuantType} \times \mathit{Decl} \times \mathit{Pred} \times \mathit{Pred} \rangle\!\rangle & \\
& & [\text{ quantor }] \\
& \mid\ \mathit{true} \mid \mathit{false} & [\text{ truth and falsehood }]
\end{array}
$$

As described in section 4.8 on page 49, CTL properties appear as normal predicates in specifications (see e. g. Figure 4.7 on page 51). In abstract syntax, presented here, they thus appear as element tests and function applications. Out of convenience, the function ctl^{meta} is introduced. It translates CTL properties from the meta language to CTL formulae over Pred (without definition):

$$\mid \mathsf{ctl}^{meta} : \mathit{Pred} \rightarrowtail \mathrm{CTL}[\mathit{Pred}]$$

5.1.7 Syntax Expression Constructors

Some convenience functions are introduced for building predicates quickly:

$$
\begin{array}{ll}
(\_ \wedge \_) & == \lambda\, p, q : \mathit{Pred} \bullet \mathit{bin}(\mathit{and}, p, q) \\
(\_ \vee \_) & == \lambda\, p, q : \mathit{Pred} \bullet \mathit{bin}(\mathit{or}, p, q) \\
(\_ \Rightarrow \_) & == \lambda\, p, q : \mathit{Pred} \bullet \mathit{bin}(\mathit{implies}, p, q) \\
(\_ \Leftrightarrow \_) & == \lambda\, p, q : \mathit{Pred} \bullet \mathit{bin}(\mathit{iff}, p, q) \\
(\forall \_ \mid \_ \bullet \_) & == \lambda\, d : \mathit{Decl};\ q, p : \mathit{Pred} \bullet \mathcal{Q}_{\mathit{forall}}\ d \mid p \bullet q \\
(\exists \_ \mid \_ \bullet \_) & == \lambda\, d : \mathit{Decl};\ q, p : \mathit{Pred} \bullet \mathcal{Q}_{\mathit{exists}}\ d \mid p \bullet q
\end{array}
$$

For a set of predicates, constructor functions are defined that compute the disjunction ($\bigvee$) and the conjunction ($\bigwedge$) of the predicates: $\bigvee \{ p_1, \ldots p_n \} = p_1 \vee \ldots \vee p_n$

For the empty set, these functions default to true or false, respectively.

A set of guarded expressions, i. e. tuples of predicates and expressions, are reduced to a single case expression by $\mathit{case\_reduce}$. One guarded expression is chosen arbitrarily as the default value.

$$
\begin{array}{l}
\bigvee == \mathsf{setreduce}\,(\_ \vee \_) \cup \{ \varnothing \mapsto \mathit{false} \} \\
\bigwedge == \mathsf{setreduce}\,(\_ \wedge \_) \cup \{ \varnothing \mapsto \mathit{true} \} \\
\mathit{case\_reduce} : \mathbb{P}_1(\mathit{Pred} \times \mathit{Expr}) \rightarrow \mathit{Expr} \\
\hline
\forall\, p : \mathit{Pred};\ e : \mathit{Expr};\ PE : \mathbb{P}_1(\mathit{Pred} \times \mathit{Expr}) \bullet \\
\quad \mathit{case\_reduce}\{ p \mapsto e \} = e \wedge \\
(p \mapsto e \notin PE \Rightarrow \mathit{case\_reduce}(\{ p \mapsto e \} \cup PE) = \\
\quad \textbf{if}\ p\ \textbf{then}\ e\ \textbf{else}\ \mathit{case\_reduce}\ PE)
\end{array}
$$

5.1.8 Declarations

This abstract syntax introduces a uniform declarations that contains variable declaration and abbreviations as well as free- and given-type declarations. Note that in Z, type-declarations may only appear in *zed-boxes*. This restriction is relaxed in the abstract syntax.

$$\begin{array}{ll}
\textbf{function}\ 40\ \text{leftassoc}\ (_ : _) & \textbf{function}\ 30\ \text{leftassoc}\ (_;\ _) \\
\textbf{function}\ 30\ \text{leftassoc}\ (_ == _) & \textbf{function}\ 30\ \text{leftassoc}\ (_ \mid _) \\
\textbf{function}\ 20\ \text{leftassoc}\ (_ ::= _) & \textbf{function}\ 1000([_ \mid _]_) \\
\textbf{function}\ 30\ \text{leftassoc}\ (_;\ _) &
\end{array}$$

$$\begin{array}{lll}
Branch ::= Const\langle\langle NAME \rangle\rangle & & \\
\quad \mid (_ \mid _)\langle\langle Branch \times Branch \rangle\rangle & & \\
Decl \quad ::= (_ : _)\langle\langle NAME \times Expr \rangle\rangle & [\ \text{variable declaration}\ ] & \\
\quad \mid (_ == _)\langle\langle NAME \times Expr \rangle\rangle & [\ \text{abbrevation}\ ] & \\
\quad \mid (_ ::= _)\langle\langle NAME \times Branch \rangle\rangle & [\ \text{free type}\ ] & \\
\quad \mid (_;\ _)\langle\langle Decl \times Decl \rangle\rangle & [\ \text{separator}\ ] & \\
\quad \mid Given\langle\langle NAME \rangle\rangle & [\ \text{givent type}\ ] & \\
\quad \mid ex\langle\langle Expr \rangle\rangle & [\ \text{schema}\ ] &
\end{array}$$

5.1.9 Characteristic Tuple

The function ct computes the characteristic tuple of a declaration. The characteristic tuple is the sequence of variable declarations. Abbreviations and type declarations are ignored. If schema calculus expressions (e. g. $S \land T$) appear in a declaration, variables can be declared in both involved schemata. In these cases, the second declaration is ignored.

$$ct : Decl \to \text{seq}(NAME \times Expr)$$

$$\begin{array}{l}
\forall\, n : NAME;\ e, e' : Expr;\ b : Branch;\ d, d' : Decl;\ bt : BinType;\ p : Pred \bullet \\
\quad ct(n : e) = \langle n \mapsto e \rangle \land \\
\quad ct(n == e) = \varnothing \land \\
\quad ct(n ::= b) = \varnothing \land \\
\quad ct(d;\ d') = ct\ d \frown ct\ d' \land \\
\quad ct(Given\, n) = \varnothing \land \\
\quad ct(ex([d \mid p])) = ct(d) \land \\
\quad ct(ex(bin_E(bt, e, e'))) = ct(ex(e)) \frown (ct(ex(e'))\rhd \\
\qquad\qquad\qquad\qquad (\text{dom}(\text{ran}(ct(ex(e)))) \times Expr))
\end{array}$$

There is also a reverse function ct^{inv} that translates a characteristic tuple into a canonical declaration.

$$ct^{inv} : (NAME \leftrightarrow Expr) \to Decl$$

$$\begin{array}{l}
\forall\, as : NAME \nrightarrow Expr \bullet \\
\quad ct^{inv}\ as = \text{setreduce}(_;\ _)((_ : _)(\!| as |\!))
\end{array}$$

A declaration always implies an additional predicate. The declared variables have to be members of their type expressions ($v : e$ implies that $v \in e$). *declPred* computes this predicate for a given declaration (including schema expressions).

$$
\begin{array}{|l}
declPred : Decl \to Pred \\
\hline
\forall\, n : NAME;\ e, e' : Expr;\ b : Branch;\ d, d' : Decl;\ bt : BinType;\ p : Pred \bullet \\
\quad declPred(n\colon e) = var_{\varnothing}\ n \in e \land \\
\quad declPred(n == e) = true \land \\
\quad declPred(n ::= b) = true \land \\
\quad declPred(d;\ d') = declPred\ d \land declPred\ d' \land \\
\quad declPred(Given n) = true \land \\
\quad declPred(ex([d \mid p])) = declPred(d) \land p \land \\
\quad declPred(ex(bin_E(bt, e, e'))) = bin(bt, declPred(ex(e)), declPred(ex(e'))) \\
\end{array}
$$

5.1.10 Specification

In $\mu\mathcal{SZ}$, schemata can be annotated by roles they have in a $\mu\mathcal{SZ}$ class. *Stype* defines the different roles. An ordinary Z schema has the type *Plain*. An additional role *Fairness* is introduced, to handle fairness constraints. Fairness constraints are indispensible for CTL model checking.

$$Stype ::= Plain \mid Data \mid Port \mid Init \mid Property \mid Transition \mid Fairness$$

The declaration of formal generic variables is a comma separated sequence of names:

$$| \ GenFormals == \mathrm{seq}\, NAME$$

$$
\begin{array}{lll}
Spec ::= ([\_ \mid \_]\_)\langle\!\langle Decl \times Pred \times GenFormals \rangle\!\rangle & & \text{[axiomatic definition]} \\
\quad \mid Schema\langle\!\langle NAME \times Stype \times Decl \times Pred \rangle\!\rangle & & \text{[schema declaration]} \\
\quad \mid Statechart\langle\!\langle State \rangle\!\rangle & & \text{[Statechart]} \\
\quad \mid Class\langle\!\langle NAME \times Spec \rangle\!\rangle & & \text{[class]} \\
\quad \mid (\_;\ \_)\langle\!\langle Spec \times Spec \rangle\!\rangle & & \text{[separator]}
\end{array}
$$

$([\_ \mid \_]_{\varnothing})$ is a convienience function to build non-generic schema expressions:
function $1000([\_ \mid \_]_{\varnothing})$

$$| \ [\_ \mid \_]_{\varnothing} == \lambda\, d : Decl;\ p : Pred \bullet [d \mid p]_{\varnothing}$$

5.1.11 Statecharts

The Statemate Extractor delivers a simplified Statechart representation. This representation is used here. The label of transition consists of one predicate—the guard—and a set of predicates representing the actions. That means, a Statemate label `I=4/I:=5; J:=I+1` is represented by $(I = 4, \{\, I' = 4, J' = J + 1 \,\})$. Note that the actions cannot simply be combined by conjunctions, since racing can occur among them. If the transition consists of several segments (combined by connectors), the guard is the conjunction of the guards of all transition segments and the action is the set of actions of the transitions segments. For a transition without guard, the guard is *true*. For a transition without actions, the empty set is used.

$$Label == Pred \times \mathbb{P}\, Pred$$

Transitions are presented as full compound transitions in the abstract syntax. They consist of a set of source state names, a set of target state names and a label. Note that the least common ancestor of a transition's source and target states must be an and-state.

$$Trans == \mathbb{F}\, NAME \times \mathbb{F}\, NAME \times Label$$

$$\begin{aligned} State ::= & \; Basic\langle\langle NAME \rangle\rangle \\ | & \; And\langle\langle NAME \times \mathbb{F}\, State \rangle\rangle \\ | & \; Xor\langle\langle NAME \times \mathbb{F}\, Trans \times \mathbb{F}\, State \rangle\rangle \end{aligned}$$

defaultState is used for $\mu\mathcal{SZ}$ classes without Statechart.

$$| \; defaultStateName : NAME$$

$$| \; defaultState == Basic\; defaultStateName$$

Some selector functions are defined for Statecharts:

$$\begin{aligned} stName &== (\lambda\, s : \mathsf{ran}\, Basic \bullet (Basic^{\sim})\, s) \cup \\ &\quad (\lambda\, s : \mathsf{ran}\, And \bullet ((And^{\sim})\, s).1) \cup \\ &\quad (\lambda\, s : \mathsf{ran}\, Xor \bullet ((Xor^{\sim})\, s).1) && [\text{ state's name }] \\ subs &== \{\, s : \mathsf{ran}\, And;\; s' : State \mid s' \in ((And^{\sim})\, s).2 \,\} \cup \\ &\quad \{\, s : \mathsf{ran}\, Xor;\; s' : State \mid s' \in ((Xor^{\sim})\, s).3 \,\} \\ & && [\text{ super-state's substates }] \\ sources &== \lambda\, t : Trans \bullet t.1 && [\text{ transition's sources }] \\ targets &== \lambda\, t : Trans \bullet t.2 && [\text{ transition's targets }] \\ label &== \lambda\, t : Trans \bullet t.3 && [\text{ transition's label }] \\ guard &== \lambda\, l : Label \bullet l.1 && [\text{ transition's guard }] \\ actions &== \lambda\, l : Label \bullet l.2 && [\text{ transition's actions }] \end{aligned}$$

5.1.12 Built-Ins

There are certain functions that are handled by the model checker itself, or which can be assumed to be replaced in later translation steps. *Built-ins* are not replaced during the translation.

$$\begin{aligned} & neq, leq, geq, less, greater, .., plus, minus : NAME \\ \hline & BuiltinRel = \{\, neq, leq, geq, less, greater \,\} \\ & BuiltinFun = \{\, .., plus, minus \,\} \end{aligned}$$

function 30 leftassoc $(\_ + \_)$ **function** 30 leftassoc $(\_ - \_)$

function 30 leftassoc $(\_ < \_)$ **function** 30 leftassoc $(\_ \leq \_)$

function 30 leftassoc $(\_ > \_)$ **function** 30 leftassoc $(\_ \geq \_)$

function 30$(_ \neq [\,_\,] _)$ **function** 60 leftassoc $(\_ .. \_)$

$$
\begin{array}{ll}
(\_ \neq [\_]\ \_) & == (\lambda\, e, E, e' : Expr \bullet (e, e') \in var(neq, \langle E \rangle)) \\
(\_ + \_) & == (\lambda\, e, e' : Expr \bullet (var_{\varnothing}\ plus)\ \omega\ (e, e')) \\
(\_ - \_) & == (\lambda\, e, e' : Expr \bullet (var_{\varnothing}\ minus)\ \omega\ (e, e')) \\
(\_ < \_) & == (\lambda\, e, e' : Expr \bullet (e, e') \in (var_{\varnothing}\ less)) \\
(\_ \leq \_) & == (\lambda\, e, e' : Expr \bullet (e, e') \in (var_{\varnothing}\ leq)) \\
(\_ > \_) & == (\lambda\, e, e' : Expr \bullet (e, e') \in (var_{\varnothing}\ greater)) \\
(\_ \geq \_) & == (\lambda\, e, e' : Expr \bullet (e, e') \in (var_{\varnothing}\ geq)) \\
(\_ .. \_) & == (\lambda\, e, e' : Expr \bullet (var_{\varnothing}\ ..)\ \omega\ (e, e'))
\end{array}
$$

5.1.13 Term Transformations

In the following sections, some rewriting is done on the abstract $\mu\mathcal{SZ}$ syntax. In order to easily formulate term rewriters that change only particular sub-terms, some auxiliar functions are defined. These functions get partial transformation functions as input ($TRANS$) and construct total transformers from them.

For example, to replace all variable applications in an expression with its primed counterparts (x to x'), the following definition can be used:

$$
\begin{array}{l}
trans_e[\, e == \{\, n : NAME \bullet var_{\varnothing}\ n \mapsto var_{\varnothing}(prime\ n)\,\}, \\
\qquad p == \varnothing, \\
\qquad d == \{\, n : NAME;\ e : Expr \bullet n{:}\, e \mapsto prime\ n{:}\, e\,]
\end{array}
$$

$$
\begin{array}{|l}
\hline
TRANS \\
e : Expr \nrightarrow Expr \\
p : Pred \nrightarrow Pred \\
d : Decl \nrightarrow Decl \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
trans_e : TRANS \rightarrow Expr \rightarrow Expr \\
\hline
\exists\, trans_{e0} : TRANS \rightarrow Expr \rightarrow Expr \bullet \\
\quad \forall f : TRANS \bullet \\
\quad \forall e, e' : Expr;\ \vec{e} : \mathrm{seq}_1\ Expr;\ v : NAME;\ p : Pred;\ d : Decl;\ i : \mathbb{Z} \bullet \\
\qquad trans_e\, f = (trans_{e0}\, f) \oplus (\lambda\, e : \mathrm{dom}\, f.e \bullet f.e\ e) \wedge \\
\qquad trans_{e0}\, f\, (var(v, \vec{e})) = var(v, (trans_e\, f) \circ \vec{e}) \wedge \\
\qquad trans_{e0}\, f\, (\{d \mid p\}) = \{(trans_d\, f\, d) \mid (trans_p\, f\, p)\} \wedge \\
\qquad trans_{e0}\, f\, (\{,,\}\vec{e}) = \{,,\}((trans_e\, f) \circ \vec{e}) \wedge \\
\qquad trans_{e0}\, f\, (\mathbb{P}\ e) = \mathbb{P}((trans_e\, f)(e)) \wedge \\
\qquad trans_{e0}\, f\, (prod\ \vec{e}) = prod((trans_e\, f) \circ \vec{e}) \wedge \\
\qquad trans_{e0}\, f\, ((,,)\vec{e}) = (,,)((trans_e\, f) \circ \vec{e}) \wedge \\
\qquad trans_{e0}\, f\, (e\ .\ i) = (trans_e\, f)(e)\ .\ i \wedge \\
\qquad trans_{e0}\, f\, (\lambda\, d \mid p \bullet e) = \lambda(trans_d\, f\, d) \mid (trans_p\, f\, p) \bullet (trans_e\, f\, e) \wedge \\
\qquad trans_{e0}\, f\, (e\ \omega\ e') = (trans_e\, f\, e)\ \omega\ (trans_e\, f\, e') \wedge \\
\qquad trans_{e0}\, f\, (\mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e') = \\
\qquad\quad \mathbf{if}(trans_p\, f\, p)\ \mathbf{then}(trans_e\, f\, e)\ \mathbf{else}(trans_e\, f\, e') \\
\hline
\end{array}
$$

$$
\begin{array}{l}
trans_p : TRANS \to Pred \to Pred \\
\hline
\exists\, trans_{p_0} : TRANS \to Pred \to Pred \bullet \\
\quad \forall f : TRANS \bullet \\
\quad \forall p, p' : Pred;\ e, e' : Expr;\ d : Decl;\ b : BinType;\ qt : QuantType \bullet \\
\qquad trans_p\, f = (trans_{p_0}\, f) \oplus (\lambda p : \mathsf{dom}\, f.p \bullet f.p\ p) \land \\
\qquad trans_{p_0}(f)(e = e') = (trans_e\, f\ e) = (trans_e\, f\ e') \land \\
\qquad trans_{p_0}(f)(e \in e') = (trans_e\, f\ e) \in (trans_e\, f\ e') \land \\
\qquad trans_{p_0}(f)(\neg\ p) = \neg\ (trans_p\, f\ p) \land \\
\qquad trans_{p_0}(f)(bin(b, p, p')) = bin(b, trans_p\, f\ p, trans_p\, f\ p') \land \\
\qquad trans_{p_0}(f)(\mathcal{Q}_{qt}\ d \mid p \bullet p') = \mathcal{Q}_{qt}\ trans_d\, f\ d \mid trans_p\, f\ p \bullet trans_p\, f\ p' \land \\
\quad trans_{p_0}\, f\ true = true \land \\
\qquad trans_{p_0}\, f\ false = false
\end{array}
$$

$$
\begin{array}{l}
trans_d : TRANS \to Decl \to Decl \\
\hline
\exists\, trans_{d_0} : TRANS \to Decl \to Decl \bullet \\
\quad \forall f : TRANS \bullet \\
\quad \forall d, d' : Decl;\ v : NAME;\ e : Expr;\ b : Branch \bullet \\
\qquad trans_d\, f = (trans_{d_0}\, f) \oplus (\lambda d : \mathsf{dom}\, f.d \bullet f.d\ d) \land \\
\qquad trans_{d_0}(f)((\_ : \_)(v, e)) = (\_ : \_)(v, trans_e\, f\ e) \land \\
\qquad trans_{d_0}(f)(v == e) = v == (trans_e\, f\ e) \land \\
\qquad trans_{d_0}(f)(v ::= b) = v ::= b \land \\
\qquad trans_{d_0}(f)(d;\ d') = (trans_d\, f\ d);\ (trans_d\, f\ d') \land \\
\qquad trans_{d_0}(f)(Given\ v) = Given\ v
\end{array}
$$

Some abbreviations are defined:

$$
\begin{array}{l}
transExpr == (\lambda f : Expr \nrightarrow Expr \bullet trans_e(\mu\, TRANS \mid e = f \land p = \varnothing \land d = \varnothing)) \\
transExpr_P == (\lambda f : Expr \nrightarrow Expr \bullet trans_p(\mu\, TRANS \mid e = f \land p = \varnothing \land d = \varnothing)) \\
transPred == (\lambda f : Pred \nrightarrow Pred \bullet trans_p(\mu\, TRANS \mid e = \varnothing \land p = f \land d = \varnothing)) \\
transDecl == (\lambda f : Decl \nrightarrow Decl \bullet trans_d(\mu\, TRANS \mid e = \varnothing \land p = \varnothing \land d = f))
\end{array}
$$

5.2 Environment

In order to perform computability analysis and rewriting, information on the context of the specification is needed. The relevant context consists of the following symbols:

- *Given types:* Type definition such as [*NewType*].
- *Free types:* Types that are defined by a free construction. In Z, these types are only abbreviation for given types and the declarations of the respective constructor variables. The free construction property is insured by an implicit predicate. The free construction property is quite important for the translation, because free types can be finite. Therefore, the environment distinguishes between given and free types.
- *Constant variables:* Variables defined in axiomatic or generic definitions. As described in section 4.8.2 on page 51, these variables are treated as *constants* and need special treatment for model checking.

- *Data and port variables:* Data and port variables are declared in the respective schemata of a class. They constitute the class' data space.
- *Abbreviations:* Abbreviations of expressions can be defined in all declarations. If they are defined in axiomatic or generic definitions, they are visible throughout the specification (section, class).
- *Schema definition:* The schemata that are defined in the specification. With $\mu\mathcal{SZ}$, schema definitions in classes have different *roles*, e. g. DATA, PORT, INIT, etc. These schemata are of special interest for the translation, since they define the Kripke structure as described in section 4.8 on page 49. Schema definitions are important in two cases: (1) if they have a role, contributing to the class semantics, i. e. DATA, PORT, INIT, TRANS, or PROPERTY; or (2) if they are referenced somewhere else in the specification.

  For the first case, only the names and types of data variables need to be collected. For the second case, it is assumed that schema references are already resolved. For this reason, it is not necessary to store the complete schema definitions in the environment.
- *Classes:* In $\mu\mathcal{SZ}$, classes can be defined and used through enrichment and configurations.
- *Root/Parent State:* The root Statechart state of a class and the state hierarchy.
- *Built-ins:* Built-in functions of the model checker (e. g. + and −). These symbols do not belong to the context of a specification but to the context of the translation process.

section *Environment* **parents** *Syntax, TypeDecl*

Env

$$
\begin{array}{l}
free, given, data, port, const, builtins : \mathbb{F}\ NAME \\
type : NAME \pfun \mathsf{seq}\ NAME \times \mathsf{Type} \\
defs : NAME \pfun \mathsf{seq}\ NAME \times Expr \\
root : State \\
params : \mathsf{seq}\ NAME \\
\hline
\mathsf{disjoint}\ (\langle free, given, data, port, const, builtins \rangle) \\
free \cup given \cup data \cup port \cup port \cup const \cup builtins \subseteq \mathsf{dom}\ type \\
\forall v : free \cup given \bullet (type\ v).1 = \varnothing \land type\ v \in \{ \varnothing \} \times \mathsf{ran}\ \mathsf{basic}
\end{array}
$$

The empty environment $\varnothing_{Env}$ contains the built-in operators already. According to the invariant of the *Env* schema, their types (see section 7.4 on page 103) have to be defined, too. Note that *neq*, is a generic function.

$$
\begin{array}{l}
\varnothing_{Env} : Env \\
T : NAME \\
\hline
\varnothing_{Env}.free = \varnothing_{Env}.given = \varnothing_{Env}.port = \varnothing_{Env}.const = \varnothing \\
\varnothing_{Env}.builtins = \{\, neq, leq, geq, less, greater, .., plus, minus \,\} \\
\varnothing_{Env}.type = \{\, neq \mapsto (\langle T \rangle, \mathsf{power}(\mathsf{prod}\langle \mathsf{basic}(T, NAME), \mathsf{basic}(T, NAME)\rangle)), \\
\qquad\qquad leq \mapsto (\varnothing, \mathsf{power}(\mathsf{prod}\langle \mathsf{number}, \mathsf{number}\rangle)), \\
\qquad\qquad geq \mapsto (\varnothing, \mathsf{power}(\mathsf{prod}\langle \mathsf{number}, \mathsf{number}\rangle)), \\
\qquad\qquad less \mapsto (\varnothing, \mathsf{power}(\mathsf{prod}\langle \mathsf{number}, \mathsf{number}\rangle)), \\
\qquad\qquad greater \mapsto (\varnothing, \mathsf{power}(\mathsf{prod}\langle \mathsf{number}, \mathsf{number}\rangle)), \\
\qquad\qquad .. \mapsto (\varnothing, \mathsf{fun}(\langle \mathsf{number}, \mathsf{number}\rangle, \mathsf{power}\ \mathsf{number})), \\
\qquad\qquad plus \mapsto (\varnothing, \mathsf{fun}(\langle \mathsf{number}, \mathsf{number}\rangle, \mathsf{number})), \\
\qquad\qquad minus \mapsto (\varnothing, \mathsf{fun}(\langle \mathsf{number}, \mathsf{number}\rangle, \mathsf{number}))\} \\
\qquad\qquad \cup \\
\qquad\qquad \{\, n : Number \bullet n \mapsto (\varnothing, \mathsf{basic}(\mathbb{Z}, \{\, n \,\}))\,\} \\
\varnothing_{Env}.defs = \varnothing \\
\varnothing_{Env}.root = defaultState \\
\varnothing_{Env}.params = \varnothing
\end{array}
$$

The type of a generic variable contains type variables. For example the type of a variable $x : \mathbb{P}\, T$, where T is a formal generic type parameter, is $\mathsf{basic}(T, NAME)$. If the generic variable x is used, the formal type parameters are instantiated by actual types. A type instantiation assigns types to the formal parameters: $ftoa : NAME \nrightarrow \mathsf{Type}$. The function *trans* instantiates a generic type with a given type instantiation.

$$
\begin{array}{l}
trans : (NAME \nrightarrow \mathsf{Type}) \rightarrow \mathsf{Type} \rightarrow \mathsf{Type} \\
\hline
\forall\, ftoa : NAME \nrightarrow \mathsf{Type} \bullet \\
\quad (\forall\, v : NAME;\ V : \mathbb{P}\, NAME \bullet \\
\quad trans(ftoa)(\mathsf{basic}(v, V)) = \mathbf{if}\ v \in \mathsf{dom}\, ftoa\ \mathbf{then}\ ftoa\, v\ \mathbf{else}\ \mathsf{basic}(v, V)) \land \\
\quad (\forall\, \tau : \mathsf{Type} \bullet trans(ftoa)(\mathsf{power}\, \tau) = \mathsf{power}(trans(ftoa)(\tau))) \land \\
\quad (\forall\, \vec{\tau} : \mathrm{seq}_1\, \mathsf{Type} \bullet trans(ftoa)(\mathsf{prod}\, \vec{\tau}) = \mathsf{prod}((trans\, ftoa) \circ \vec{\tau})) \land \\
\quad (\forall\, \vec{\tau} : \mathrm{seq}_1\, \mathsf{Type};\ \tau : \mathsf{Type} \bullet \\
\quad trans(ftoa)(\mathsf{fun}(\vec{\tau}, \tau)) = \mathsf{fun}((trans\, ftoa) \circ \vec{\tau}, trans(ftoa)(\tau)))
\end{array}
$$

getType retrieves the type of a variable, with the given assignment of the generic parameters. The assignment is empty, if the variable is not generic. The actual generic parameters are given as a sequence of types ($a : \mathrm{seq}\, \mathsf{Type}$) that has to match the sequence of formal parameters stored with the type: $\#a = \#(\mathcal{E}.type\ v).1$. A function is built from the sequences by zipping them together and dropping the index: $\mathsf{ran}(\mathsf{zip}(first(\mathcal{E}.type\ v), a))$.

$$
\begin{array}{l}
getType : Env \rightarrow (NAME \times \mathrm{seq}\, \mathsf{Type}) \nrightarrow \mathsf{Type} \\
\hline
\forall\, \mathcal{E} : Env;\ v : NAME;\ a : \mathrm{seq}\, \mathsf{Type} \bullet \\
\quad ((v, a) \in \mathsf{dom}(getType\, \mathcal{E}) \Leftrightarrow v \in \mathsf{dom}\, \mathcal{E}.type \land \#a = \#(\mathcal{E}.type\ v).1) \land \\
\quad getType(\mathcal{E})(v, a) = trans(\mathsf{ran}(\mathsf{zip}((\mathcal{E}.type\ v).1, a)))(\mathcal{E}.type\ v).2
\end{array}
$$

getDef retrieves the definition of an abbreviation. If the abbreviation is generic, the actual parameters are given as a sequence of expressions. Occurrences of generic parameters in the defining expression have to be actualized in a similar way to the actualization of generic types in *getType*. The definition of *getDef* is quite similar to *getType*. It is therefore skipped here.

getType	Get the type of a data, port or constant variable.
getDef	Get the definition of an abbreviation.
getData, *getPort*	Get the names of the data or port variables of the current class.
getConst	Get the names of the constant variables.
getParams	Get the generic parameters of the current environment. Returns the list of names in a generic definition and the empty set otherwise.
getGiven	Get the set of given types.
getFree	Get the set of free types with their definitions.
getBuiltins	Get the set of built-in functions.
getRoot	Get the root Statechart state. With this, the Statechart of a class is accessed.
getParent	Get the parent state of a given Statechart state.
newName	Compute some name that is not used in the environment.

Table 5.1: Environment Retrieval Functions

$$
\begin{array}{l}
getDef : Env \rightarrow NAME \times \operatorname{seq} Expr \nrightarrow Expr \\
\hline
\forall \mathcal{E} : Env;\ v : NAME;\ a : \operatorname{seq} Expr \bullet \\
\quad ((v, a) \in \mathsf{dom}(getDef\ \mathcal{E}) \Leftrightarrow v \in \mathsf{dom}\ \mathcal{E}.defs \land \\
\qquad\qquad\qquad\qquad\qquad\qquad \#a = \#((\mathcal{E}.defs\ v).1)) \land \\
\quad getDef\,(\mathcal{E})(v, a) = \\
\qquad (transExpr(\mathsf{ran}(\mathsf{zip}(var_{\varnothing} \circ (\mathcal{E}.defs\ v).1, a)))((\mathcal{E}.defs\ v).2))
\end{array}
$$

Various retrieval functions, see Table 5.1 for explanations:

$$
\begin{array}{l}
getData, getPort : Env \rightarrow \mathbb{F}\ NAME \\
getParams : Env \rightarrow \operatorname{seq} NAME \\
getGiven : Env \rightarrow \mathbb{P}\ NAME \\
getFree : Env \rightarrow \mathbb{F}\ NAME \\
getBuiltins : Env \rightarrow \mathbb{F}\ NAME \\
getSymbols : Env \rightarrow \mathbb{F}\ NAME \\
\hline
\forall \mathcal{E} : Env \bullet \\
\quad getData\ \mathcal{E} = \mathcal{E}.data \land \\
\quad getPort\ \mathcal{E} = \mathcal{E}.port \land \\
\quad getParams\ \mathcal{E} = \mathcal{E}.params \land \\
\quad getGiven\ \mathcal{E} = \mathcal{E}.given \land \\
\quad getFree\ \mathcal{E} = \mathcal{E}.free \land \\
\quad getBuiltins\ \mathcal{E} = \mathcal{E}.builtins \land \\
\quad getSymbols\ \mathcal{E} = \mathsf{dom}\ \mathcal{E}.type \cup \mathsf{dom}\ \mathcal{E}.defs
\end{array}
$$

Select a not yet declared name:

$$
\begin{array}{l}
newName : Env \rightarrow NAME \\
\hline
\forall \mathcal{E} : Env \bullet newName\ \mathcal{E} \notin (getSymbols\ \mathcal{E})
\end{array}
$$

Get the root state (*getRoot*), all states (*getStates*), and the parent state function (*getParent*) of the current class' Statechart. The parent state function returns for some state name the parent state, if the state name exists.

$$
\begin{array}{l}
getRoot : Env \to State \\
getStates : Env \to \mathbb{F}\, State \\
getParent : Env \to NAME \nrightarrow State \\
\hline
\forall\, \mathcal{E} : Env \bullet \\
\quad getRoot\, \mathcal{E} = \mathcal{E}.root \land \\
\quad getStates\, \mathcal{E} = (subs^*)(\!|\{\, getRoot\, \mathcal{E} \,\}|) \land \\
\quad \mathsf{dom}(getParent\, \mathcal{E}) = stName(\!|getStates\ \mathcal{E} \setminus \{\, \mathcal{E}.root \,\}|) \land \\
\quad (\forall\, n : \mathsf{dom}(getParent\, \mathcal{E}) \bullet \\
\qquad getParent\ \mathcal{E}\ n = (\mu\, s : getStates\, \mathcal{E} \mid n \in (stName \circ subs)(\!|\{\, s \,\}|)))
\end{array}
$$

Add some declarations to the environment. A sequence of generic parameters can be given in order to declare the symbols to be generic. The declarations are given by a function, mapping names to their types.

$$
\begin{array}{l}
addDecls : Env \to \mathrm{seq}\, NAME \times (NAME \nrightarrow \mathsf{Type}) \to Env \\
\hline
\forall\, Env;\ \mathcal{E} : Env;\ params' : \mathrm{seq}\, NAME;\ vars : NAME \nrightarrow \mathsf{Type} \bullet \\
\quad addDecls(\theta Env)(params', vars) = \mathcal{E} \Leftrightarrow \\
\qquad (\exists\, type == \mathcal{E}.type \oplus ((\lambda\, \tau : \mathsf{Type} \bullet params' \mapsto \tau) \circ vars) \bullet \theta Env = \mathcal{E})
\end{array}
$$

Add some abbreviations to the environment. A sequence of generic parameters can be given in order to declare the abbreviations to be generic. The abbreviations are given by a function mapping names to their defining expressions.

$$
\begin{array}{l}
addDefs : Env \to \mathrm{seq}\, NAME \times (NAME \nrightarrow Expr) \to Env \\
\hline
\forall\, Env;\ \mathcal{E} : Env;\ params' : \mathrm{seq}\, NAME;\ vars : NAME \nrightarrow Expr \bullet \\
\quad addDefs(\theta Env)(params', vars) = \mathcal{E} \Leftrightarrow \\
\quad (\exists\, defs == \mathcal{E}.defs \oplus ((\lambda\, e : Expr \bullet params' \mapsto e) \circ vars) \bullet \theta Env = \mathcal{E})
\end{array}
$$

Add generic parameters to the environment.

$$
\begin{array}{l}
addParams : Env \to \mathrm{seq}\, NAME \to Env \\
\hline
\forall\, Env;\ \mathcal{E} : Env;\ new\_params : \mathrm{seq}\, NAME \bullet \\
\quad addParams(\theta Env)(new\_params) = \mathcal{E} \Leftrightarrow \\
\quad (\exists\, params == new\_params \bullet \theta Env = \mathcal{E})
\end{array}
$$

Add given and free types.

$$
\begin{array}{l}
addGiven : Env \to NAME \to Env \\
addFree : Env \to NAME \to Env \\
\hline
\forall\, Env;\ \mathcal{E} : Env;\ n : NAME \bullet \\
\quad (addGiven(\theta\, Env)(n) = \mathcal{E} \Leftrightarrow (\exists\, given == given \cup \{\, n \,\} \bullet \theta\, Env = \mathcal{E})) \land \\
\quad (addFree(\theta\, Env)(n) = \mathcal{E} \Leftrightarrow (\exists\, free == free \cup \{\, n \,\} \bullet \theta\, Env = \mathcal{E}))
\end{array}
$$

Add declarations of an environment as data-, port-, and constant variable declarations.

$$
\begin{array}{l}
addData, addPort, addConst : Env \to Env \to Env \\
\hline
\forall\, Env;\ \mathcal{E}, \mathcal{E}' : Env \bullet \\
\quad (addData(\theta\, Env)(\mathcal{E}') = \mathcal{E} \Leftrightarrow \\
\qquad (\exists\, type == type \oplus \mathcal{E}'.type; \\
\qquad\quad data == data \cup (\mathsf{dom}\, \mathcal{E}'.type \setminus (\mathcal{E}'.free \cup \mathcal{E}'.given)) \bullet \theta Env = \mathcal{E})) \\
\quad \land \\
\quad (addPort(\theta\, Env)(\mathcal{E}') = \mathcal{E} \Leftrightarrow \\
\qquad (\exists\, type == type \oplus \mathcal{E}'.type; \\
\qquad\quad port == port \cup (\mathsf{dom}\, \mathcal{E}'.type \setminus (\mathcal{E}'.free \cup \mathcal{E}'.given)) \bullet \theta Env = \mathcal{E})) \\
\quad \land \\
\quad (addConst(\theta\, Env)(\mathcal{E}') = \mathcal{E} \Leftrightarrow \\
\qquad (\exists\, type == type \oplus \mathcal{E}'.type; \\
\qquad\quad const == const \cup (\mathsf{dom}\, \mathcal{E}'.type \setminus (\mathcal{E}'.free \cup \mathcal{E}'.given)) \\
\qquad \bullet \theta Env = \mathcal{E}))
\end{array}
$$

Add a Statechart to the environment.

$$
\begin{array}{l}
addStatechart : Env \to State \to Env \\
\hline
\forall\, Env;\ \mathcal{E} : Env;\ s : State \bullet \\
\quad addStatechart(\theta Env)(s) = \mathcal{E} \Leftrightarrow (\exists\, root == s \bullet \theta Env = \mathcal{E})
\end{array}
$$

Join two environments. The two environments are supposed to be disjunct, i. e. not declaring the same symbols. If the environments are not disjunct, the second environment overrides declarations of the first one.

$$
\begin{array}{l}
joinEnv : Env \times Env \nrightarrow Env \\
\hline
\forall\, Env;\ \mathcal{E}, \mathcal{E}' : Env \bullet \\
\quad joinEnv(\mathcal{E}, \mathcal{E}') = \theta Env \Leftrightarrow \\
\qquad type = \mathcal{E}.type \oplus \mathcal{E}'.type \land \\
\qquad defs = \mathcal{E}.defs \oplus \mathcal{E}'.defs \land \\
\qquad data = \mathcal{E}.data \cup \mathcal{E}'.data \land \\
\qquad port = \mathcal{E}.port \cup \mathcal{E}'.port \land \\
\qquad given = \mathcal{E}.given \cup \mathcal{E}'.given \land \\
\qquad free = \mathcal{E}.free \cup \mathcal{E}'.free \land \\
\qquad builtins = \mathcal{E}.builtins \cup \mathcal{E}'.builtins \land \\
\qquad const = \mathcal{E}.const \cup \mathcal{E}'.const \land \\
\qquad root = \textbf{if}\ \mathcal{E}'.root = defaultState\ \textbf{then}\ \mathcal{E}.root\ \textbf{else}\ \mathcal{E}'.root \land \\
\qquad params = \mathcal{E}.params \frown \mathcal{E}'.params
\end{array}
$$

Get the data schema of the current environment.

$$
\begin{array}{l}
\Delta\mathrm{DATA}, \mathrm{DATA} : Env \to Decl \\
\hline
\forall\, Env \bullet \mathrm{DATA}(\theta\, Env) = \\
\quad \mathsf{setreduce}\ (\_;\ \_)((\_ : \_)(\!|(data \cup port) \lhd (\mathsf{typeToExpr} \circ second \circ type)|\!)) \\
\forall\, Env \bullet \Delta\mathrm{DATA}(\theta\, Env) = \\
\quad \mathsf{setreduce}\ (\_;\ \_)((\_ : \_)(\!|(data \cup port) \lhd (\mathsf{typeToExpr} \circ second \circ type)|\!) \\
\qquad\qquad \cup \\
\qquad\qquad (\_ : \_)(\!|((data \cup port) \lhd (\mathsf{typeToExpr} \circ second \circ type)) \circ prime^{\sim}|\!))
\end{array}
$$

Add the variables declared in the given declaration to the environment. The variables are added with the given types, rather than the types of the declaration. The assignments of variables to the types is done with respect to the declaration's characteristic tuple. The i-th variable in the characteristic tuple is assigned the i-th type in the given type sequence.

$$addVarsWithTypes : Env \rightarrow Decl \times \mathrm{seq}_1\ \mathsf{Type} \nrightarrow Env$$

$$\begin{aligned}
&\forall \mathcal{E} : Env;\ d : Decl;\ \vec{\tau} : \mathrm{seq}_1\ \mathsf{Type} \bullet \\
&\quad ((d, \vec{\tau}) \in \mathsf{dom}(addVarsWithTypes\ \mathcal{E}) \Leftrightarrow \#(ct\ d) = \#\vec{\tau}) \land \\
&\quad ((d, \vec{\tau}) \in \mathsf{dom}(addVarsWithTypes\ \mathcal{E}) \Rightarrow \\
&\qquad addVarsWithTypes(\mathcal{E})(d, \vec{\tau}) = \\
&\qquad\quad addDecls(\mathcal{E})(\varnothing, \mathsf{ran}(\mathsf{zip}(first \circ (ct\ d), \vec{\tau}))))
\end{aligned}$$

$$addVarsWithDefs : Env \rightarrow Decl \times \mathrm{seq}_1\ \mathsf{Type} \times \mathrm{seq}_1\ Expr \nrightarrow Env$$

$$\begin{aligned}
&\forall \mathcal{E} : Env;\ d : Decl;\ \vec{\tau} : \mathrm{seq}_1\ \mathsf{Type};\ \vec{e} : \mathrm{seq}_1\ Expr \bullet \\
&\quad ((d, \vec{\tau}, \vec{e}) \in \mathsf{dom}(addVarsWithDefs\ \mathcal{E}) \Leftrightarrow \#(ct\ d) = \#\vec{\tau} = \#\vec{e}) \land \\
&\quad ((d, \vec{\tau}, \vec{e}) \in \mathsf{dom}(addVarsWithDefs\ \mathcal{E}) \Rightarrow \\
&\qquad addVarsWithDefs(\mathcal{E})(d, \vec{\tau}, \vec{e}) = \\
&\qquad\quad addDefs(\mathcal{E})(\varnothing, \mathsf{ran}(\mathsf{zip}(first \circ ct\ d, \vec{e}))))
\end{aligned}$$

Chapter 6

Statecharts

This Chapter describes the necessary translation for model checking Statecharts. It describes the semantic-preserving translation of Statecharts into a Z state transition relation. The state transition relation will later be translated into the input language of the model checker.

First, a short introduction to Statecharts semantics is given. In particular, the problems that are raised by its semantics are discussed. Second, the basic principles of the translation are described. These are how Statechart steps are mapped to model checker steps (the *one-to-one step mapping*), how time and timeouts are handled, how the Statemate configuration (the set of active states) is represented, and how the problem of *writing* variables and racing is handled (section 6.2 on page 78). For the latter, the notion of *locks* and *places* is introduced. With the locks and places, the Statechart translation can be done in a straightforward fashion. This is described in section 6.3 on page 86. Concluding, an example translation is shown (section 6.4 on page 92).

6.1 Statecharts and their Semantics

6.1.1 Overview of Different Statechart Semantics

The Statechart formalism belongs to the family of *synchronous* languages. All synchronous languages have in common that parallel sub-systems perform steps of concurrent subsystems or processes simultaneously. Communication takes place via events or global variables. Events can be raised by any process; every other processes may react to them (broadcasting). Events disappear after one step and are not queued or stored in any other way. Therefore, if an event is not expected, it disappears. Because of this behavior, events in synchronous languages are often called *volatile variables*. Other representatives of synchronous languages are Esterel, presented by Berry et. al. [3], Lustre, presented by Caspi et. al. [15], and VHDL.

Synchronous languages feature major advantages: They do not introduce nondeterminism through interleaving and they have a rather simple communication mechanism (no event queues are needed). They quite naturally model logical circuits, since

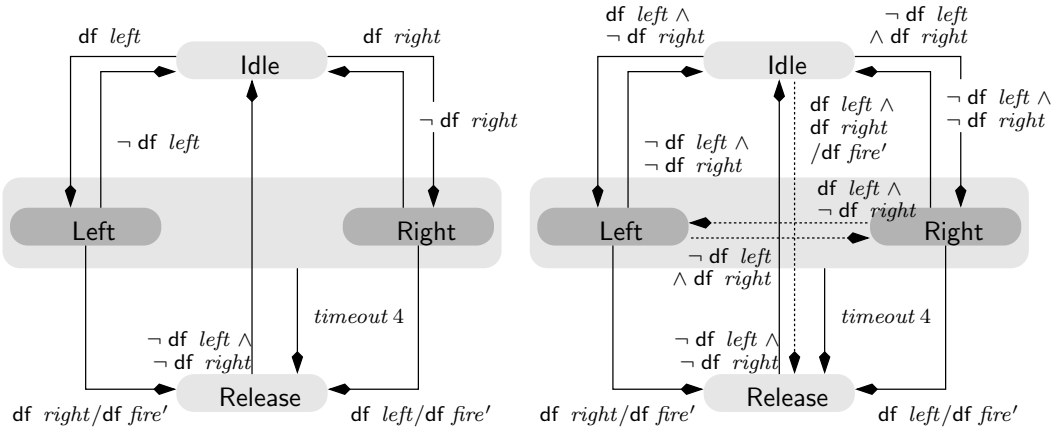


Figure 6.1: Statechart Specification of a Two Hand Press. *If the left button is pressed, the system changes to state Left and waits for the right button to be pressed. If this does not happen in a certain interval, it waits until both buttons are released and returns to the idle state. Otherwise, it starts the press and waits until both buttons are released. The specification (a) is quite clear, but neglects simultaneous pressing of both buttons. This situation is handled correctly in (b).*

---

these are also synchronous (synchronized by a global clock) and use similar means of communication.

Synchronous languages are not well suited to represent problems that appear in distributed systems or concurrent software systems, as no global synchronization takes place in these systems. Modeling languages with some kind of interleaving semantics are much more adequate for such systems.

Even if synchrony can be assumed, the unreliable means of communication, introduced by Statecharts, are disadvantageous. When specifying with Statecharts, one has to be careful not to miss any events. Consider for example the Statechart specifications of the control of a two hand press in Figure 6.1. Statechart (a) ignores simultaneous pressing of both buttons. Thus, if both buttons are pressed simultaneously, one will be ignored. This kind of error is quite likely to happen when specifying with Statecharts. Statechart (b) avoids the problem. However, this Statechart is considerably more complex and incomprehensible, because it considers all special cases.

Errors of this type are quite hard to reveal and to reproduce by ordinary testing, since it is often improbable that two events occur simultaneously. However, such errors are easily detected by verification techniques such as model checking.

These problems are better handled by asynchronous languages such as process algebras (Milner's CCS [43], Hoare's CSP [29], etc.) that queue incoming events or block the sender until the receiver is ready.

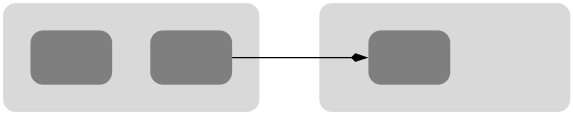


Figure 6.2: Interlevel Transition

6.1.2 Statemate Semantics

Since the first presentation of Statecharts by Harel [26], much has been achieved in defining the semantics of Statecharts, leading to a number of different Statecharts variants. Moreover, the formalism is used as a visualization frontend for many reactive specification languages, adopting the language specific semantics. An overview of these formalisms has been presented by von der Beeck [50].

The basis of Statecharts are finite state machines that have well defined and accepted semantics. Statecharts add two concepts: hierarchy and parallelism, both raising fundamental semantical questions.

- *Hierarchy:* In fact, hierarchy is no problem of itself, but in relation to interlevel transitions. Interlevel transitions cross the state hierarchy. For example the transition depicted in Figure 6.2 is an interlevel transition. Because of the interlevel transitions, it is hard to find compositional semantics. For that reason, many Statechart dialects disallow interlevel transitions. It is still possible to define a compositional Statechart semantics as shown by Huizing and de Roever [32].
- *Parallelism:* When working with finite state machines, the number of states to be drawn is likely to grow exponentially with the number of features added to the specification. As Harel [26] points out, this is the main reason for introducing parallel states (or *orthogonal* states, as he calls them). However, the introduction of parallelism raises the known problem of concurrency—which finite state machines avoid. Therefore, a model for handling concurrency and synchronization has to be chosen for Statecharts. Besides avoiding state explosion, support of concurrency additionally opens the usage of Statecharts for application domains like reactive systems.

Harel has proposed synchronous semantics for Statecharts, so that all parallel states perform a step synchronously. Thereby, a very simple semantics for *events* can be applied. Events are emitted and exist exactly in one state. During this state, all receivers can receive the event. No more synchronization, no blocking, and no event queues are needed for communication. By contrast, in process algebras, concurrent processes are synchronized to exchange events. This is not necessary in a synchronous language, since all processes are synchronized within each step anyway.

A problem of synchronous semantics is to find reasonable means of computational abstraction. In a common programming language, for example, one would expect that the statement: `x := 3; x := x+1` is equivalent to `x := 4`. In Statecharts, however, this is not true, since parallel processes synchronize on each step of the computation. In the

first case, a parallel process can observe the step where `x = 3`, whereas in the second case, this step does not exist. Thus, the environment can and must observe how many steps a computation takes.

Different solutions have been proposed to tackle this problem for synchronous languages, some of them were also applied to Statecharts.

- One of the first Statechart semantics was presented by Pnueli and Shalev [46]. Here a transition that emits an event can trigger further transitions within the same step. Transitions are executed until no more events are added. This approach abstracts internal communication. Other synchronous languages such as Esterel adopt this semantics as well. The means of abstraction apply only for events. A statement such as [*emit E*; *when E emit F*] is equivalent to [*emit E*; *emit F*], or if E is not visible to the environment [*emit F*]. However, these means of abstraction apply only to events. Statements such as `x := 3; x:= x+1` are not supported, because one variable can have only one value during a step.

  This also leads to the *global consistency problem*, which is raised, for instance, if a specification contains the statements [*when E emit F*] and [*when* $\neg$ *F emit E*]. The global consistency problem is the major drawback of this approach. Arbitrarily large chains of events can cause inconsistencies and the larger a specification gets, the more likely it is for such a problem to occur. It can happen especially if different modules are put together. Therefore it is called *global* consistency problem. It makes it almost impossible to handle large specifications. The global consistency problem is also the main reason why Harel and i-Logix did not chose this semantics for Statemate.
- Multi clocked semantics can assign each variable its own clock with its own clock rate. With this, it is possible to introduce local variables with faster clock rates that can be used for internal calculations. Lustre has a multi clocked semantics.
- In Statemate, the semicolon operator in the action language is given a special meaning. An action `x := 3; x:= x+1` causes raising on `x`. For loops, Statement introduces *immediate variables*, prefixed by a `$`. These variables can have different values during on step. However, they cannot be used for internal communication and are thus not suitable for abstraction.

6.1.3 Principles of the Translation

For symbolic model checking, a system is represented by a Kripke structure and a set of initial states, as presented in section 4.1 on page 35. Here, the state space is modeled via a set of variables with finite value domains. These variables can easily be transformed to a set of Boolean variables.

In order to model check Statecharts or $\mu\mathcal{SZ}$ classes, the state space is built up by the data variables plus some variables representing the set of active Statechart states, i. e. the configuration (see section 6.1.5 on page 76). The state transition relation is then deduced from the behavioral semantics of the Statecharts. This implies that one step of

the original Statechart is represented by exactly one step of the model checking algorithm.[1] The *one-to-one step mapping* has several advantages:

- No extra steps are needed to compute reachable states.
- No extra states have to be added to the state space that would be needed to track the intermediate steps.
- No intermediate states appear in the set of reachable states, so that the CTL formula next operators ($\mathcal{AX}, \mathcal{EX}$) always represent the next Statechart state and not some intermediate state.

This approach, however, suffers from one disadvantage: The state transition relation may become quite complex, leading to a large BDD representation. Adding a fixed number of intermediate steps causes linear growth of the number of steps needed for the computation of the reachable states, whereas the complexity of the BDD algorithms is quadratic with respect to the size of the BDDs. Therefore, computing some extra steps may be cheaper than a large transition relation.

This becomes true for complex computations in the transitions' actions. Statemate supports, for example, loops in the transitions' actions. Computing these loops (if possible) in a single step causes a quite complex transition relation, since the loop has to be completely unrolled. Moreover, "while-loops" cannot be supported at all. The same applies for Z functions. As described in section 7.4 on page 103, functions are only supported if they can be expanded to flat expressions. This also leads to a large transition relation. If a Z function was to be computed in several steps, however, the one-to-one mapping would not be possible. While giving up the one-to-one mapping might make sense for complex computations, it does not make sense for Statecharts. Computing the Statecharts transition relation in several steps would make the relation only more complex. Therefore, loops etc. are not supported.

6.1.4 Statecharts and Time

Statecharts adopt the *synchrony hypothesis* formulated by Berry and Gonthier [4]. The synchrony hypothesis assumes for the system in question that it reacts so fast that it is impossible for the environment to observe the computation time. That makes it is possible to abstract from the actual computation time and to claim that the computation takes place in *zero time*. The synchrony hypothesis offers a feasible abstraction from the details of the system's implementation. A consequence is, however, that it is impossible to specify real-time constraints, i. e. that some computation has only limited time to conclude. The only real-time support offered by Statecharts are therefore timeouts.

Statemate offers two time models, the *synchronous* and the *asynchronous time model*. In the synchronous time model, time elapses after each step, whereas in the asynchronous

[1] The *basic step algorithm* presented by Harel and Naamad [27], for example, computes a Statechart step with seven intermediate steps (the objective of their Statecharts semantics is definitely not model checking).

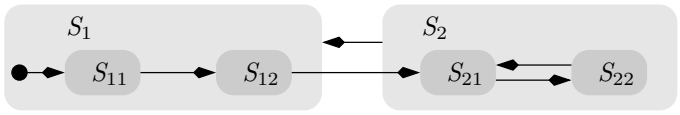


Figure 6.3: Example Statechart

time model, only elapses if the Statechart cannot perform further steps. However, even though no time elapses in the Statemate model, Statemate still allows an external observer to interact with the system. This reduces the idea of *zero time* to absurdity, since zero time only applies to some virtual system time and not to the real-time of the environment. Moreover, as pointed out by Harel and Naamad [27], the concrete semantics, implemented in Statemate, *differs slightly* in the different Statemate tools. For these reasons, the asynchronous time model is not subject of further consideration here.

Statecharts support time via timeouts. Timeouts are special events that are raised some time after an arbitrary condition (*activation condition*) becomes true (e. g. five time steps after Statechart state S was entered). These timeouts need special treatment for the Statechart representation by a transition relation. Since the state space has to be finite, it is not possible to introduce a clock variable that is incremented for each step. A clock variable with a finite domain would limit the possible traces to traces with a fixed length. Timeouts are realized via *timer variables* that are initialized when the activation condition is true and are decremented thereafter.

Therefore, by contrast to Statemate's timeouts, an upper bound of the length of a timeout has to be known at compile time, in order to be able to declare the clock variable. Note that due to this representation, timeouts enlarge the state space according to their length.

6.1.5 Representation of the Configuration

In order to represent the Statechart semantics by a state transition relation, the configuration (the set of active states) has to be represented somehow.

There are different ways to model the configuration. The model of the Statechart configuration is crucial for model checking, since it strongly affects the size of the state space and the transition relation. Besides being small, the model should also preserve independence of the specification in order to keep the BDD small. If two independent parts of the specification use the same variable to represent their configuration, the BDD, representing the transition relation, grows significantly.

It is easy to compute the number of possible configurations of a Statechart over the Statechart's structure. This is done by the function $\#_c$. For a basic state, the number of configurations is one, for an and-state it is the product of each of its sub-state's number of configurations, and for an xor-state it is the sum thereof.

Configuration						Size Optimal		Best		
S_1	S_{11}	S_{12}	S_2	S_{21}	S_{22}	v_1	v_2	v_1	v_2	v_3
×	×	-	-	-	-	1	1	1	1	-
×	-	×	-	-	-	1	0	1	0	-
-	-	-	×	×	-	0	1	0	-	1
-	-	-	×	-	×	0	0	0	-	0

Table 6.1: Configuration Model

$$
\begin{array}{|l}
\#_c : \mathit{State} \rightarrow \mathbb{N} \\
\hline
\forall\, n : \mathit{NAME} \bullet \#_c(\mathit{Basic}\ n) = 1 \\
\forall\, n : \mathit{NAME};\ \vec{S} : \mathrm{seq}_1\ \mathit{State} \bullet \#_c(\mathit{And}(n, \mathsf{ran}\ \vec{S})) = \prod_{i:\mathsf{dom}\ \vec{S}} \#_c(\vec{S}\ i) \\
\forall\, n : \mathit{NAME};\ T : \mathbb{F}\ \mathit{Trans};\ \vec{S} : \mathrm{seq}_1\ \mathit{State} \bullet \#_c(\mathit{Xor}(n, T, \mathsf{ran}\ \vec{S})) = \sum_{i:\mathsf{dom}\ \vec{S}} \#_c(\vec{S}\ i)
\end{array}
$$

A configuration model is optimal if for some Statechart with root state s, the number of states needed to represent it equals $\#_c\, s$. Note that this reflects only the Statechart's structure and ignores its behavioral characteristics. The fact, that some combinations of active states might be unreachable, is not taken into consideration. Such an analysis is almost as complicated as proving temporal logic formulae.

Choosing an optimal configuration model with respect to its size, does not necessarily lead to the best result. As stated in section 4.6 on page 44, BDD-based model checking depends only indirectly on the size of the state space, but directly on the size of the BDD representing the transition relation. This size is minimized if variables do not depend too much on each other or do not appear in different sub-terms of the formula. Therefore, it can be advantageous to introduce extra variables in order to maintain independence.

The minimal configuration model for the example Statechart depicted in Figure 6.3 on the preceding page had four states. This can be represented with two Boolean variables: one to represent which super-state is active (S_1 or S_2) and one to represent which sub-state is active (S_{11} or S_{12} if S_1 is active and S_{21} or S_{22} if S_2 is active). The second variable is reused for both super-states.

In order to preserve independence, it is better to introduce a third variable to represent the sub-states of super-state S_2. This is the approach that is used in this work.

The configuration is modeled in storing the active sub-state for each xor-state. In this model, a state is active if and only if it is the root state, or its parent state is an active and-state, or its parents state is an active xor-state and it is the active sub-state of its parent. The advantages of this approach are:

- It is quite moderate in size, yet not optimal.
- It preserves independence.
- History can be modeled easily.
- In the transition relation, configuration variables of inactive states do not have to be considered. This reduces the size of the BDDs significantly.

6.2 Resolving Racing in Statecharts

6.2.1 Racing and Persistency in Statecharts

section *Racing* **parents** *Syntax,Environment,Aux*

Variables as used in $\mu\mathcal{SZ}$ or Statecharts need a special treatment when being translated to Z. The problems to be addressed are *racing* and *persistency of non-written variables*.

The situation that two or more parallel charts of a Statechart write the same variable during one single step is called *racing*. To resolve this situation, only one of the charts is allowed to write, whereas the others fail. In other words, one write access is executed and the others are ignored. Which write access succeeds, is chosen non-deterministically.

To model this behavior, the notion of *writing a variable* has to be determined first. In a common imperative programming language, such as the Statemate's action language, write access to a variable is easy to determine, since the variable appears on the left side of an assignment statement, for example `V:=4`. In Z, this is not so easy, since values of variables are changed by predicates over a variable's primed counterpart (for example $4 \geq x' \geq x \land x' \in S$). $\mu\mathcal{SZ}$ defines a variable to be written by an operation if its primed counterpart is used in the respective predicate.

Even without parallel charts, some special treatment of variables is needed in order to establish assignment semantics for variables. In Statecharts, a variable that is not assigned any value during one step, keeps its value. In a Z transition relation, this has to be stated explicitly.

Büssow and Grieskamp [13] proposed the concept of *locks* and *places* to handle the problem of persistency and racing.

6.2.2 Derived Variables

In Z, it is a common practice to define the value of state variables through an invariant rather than setting its value explicitly in an operation. This is done for the variable y in data schema D. These variables change their value in order to obey to the property without being explicitly written. This conflicts with $\mu\mathcal{SZ}$' notion of variable persistency, were variables that are not written in an operation, keep their values.

DATA D

$$x, y, z : 0 \,..\, 255$$

$$y = 2 * x \land z > x$$

For this, $\mu\mathcal{SZ}$ introduces the notion of *derived variables*. By contrast to normal data variables, there is no racing on derived variables and they are not kept persistent automatically. Derived variables may not be used in their primed form, which means that they may not be written. Their value is determined by the invariant of the data space.

In schema D, y, z can be declared as derived. Their values are *derived* from the value of x. Thus, if the value of x is changed, the values of y and z are adapted automatically.

The behavior of derived variables corresponds with the behavior of normal Z variables. There is no persistency nor racing. Therefore, no extra effort is needed to support them. The measure presented in the following to support data variables, must not be applied to derived variables. The translation takes this into account implicitly: Support of data variables relies on the occurrence in primed form (*writing occurrence*). Derived variables must not occur in primed form. Therefore, they are treated as normal Z variables

6.2.3 Locks and Places

Racing and persistency are handled by *variable locks* and *places*. Places give the concurrent activities identities. As presented in [14], each activity (i. e. a Statechart transition) is assigned a unique place, identifying the activity. Each variable has a lock variable defining the place that currently has writing access. There is a special place *none* that is chosen if no activity is writing the variable and the variable should keep its value. If two places write the same variable, they are said to be in *conflict* for this variable.

For a variable, the following situations may appear:

- No action is writing the variable during a step; the variable should keep its value (default action). This happens if no action that writes the variable is executed during the step. Volatile variables (events) are reset in this case. Nevertheless, these variables are not subject of further consideration here.
- A single action is writing the variable; the variable gets the value assigned by this very action. It is important that the choice is selected from the writing action. Non-writing actions or the default action may not be executed, i. e. they must not hold the lock for the variable. Note that the assignment may be non-deterministic, in this case one admissible value is chosen non-deterministically and assigned to the variable.
- Several actions are writing the variable; one action is selected non-deterministically and a value is assigned as in the previous case.

The function $used$ computes the free variables of a predicate (definition omitted). $places$ represents the set of names that are used for places. $lock$ computes the lock variable for a variable name. $none$ represents the *none* place, that is the place that has a variable's lock if no other place writes the variable.

$$
\begin{array}{l}
used : Env \times Pred \to \mathbb{P}\, NAME \\
places : \mathbb{P}\, special \\
lock : NAME \to special \\
none : special
\end{array}
$$

Example: Consider a place plc and a variable v with the lock v_{lock}. The variable v is written at the place by some action, for example $v' = 4$. This action is transformed to $v_{lock} \neq none \land (v_{lock} = plc \Rightarrow v' = 4)$. This ensures that the variable is written by some place ($v_{lock} \neq none$), plc or another, but not $none$. Furthermore, it writes the variable, if plc has the lock. If the action is not executed, because, in case of a Statechart transition,

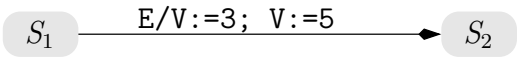


Figure 6.4: Racing Caused by Transition Segments (Statemate Syntax)

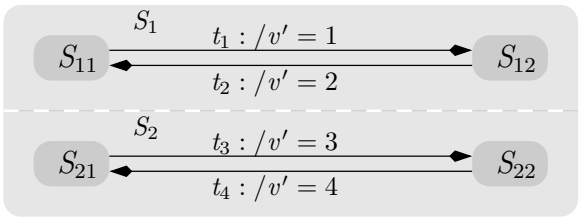


Figure 6.5: Conflicting and None-Conflicting Transitions. *Transitions t_1 and t_2 are never executed in the same step and can thus share a place. The same holds for t_3 and t_4. The two pairs, however, need different places since they are executed simultaneously and cause racing on v.*

the transition is not executed, $v_{lock} \neq plc$ has to be ensured. The same holds for places that never write the variable. Thus, if no place writes the variable, v_{lock} equals *none*. To establish persistency, $v_{lock} = none \Rightarrow v' = v$ is added as invariant for v_{lock}. Note that the places and locks are introduced to handle both racing and persistency.

Places are modeled as special identifiers:

$$| \; Place : \mathbb{P}\, special$$

6.2.4 Sharing Places

Lock variables range over the possible places and are added to the data space. For model checking, this approach causes some overhead to the size of the data space, because the number of system states is increased. It is important to keep this overhead as small as possible. In order to warrant persistency, each variable that is written by at least one place needs a lock variable with at least two values, i. e. *none* and the places of the writing actions. The actual growth of the state space depends on how many values the lock variable can take, i. e. the number of places that can write the variable. This number should thus be kept as small as possible. Therefore, places have to be shared among actions that are not in conflict. In order to compute the necessary places and the possible sharing, analysis of the Statechart is needed. This will be described in the following.

For Statecharts, each transition segment is a place, since racing already occurs, if different transition segments of a transition write the same variable. In Figure 6.4, the transitions cause racing on v. Assigning each transition segment a unique place leads to a relatively high number of places needed for a Statechart. Most transitions are not in conflict, however, since they cannot fire simultaneously or do not write the same variable. Moreover, users of Statecharts usually avoid racing so that it appears quite rarely. Having

model checking in mind, a more sophisticated solution is needed in order to reduce the number of places. The problem is that in the above presented approach, the domain of each variable lock is the set of all places or at least the places that write the variable.

In order to minimize the needed places, first, places are assigned per variable. That means an activity may get a different place for each variable it writes. Secondly, activities that cannot be executed simultaneously and thus do not compete for writing, share places. For this, the Statechart is analyzed to search for potential racing occurrences. This is done only on the basis of the Statechart's structure and not by considering the semantics of the guards or the like. In Figure 6.5 on the preceding page, the two pairs of transitions t_1, t_2 and t_3, t_4 can each share a place.

6.2.4.1 Writing Occurrences

Racing can already occur in a single transition, if several transition segments write the same variable as shown in Figure 6.4 on the facing page. Therefore, places have to be assigned to transition segments. In the abstract syntax used here (see section 5.1 on page 55), such a transition is defined by a transition $t \in \mathit{Trans}$ and the specific action predicate $p \in \mathit{actions}(\mathit{label}\ t)$. This pair is called a *writing occurrence*.

$$\mathit{wrtOcc} : \mathbb{P}(\mathit{Trans} \times \mathit{Pred})$$
$$\forall\, t : \mathit{Trans};\ p : \mathit{Pred} \bullet (t, p) \in \mathit{wrtOcc} \Leftrightarrow p \in \mathit{actions}(\mathit{label}\ t)$$

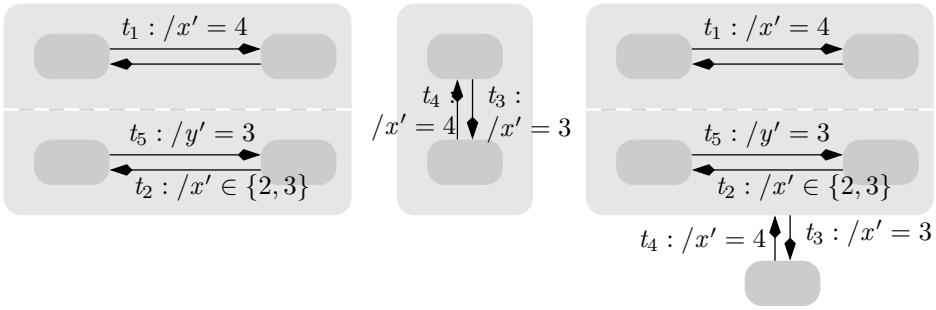
6.2.4.2 Non-Conflicting Writing Occurrences

For a given Statechart, each data-variable is written by a set of writing occurrences. A subset of this set is called a *set of non-conflicting writing occurrences*, if all pairs of its members cannot write the variable simultaneously. Thus all members of a set of non-conflicting writing occurrences can share a place. In order to share places, these sets of non-conflicting writing occurrences have to be computed.

To minimize the number of places needed for a variable, a "minimal" set of sets of writing occurrences has to be computed. This set has to fulfill the following requirements:

- The writing occurrences of a non-conflicting set for the variable have to be pairwise non-conflicting.
- Each transition segment (writing occurrence) that writes the variable has to be in at least one set.
- The number of sets has to be as small as possible, thus the union of two sets of non-conflicting writing occurrences must not be a set of non-conflicting writing occurrences. Note, however, that a single writing occurrence may appear in several sets.

As an example consider Figure 6.6 on the following page.



(a) (b) (c)

Figure 6.6: Sets of Non-Conflicting Writing Occurrences. *In (a), the two transitions t_1 and t_2 concurrently write x. Thus, for x the set of non-conflicting writing occurrences is $\{\{ t_1 \}, \{ t_2 \}\}$. In (b), there is no conflict between the transitions t_3 and t_4, thus t_3 and t_4 form a set of non-conflicting writing occurrences: $\{\{ t_3, t_4 \}\}$. When the two charts are combined (c), a set of non-conflicting writing occurrences is, e. g. $\{\{ t_1, t_3, t_4 \}, \{ t_2, t_3, t_4 \}\}$. Since t_1, t_3, and t_4 are all non-conflicting, they can be put into one set. By doing this, the number of places is kept small and so does the domain of the lock variables. Note that t_3 and t_4 are assigned several places. This does not cause a problem, however, the set $\{\{ t_1 \}, \{ t_2, t_3, t_4 \}\}$ can be chosen as well; this is done in the solution presented here.*

6.2.4.3 Computing Writing Occurrences for Statecharts

The sets of non-conflicting writing occurrences are firstly computed for a single transition. As mentioned above, several actions in one transition (resulting from several transition segments), cause racing. Thus, for some variable v, each action p that writes this variable, forms a writing occurrences set with only one member.

Racing can only occur for port- and data-variables, thus only these are considered in the definition of the writing occurrences.

The function *used* computes the free variables of a predicate under a given environment. The environment is needed to resolve abbreviations. *prime* computes the primed counterpart of a variable: $prime\ v = v'$. Thus, $prime^{\sim}(\!| used(\mathcal{E}, p) |\!)$ computes the unprimed versions of all primed free variables of the predicate p. With this, for some transition t, *raceTrans* assigns each variable those writing occurrences of t that write the variable. Variables that are not written by any action of t are assigned the empty set.

$$
\begin{array}{|l}
raceTrans : Env \times Trans \to NAME \to \mathbb{F}(\mathbb{F}\ wrtOcc) \\
\hline
\forall\, \mathcal{E} : Env;\ t : Trans \bullet \\
\quad raceTrans(\mathcal{E}, t) = (NAME \times \{ \varnothing \}) \oplus \\
\quad\quad (\lambda\, v : getData\ \mathcal{E} \cup getPort\ \mathcal{E} \bullet \\
\quad\quad\quad \{ p : actions(label\ t) \mid v \in prime^{\sim}(\!| used(\mathcal{E}, p) |\!) \bullet \{ t \mapsto p \} \})
\end{array}
$$

Note that no writing occurrences for derived variables are found, since these variables must not occur in primed form.

The sets of writing occurrences are now computed for each state. Basic-states do not cause any write access to variables, therefore their writing occurrences sets are empty

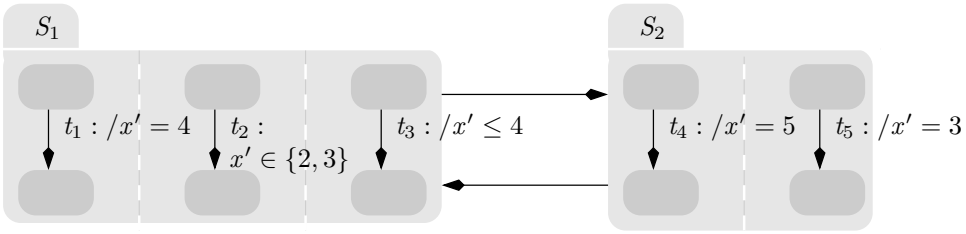


Figure 6.7: Combining Writing Occurrences. *For variable x, the Statechart state S_1 has three sets of writing occurrences ($\{\{t_1\}, \{t_2\}, \{t_3\}$) and thus needs at least three places, whereas S_2 has only two sets ($\{\{t_4\}, \{t_5\}$). The resulting set of non-conflicting writing occurrences is, $\{\{t_1, t_4\}, \{t_2, t_5\}, \{t_3\}$. Each of the original writing occurrences sets is subset of exactly one resulting set.*

$$combine\Big\{\overbrace{\{\{t_3\}, \{t_2\}\}}^{conflicting}, \underbrace{\{t_1\}\}, \{\{t_4\}}_{non\text{-}conflicting}, \{t_5\}\}\Big\} = \{\underbrace{\{t_1, t_4\}}_{non\text{-}conflicting}, \{t_2, t_5\}, \{t_3\}\}$$

for every variable. And-states introduce racing, and their racing occurrences sets are therefore combined. So if two sub-charts of an and-state write the same variable, they are conflicting. Xor-states do not introduce racing. The writing occurrences sets are combined in a minimal way. This is done by *combine*. See Figure 6.7 for an example.

$$\begin{array}{|l}
raceState : Env \times State \to NAME \to \mathbb{F}(\mathbb{F}\, wrtOcc) \\
\hline
\forall\, \mathcal{E} : Env;\ n : NAME \bullet \\
\quad raceState(\mathcal{E}, Basic\, n) = NAME \times \{\varnothing\} \\
\forall\, \mathcal{E} : Env;\ n, v : NAME;\ S : \mathbb{F}\, State \bullet \\
\quad raceState(\mathcal{E}, And(n, S))(v) = \bigcup\{s : S \bullet raceState(\mathcal{E}, s)(v)\} \\
\forall\, \mathcal{E} : Env;\ n, v : NAME;\ T : \mathbb{F}\, Trans;\ S : \mathbb{F}\, State \bullet \\
\quad raceState(\mathcal{E}, Xor(n, T, S))(v) = \\
\qquad combine(\{s : S \bullet raceState(\mathcal{E}, s)(v)\} \cup \\
\qquad\quad \{t : T \bullet raceTrans(\mathcal{E}, t)(v)\})
\end{array}$$

$$\begin{array}{|l}
combine : \mathbb{F}(\mathbb{F}(\mathbb{F}\, wrtOcc)) \to \mathbb{F}(\mathbb{F}\, wrtOcc) \\
\hline
\forall\, TPP : \mathbb{F}(\mathbb{F}(\mathbb{F}\, wrtOcc)) \bullet \\
\quad \bigcup(combine\ TPP) = \bigcup(\bigcup TPP) \wedge \\
\quad (\forall\, TP : TPP \bullet \forall\, tp : TP \bullet \\
\qquad \exists_1\, tp' : combine\ TPP \bullet \\
\qquad\quad tp \subseteq tp' \wedge \\
\qquad\quad (\forall\, tp'' : combine\ TPP \setminus \{tp'\} \bullet tp \cap tp'' = \varnothing)) \wedge \\
\quad \#(combine\ TPP) = max(\#(\!|\, TPP \,|\!))
\end{array}$$

6.2.4.4 Assigning Places to Sets of Writing Occurrences

A general function is defined which assigns places to sets of writing occurrences. The individual writing occurrences in a set of writing occurrences *tp* share the place *placeAssigner tp*. This assignment is carried out independently of the actual non-

conflicting sets of a concrete Statechart or $\mu\mathcal{SZ}$ class.

$$placeAssigner : \mathbb{F}\, wrtOcc \rightarrowtail Place$$

Some auxiliary functions are defined that compute writing occurrences of a given specification. The following information is computed from a specification defined by an environment $\mathcal{E}$ (see sectionsec:env). The functions compute:

- which places can write a variable ($varPlaces$),
- which places can be used by a writing occurrence ($occPlaces$),
- which places are used by a transition ($transPlaces$), and
- which places are used by a state ($statePlaces$).

These definitions are needed to define the Statechart's state transition in the following sections.

Each variable is assigned the set of places that may write this variable. This is the domain of its lock variable.

$$varPlaces : Env \rightarrow NAME \rightarrow \mathbb{F}\, Place$$

$$\forall \mathcal{E} : Env;\ v : NAME \bullet varPlaces(\mathcal{E})(v) = placeAssigner(\!| raceState(\mathcal{E}, getRoot\, \mathcal{E})(v) |\!)$$

For each variable, each writing occurrence is assigned the set of places under which it may write the variable. Note that due to the definition of $combine$, these sets do not have more than one element.

$$occPlaces : Env \rightarrow wrtOcc \rightarrow NAME \rightarrow \mathbb{F}\, Place$$

$$\forall \mathcal{E} : Env;\ t : Trans;\ p : Pred;\ v : NAME \bullet occPlaces(\mathcal{E})(t, p)(v) = \{\, TP : raceState(\mathcal{E}, getRoot\, \mathcal{E})(v) \mid (t, p) \in TP \bullet placeAssigner(TP) \,\}$$

The places of a transition is computed by $transPlaces$ as the union of the places of the transition segments:

$$transPlaces : Env \rightarrow Trans \rightarrow NAME \rightarrow \mathbb{F}\, Place$$

$$\forall \mathcal{E} : Env;\ t : Trans;\ v : NAME \bullet transPlaces(\mathcal{E})(t)(v) = \{\, T : raceState(\mathcal{E}, getRoot\, \mathcal{E})(v) \mid t \in \mathsf{dom}\ T \bullet placeAssigner(T) \,\}$$

Each state is assigned the set of places of its (arena) transitions and its sub-states by the function $statePlaces$:

$$
\begin{array}{l}
statePlaces : Env \to State \nrightarrow NAME \to \mathbb{F}\, Place \\
\hline
\forall\, \mathcal{E} : Env;\ n : NAME \bullet statePlaces(\mathcal{E})(Basic(n)) = NAME \times \{ \varnothing \} \\
\forall\, \mathcal{E} : Env;\ n, v : NAME;\ S : \mathbb{F}\, State \bullet \\
\quad statePlaces(\mathcal{E})(And(n, S))(v) = \bigcup \{ s : S \bullet statePlaces(\mathcal{E})(s)(v) \} \\
\forall\, \mathcal{E} : Env;\ n, v : NAME;\ TT : \mathbb{F}\, Trans;\ S : \mathbb{F}\, State \bullet \\
\quad statePlaces(\mathcal{E})(Xor(n, TT, S))(v) = \\
\qquad \bigcup \{ s : S \bullet statePlaces(\mathcal{E})(s)(v) \} \cup \\
\qquad \bigcup \{ t : TT \bullet transPlaces(\mathcal{E})(t)(v) \}
\end{array}
$$

6.2.5 Rewriting Actions

In order to support the locks, actions have to be rewritten. Consider for example an action $x' = 4 \land y' = 3$ that writes the variables x and y with locks x_{lock} and y_{lock} respectively. Assume the action has the place plc_x to write x and plc_y to write y. The action is rewritten as follows (the formula is annotated with the auxiliary function defined in the following):

$$
\overbrace{x_{lock} \neq none \lor y_{lock} \neq none}^{someVarWritten} \quad \land
$$

$$
\left(\underbrace{(x_{lock} \in \{ plc_x \} \lor y_{lock} \in \{ plc_y \})}_{someVarLocked} \Rightarrow \overbrace{(x_{lock} = plc_x \land y_{lock} = plc_y)}^{allVarLocked} \land \underbrace{(x' = 4 \land y' = 3)}_{\text{original action}} \right)
$$

The action always writes all of the variables it intends to write, or none (*allVarLocked*). If it writes none, there has to be another action writing at least one of the variables (*someVarWritten*). If it has the lock of one variable, it must have the lock for both and will then write these variables (*someVarLocked* $\Rightarrow$ *allVarLocked*).

This rewriting is defined formally with the function *trSimpleAct*. For this, firstly the above mentioned auxiliary functions that compute needed predicates are defined in the following.

For a writing occurrence, the function *someVarWritten* computes a predicate which states that at least one of the variables, being written by the occurrence is actually written. The function *someVarLocked* states that the occurrence has at least one variable locked and *allVarLocked* that it has all variables locked.

$$
\begin{array}{l}
someVarWritten, someVarLocked, allVarLocked : Env \to wrtOcc \to Pred \\
\hline
\forall \mathcal{E} : Env;\ tp : Trans \times Pred \bullet \\
\quad someVarWritten(\mathcal{E})(tp) = \\
\qquad \bigvee \{\, v : NAME \mid \\
\qquad\quad occPlaces(\mathcal{E})(tp)(v) \neq \varnothing \bullet \\
\qquad\quad \neg\, (var_{\varnothing}(lock\ v) = var_{\varnothing}\ none) \,\} \land \\
\quad someVarLocked(\mathcal{E})(tp) = \\
\qquad \bigvee \{\, v : NAME \mid \\
\qquad\quad occPlaces(\mathcal{E})(tp)(v) \neq \varnothing \bullet \\
\qquad\quad var_{\varnothing}(lock\ v) \in \{,,\}(var_{\varnothing} \circ (order(occPlaces(\mathcal{E})(tp)(v)))) \,\} \land \\
\quad allVarLocked(\mathcal{E})(tp) = \\
\qquad \bigwedge \{\, v : NAME \mid \\
\qquad\quad occPlaces(\mathcal{E})(tp)(v) \neq \varnothing \bullet \\
\qquad\quad var_{\varnothing}(lock\ v) \in \{,,\}(var_{\varnothing} \circ order\ (occPlaces(\mathcal{E})(tp)(v))) \,\}
\end{array}
$$

trSimpleAct augments an action as discussed before. It adds the necessary behavior to obey the place/locking system.

$$
\begin{array}{l}
trSimpleAct : Env \to Trans \times Pred \to Pred \\
\hline
\forall \mathcal{E} : Env;\ t : Trans;\ p : Pred \bullet \\
\quad trSimpleAct(\mathcal{E})(t, p) = \\
\qquad someVarWritten(\mathcal{E})(t, p) \land \\
\qquad (someVarLocked(\mathcal{E})(t, p) \Rightarrow allVarLocked(\mathcal{E})(t, p)) \land p
\end{array}
$$

6.3 Translating Statecharts into a State Transition Relation

section *Statecharts* **parents** *Racing*

As discussed before, the Statechart defining the reactive behavior of an $\mu\mathcal{SZ}$ class has to be translated into a Z state transition relation. This translation defines the semantics of the Statechart embedding in $\mu\mathcal{SZ}$ and the mapping into a Kripke structure.

The state transition relation is built up hierarchical. For each Statechart state, a state transition relation is defined, describing the "sub-behavior" that is performed, if this state is active during a step (including not left and not entered).

For this, some basic facts of the Statechart semantics are used. Each Statechart transition has exactly one *arena state*: the least common ancestor of states it passes through. An arena state is always an xor-state, since a transition, whose arena state is an and-state, is a transition between parallel states, and that is inadmissible. Figure 6.8 gives an example for arena states.

In a state (or configuration), several transitions can be ready to fire (all transitions whose guards are true and whose source states are active). Some of these transitions can fire simultaneously, whereas others are in conflict. Transitions that are in parallel, i. e. whose arena state's least common ancestor state is an and-state, can fire simultaneously. Conflicting transitions, whose arena state's least common ancestor is an xor-state must not fire simultaneously. Thus, in order to perform a valid step, from each set of conflicting transitions one transition has to be selected that is executed. In Figure 6.8, the

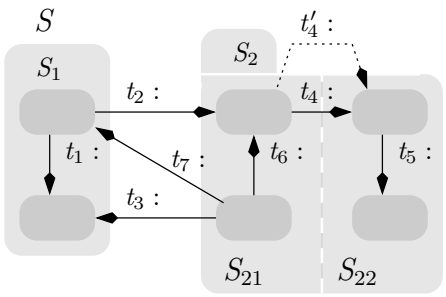


Figure 6.8: Arena States and Transitions. *The arena state of t_1 is S_1, of t_2, t_3 it is S and of t_5 it is S_{22}. The arena state of transition t_4 is the and-state S_2. The transition is therefore illegal. However, Statemate allows such a transition and interprets it as t'_4. Thus, its arena state is the xor-state S and the transition is legal.*

---

least common ancestor of the arena states of the transition t_5 and t_6 is S_2—an and-state. These transitions may thus fire simultaneously.

Here, the transitions with the greatest arena state (with respect to the state hierarchy, closest to the root state) have priority. Only if transitions have the same arena-state, a non-deterministic choice is made. In order to observe this behavior, the transitions are considered in the state transition relation of their arena state. The transitions that have a common arena state are called this state's *arena transitions*. In the abstract $\mu\mathcal{SZ}$ syntax presented in section 5.1 on page 55, an xor-state-node already holds the arena-states of this state. In Figure 6.8, the transitions t_3 and t_7 have the same arena state (S), and a non-deterministic choice between them has to be made. The arena state of the transition t_6, in contrary, is S_{21} and t_6 thus has lower priority.

Due to this prioritization, for an xor-state, all arena-transitions of its sub-states have lower priority than its own arena-transitions. Therefore, if one of its arena transitions can fire, the sub-states do not have to be regarded anymore.

The function *stateTrans* assigns each Statechart state its transition relation. A state transition relation is assigned to each Statechart states, with to the following rules:

- *Basic-State:* The transition relation does nothing, i. e. the sub-Statechart idles. Thus, for some basic-state S: $\mathit{stateTrans}\ S = \mathit{true}$.
- *And-State:* The transition relation of an and-state is defined as the conjunction of the transition relation of its sub-states. Thus, for an and-state S with sub-states $S_1, \ldots, S_n$: $\mathit{stateTrans}\ S = \mathit{stateTrans}\ S_1 \wedge \ldots \wedge \mathit{stateTrans}\ S_n$.
- *Xor-State:* The transition relation of an xor-state is defined as follows: If one of its *arena-transitions* can fire, execute this transition and otherwise execute the transition relation of the active sub-state. Thus, for an xor-state S with configuration variable conf_S:

$$stateTrans\ S = \textbf{if}\ some\_transition\_can\_fire \\ \textbf{then}\ transition\_fires \\ \textbf{else}\ conf_S = S_i \Rightarrow conf'_S = S_i \land stateTrans\ S_i$$

6.3.1 Configuration

The current state, the *configuration* of a Statechart, is modeled by the *configuration variables*. For each xor-state, a variable is introduced that holds the currently active sub-state. The function *conf* maps state names to their configuration variable names, and *st* maps state names to their occurrence names. For an xor-state S with sub-states S_1, S_2 this is $conf\ S \in \{\ st\ S_1, st\ S_2\ \}$.

The function *instatePred* computes the *instate predicate* for a state. This predicate can be used to test whether a state is active or not in the proposed configuration. *instateNext* forces the given state to be entered.

$$\begin{array}{l} conf : NAME \rightarrowtail special \\ st : NAME \rightarrowtail special \\ instatePred, instateNext : Env \rightarrow NAME \rightarrow Pred \\ \hline \forall\, \mathcal{E} : Env;\ s : NAME \bullet \\ \quad instatePred\ \mathcal{E}\ s = \\ \quad\quad var_\varnothing(conf(st(stName(getParent\ \mathcal{E}\ s)))) = var_\varnothing(st\ s) \land \\ \quad instateNext\ \mathcal{E}\ s = \\ \quad\quad var_\varnothing(prime(conf(st(stName(getParent\ \mathcal{E}\ s))))) = var_\varnothing(st\ s) \end{array}$$

6.3.2 In-State Predicates

The current configuration can be accessed from predicates with the instate relation. For some Statechart state S, instate S is true if and only if the state S is active. The instate relation has to be replaced by the respective instate predicates. The function *transInstates* translates all occurrences of instate into a predicate.

$$\begin{array}{l} Instate : NAME \\ transInstates : Env \rightarrow Pred \rightarrow Pred \\ \hline \forall\, \mathcal{E} : Env \bullet transInstates\ \mathcal{E} = \\ \quad transPred\ \{\ s : NAME \bullet (var_\varnothing\ s) \in (var_\varnothing\ Instate) \mapsto instatePred\ \mathcal{E}\ s\ \} \end{array}$$

6.3.3 Timeouts

Statecharts support timeouts. Timeouts have a trigger and a delay. The trigger is an event itself. If the time delay has elapsed since the last occurrence of the trigger event, a timeout-event is fired automatically. The most important application of timeouts is to limit the time a state is active.

$\mu\mathcal{SZ}$ has timeouts $\mu\mathcal{SZ}$ as well. The are modelled by the delay : State $\rightarrow \mathbb{N}$ function. The delay function returns the time a Statechart state has been active. A timeout can be realized by: delay $s = 10$. This predicate is true 10 steps after state s is entered.

Timeouts can be impelemented by *timeout counters*. Timeout counters are incremented until the timeout trigger becomes true. Once that happens they are reset. This is only possible if the timeouts are bounded, i. e. if a maximum value can be found for the counter. For $\mathsf{delay}\, s = 10$, a timeout counter with a maximum of 11 can be used. If 11 is reached, the counting can be stopped.

In order to determine the maximum values, timeouts are only supported if they appear as: $\mathsf{delay}\ s\ rel\ n$ (where $rel \in \{=, \neq, <, >, \leq, \geq\}$ and n is a number). The maximal value is $n+1$. Formalization of the analysis is omitted here. It is assumed that the environment contains the necessary information and that there is a function *getTimers* that returns all timers (i. e. a triple containing the timer name, the trigger predicate and the maximum value):

$$| \ getTimers : Env \to \mathbb{F}(NAME \times Pred \times Number)$$

Note that it is possible to support arbitrary trigger predicates. Therefore, a predicate is stored with the timer and not a state name. However, $\mu\mathcal{SZ}$ allows only delays for the entering of states. The trigger predicate for a state s is: $\neg\ (instatePred\ s) \land instateNext\ s)$.

The delay function is declared in the meta-language:

$$| \ Delay : Expr \to Expr$$

The function *Timer* maps a state name to the name of the respective timer.

$$\begin{array}{l} | \ Timer : NAME \to special \\ | \ transTimers : Env \to Pred \to Pred \\ \hline \forall\, \mathcal{E} : Env \bullet\ transTimers\, \mathcal{E} = \\ \quad transExpr_P\, \{\, s : NAME;\ n : Number;\ rel : BuiltinRel \bullet \\ \qquad\qquad Delay(var_{\varnothing}\ s) \mapsto var_{\varnothing}(Timer\ s)\, \} \end{array}$$

The necessary declarations for the timers are computed by *timerDecls*.

$$\begin{array}{l} | \ timerDecls : Env \to Decl \\ \hline \forall\, \mathcal{E} : Env \bullet\ timerDecls\, \mathcal{E} = \\ \quad \mathsf{setreduce}\,(\_;\ \_)\, \{\, t : getTimers\ \mathcal{E} \bullet \\ \qquad\qquad t.1 : (var_{\varnothing}(Num\, 0)) \mathinner{..} (var_{\varnothing}(t.3))\, \} \end{array}$$

timerPred computes the predicate that controls the counters. It resets the counter, if the trigger is true and increments it up to its maximum otherwise.

$$\begin{array}{l} | \ timerPred : Env \to Pred \\ \hline \forall\, \mathcal{E} : Env \bullet \\ \quad timerPred\, \mathcal{E} = \bigwedge \{\, t : getTimers\ \mathcal{E} \bullet \\ \qquad var_{\varnothing}(prime\, t.1) = (\mathbf{if}\ t.2\ \ \mathbf{then}\ var_{\varnothing}(Num\, 0) \\ \qquad\qquad\qquad\qquad \mathbf{else}\, (\mathbf{if}\, (var_{\varnothing}\ t.1) \leq var_{\varnothing}(t.3) \\ \qquad\qquad\qquad\qquad\quad \mathbf{then}\, (var_{\varnothing}\ t.1) + var_{\varnothing}(Num\, 1) \\ \qquad\qquad\qquad\qquad\quad \mathbf{else}\, (var_{\varnothing}\ t.1)))\, \} \end{array}$$

6.3.4 Guards and Actions

A transition can fire if its guard is true and all source states are active. This statement is computed by $transGuard$. Thus, if a transition's arena state is active and $transGuard(\mathcal{E})(t)$ is true, the transition may fire:

$$
\begin{array}{|l}
transGuard : Env \to Trans \to Pred \\
\hline
\forall\, \mathcal{E} : Env;\ t : Trans \bullet \\
\quad transGuard(\mathcal{E})(t) = \\
\qquad ((transInstates\ \mathcal{E}) \circ (transTimers\ \mathcal{E}))(guard(label\ t)) \land \\
\qquad \bigwedge \{\, s : sources\ t \bullet instatePred\ \mathcal{E}\ s \,\}
\end{array}
$$

If a transition fires, its (simple) action is executed and all target states are entered:

$$
\begin{array}{|l}
transAction : Env \to State \to Trans \to Pred \\
\hline
\forall\, \mathcal{E} : Env;\ s : State;\ t : Trans \bullet \\
\quad transAction(\mathcal{E})(s)(t) = \\
\qquad \bigwedge \{\, p : actions(label t) \bullet \\
\qquad\quad trSimpleAct(\mathcal{E})(t, ((transInstates\ \mathcal{E}) \circ (transTimers\ \mathcal{E}))(p)) \,\} \land \\
\qquad \bigwedge \{\, s : targets\ t \bullet instateNext\ \mathcal{E}\ s \,\}
\end{array}
$$

If a transition fires, no other transition of its arena state or one of the arena state's sub-states can fire. Thus all places of the arena state that do not belong to the firing transition must be disabled, i. e. they must not lock any variable. This is expressed by $protectOtherPlaces$:

$$
\begin{array}{|l}
protectOtherPlaces : Env \to State \to Trans \to Pred \\
\hline
\forall\, \mathcal{E} : Env;\ s : State;\ t : Trans \bullet \\
\quad protectOtherPlaces\ \mathcal{E}\ s\ t = \\
\qquad \bigwedge \{\, v : NAME \mid \\
\qquad\quad transPlaces(\mathcal{E})(t)(v) \neq statePlaces(\mathcal{E})(s)(v) \bullet \\
\qquad\quad \neg\ (var_{\varnothing}(lock\ v) \in \\
\qquad\qquad \{,,\}(var_{\varnothing} \circ order\ (statePlaces(\mathcal{E})(s)(v) \setminus \\
\qquad\qquad\qquad\qquad transPlaces(\mathcal{E})(t)(v)))) \,\}
\end{array}
$$

The complete effect of a firing transition is expressed by $trFullAct$:

$$
\begin{array}{|l}
trFullAct : Env \to State \to Trans \to Pred \\
\hline
\forall\, \mathcal{E} : Env;\ s : State;\ t : Trans \bullet \\
\quad trFullAct\ \mathcal{E}\ s\ t = transGuard\ \mathcal{E}\ t \land \\
\qquad\qquad\qquad\qquad transAction\ \mathcal{E}\ s\ t \land \\
\qquad\qquad\qquad\qquad protectOtherPlaces\ \mathcal{E}\ s\ t
\end{array}
$$

6.3.5 Statechart State Transition Relation

Now, the state-transition relation induced by the Statechart is defined recursively in assigning a state transition relation to each state. For some steps, this relation is applied, if it has been active in the pre-state and none of its ancestor states fires a transition.

$$
\begin{array}{|l}
stateTrans : Env \to State \to Pred \\
\hline
\forall \mathcal{E} : Env;\ n : NAME \bullet \\
\quad stateTrans(\mathcal{E})(Basic\ n) = true \\
\forall \mathcal{E} : Env;\ n : NAME;\ S : \mathbb{F}\ State \bullet \\
\quad stateTrans(\mathcal{E})(And(n, S)) = \bigwedge (stateTrans(\mathcal{E})(\!| S |\!)) \\
\forall \mathcal{E} : Env;\ n : NAME;\ TT : \mathbb{F}\ Trans;\ S : \mathbb{F}\ State \bullet \\
\exists\ s == Xor(n, TT, S) \bullet \\
\exists\ someTransCanFire == \bigvee (transGuard(\mathcal{E})(\!| TT |\!)); \\
\quad someTransFires == \bigvee (trFullAct(\mathcal{E})(s)(\!| TT |\!)); \\
\quad noPlacesUsed == \\
\qquad \bigwedge \{\, v : NAME \mid \\
\qquad\quad statePlaces(\mathcal{E})(s)(v) \neq \varnothing \bullet \\
\qquad\quad \neg\ (var_{\varnothing}(lock\ v) \in \{,,\}(var_{\varnothing} \circ order\ (statePlaces(\mathcal{E})(s)(v)))) \} ; \\
\quad activeStateFires == \\
\qquad \bigwedge \{\, s : S \bullet var_{\varnothing}(conf\ n) = var_{\varnothing}(st(stName\ s)) \Rightarrow stateTrans(\mathcal{E})(s) \,\} \bullet \\
\quad stateTrans(\mathcal{E})(Xor(n, TT, S)) = \\
\qquad (someTransCanFire \Rightarrow someTransFires) \land \\
\qquad \neg\ someTransCanFire \Rightarrow (var_{\varnothing}(prime(conf\ n)) = var_{\varnothing}(conf\ n)) \land \\
\qquad\qquad\qquad noPlacesUsed \land \\
\qquad\qquad\qquad activeStateFires
\end{array}
$$

6.3.6 Declarations

The Statechart translation needs additional variable declarations. The function *lockDecls* computes the declarations needed for the lock variables. The function *confDecls* computes the declarations for the configurations.

$$
\begin{array}{|l}
lockDecls : Env \to Decl \\
\hline
\forall \mathcal{E} : Env \bullet \exists\ var\_places == statePlaces(\mathcal{E})(getRoot\ \mathcal{E}) \bullet \\
\quad lockDecls\ \mathcal{E} = \\
\qquad \mathsf{setreduce}(\_;\ \_) \\
\qquad\quad \{\, v : getData\ \mathcal{E} \bullet \\
\qquad\quad\ lock\ v : \{,,\}(var_{\varnothing} \circ order\ (var\_places(v) \cup \{\, none \,\})) \}
\end{array}
$$

$$
\begin{array}{|l}
confDecls : Env \to Decl \\
\hline
\forall \mathcal{E} : Env \bullet \\
\quad confDecls\ \mathcal{E} = \mathsf{setreduce} \\
\qquad (\_;\ \_) \\
\qquad \{\, s : \mathsf{ran}(getParent\ \mathcal{E}) \cap \mathsf{ran}\ Xor \bullet \\
\qquad\ conf\ (stName\ s) : \{,,\}(var_{\varnothing} \circ st \circ order\ (stName(\!| subs(\!| \{\, s \,\} |\!)|\!))) \,\}
\end{array}
$$

All additional declarations needed for the Statechart are computed by *statechartDecls*.

$$
\begin{array}{|l}
statechartDecls : Env \to Decl \\
\hline
\forall \mathcal{E} : Env \bullet statechartDecls\ \mathcal{E} = (lockDecls\ \mathcal{E}); \\
\qquad\qquad\qquad\qquad\qquad\quad (confDecls\ \mathcal{E}); \\
\qquad\qquad\qquad\qquad\qquad\quad (timerDecls\ \mathcal{E})
\end{array}
$$

The private data variables of the class are written only by the class, whereas the port variables can also be written by the environment. Thus, only the data variables are kept persistent if they are not written by the class. It is assumed that the port-variables' locks are also exported in order to resolve racing with the environment. In each step, the value of a port-variable is defined by the class, if it holds the lock, and is defined by the environment otherwise. In the latter case, invariants of the port-variables must not be violated by the environment.

$$
\begin{array}{|l}
persistencyPred : Env \to Pred \\
\hline
\forall \mathcal{E} : Env \bullet \\
\quad persistencyPred\ \mathcal{E} = \bigwedge \{\, v : getData\ \mathcal{E} \mid \\
\qquad\qquad v \in \mathsf{dom}(statePlaces(\mathcal{E})(getRoot\ \mathcal{E})) \bullet \\
\qquad\qquad (var_{\varnothing}(lock\ v) = var_{\varnothing}\ none) \Rightarrow \\
\qquad\qquad\quad (var_{\varnothing}(prime\ v) = var_{\varnothing}\ v)\,\}
\end{array}
$$

The state transition relation for the Statechart of a class is computed by *statechartSemantics*. In addition to the actual transition relation, computed by *stateTrans*, it consists of the persistency predicate *persistencyPred* and the *timerPred*.

$$
\begin{array}{|l}
statechartSemantics : Env \to Pred \\
\hline
\forall \mathcal{E} : Env \bullet statechartSemantics\ \mathcal{E} = stateTrans(\mathcal{E})(getRoot\ \mathcal{E}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (persistencyPred\ \mathcal{E}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\quad (timerPred\ \mathcal{E})
\end{array}
$$

That's it.

6.4 Statecharts Translation by Example

section *StatechartExample* **parents** *Syntax,Statecharts*

In this section, the semantic conversion is explained by an example. The translation of the Statechart shown in Figure 6.9 on the facing page is presented.

6.4.1 Abstract Syntax of the Example Statechart

At first, the Statechart is translated into the abstract syntax, defined in Sect. 5.1 on page 55. For this, names for the variables (X and Y) and for the states S_x have to be declared. Obviously, these names are supposed to be mutually not equal.

$$
\begin{array}{|l}
X, Y, root, S_1, S_2, S_{11}, S_{12}, S_{111}, S_{112}, S_{22}, S_{21}, PLC_1, PLC_2 : NAME \\
\hline
\langle X, Y, root, S_1, S_2, S_{11}, S_{12}, S_{111}, S_{112}, S_{22}, S_{21} \rangle \in \mathbb{N} \rightarrowtail NAME
\end{array}
$$

$$
\begin{array}{ll}
x == var_{\varnothing}\ X & y == var_{\varnothing}\ Y \\
x' == var_{\varnothing}(prime\ X) & y' == var_{\varnothing}(prime\ Y)
\end{array}
$$

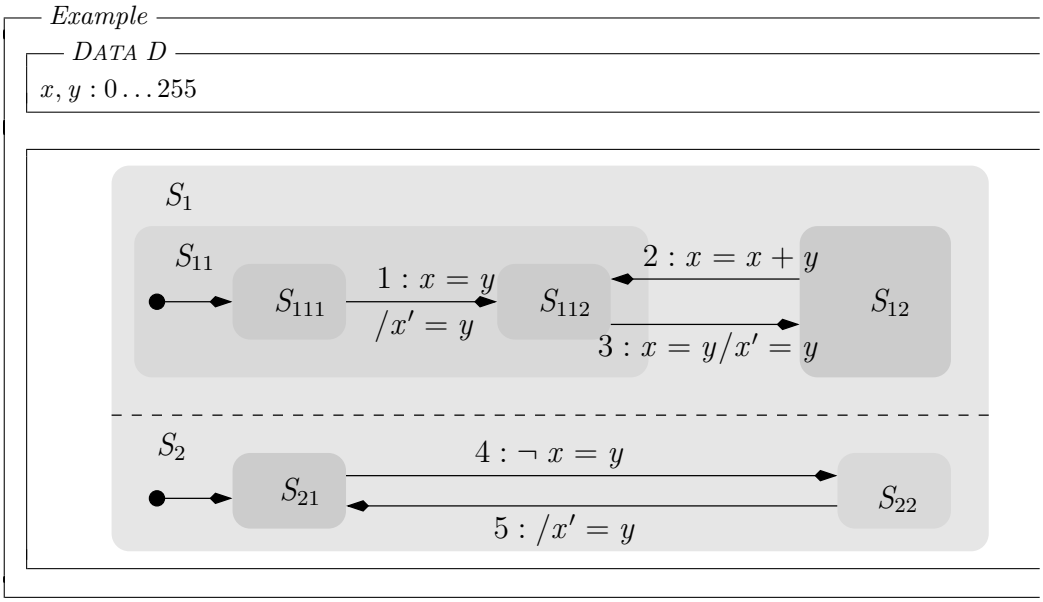


Figure 6.9: Example Class with Statechart

$$
\begin{aligned}
&\mathit{the\_root} == \mathit{And}(\mathit{root}, \{\, \mathit{the\_S_1}, \mathit{the\_S_2} \,\}) \\
&\mathit{the\_S_1} \;\; == \mathit{Xor}(S_1, \{\, t_2, t_3 \,\}, \{\, \mathit{the\_S_{11}}, \mathit{Basic}\ S_{12} \,\}) \\
&\mathit{the\_S_{11}} == \mathit{Xor}(S_{11}, \{\, t_1 \,\}, \{\, \mathit{Basic}\ S_{111}, \mathit{Basic}\ S_{112} \,\}) \\
&\mathit{the\_S_2} \;\; == \mathit{Xor}(S_2, \{\, t_4, t_5 \,\}, \{\, \mathit{Basic}\ S_{21}, \mathit{Basic}\ S_{22} \,\})
\end{aligned}
$$

$$
\begin{aligned}
&t_1 == (\{\, S_{111} \,\}, \{\, S_{112} \,\}, (x = y, \{\, y' = (x + y) \,\})) \\
&t_2 == (\{\, S_{11}, S_{112} \,\}, \{\, S_{12} \,\}, (x = y, \{\, x' = y \,\})) \\
&t_3 == (\{\, S_{12} \,\}, \{\, S_{11}, S_{111} \,\}, (x = (y + x), \{\, \mathit{true}, \mathit{true} \,\})) \\
&t_4 == (\{\, S_{21} \,\}, \{\, S_{22} \,\}, (\neg\, (x = y), \{\, \mathit{true} \,\})) \\
&t_5 == (\{\, S_{22} \,\}, \{\, S_{21} \,\}, (\mathit{true}, \{\, x' = y \,\}))
\end{aligned}
$$

Note that due to the default completion, transition t_3 consists of two transition segments. Therefore it has two actions: *true* and *true*. The transition also has two targets, since it enters state S_{11} and S_{111}. The interlevel transition t_2 has two source states: S_{11} and S_{112}, which have to be active if t_2 is to fire.

6.4.2 Augmentation of the Data Space

Two places are needed for the given Statechart, since the two parallel charts write x concurrently. For the states S_1 and S_2, the places named PLC_1 and PLC_2 are introduced, respectively. PLC_1 is also used for the variable y.

$$
\begin{array}{|l}
PLC_1, PLC_2 : special \\
STATE, D, T : NAME
\end{array}
$$

$$
\begin{array}{ll}
conf\_S_1 & == var_{\varnothing}(conf\ S_1) \\
conf\_S_1' & == var_{\varnothing}(prime(conf\ S_1)) \\
conf\_S_2 & == var_{\varnothing}(conf\ S_2) \\
conf\_S_2' & == var_{\varnothing}(prime(conf\ S_2)) \\
conf\_S_{11} & == var_{\varnothing}(conf\ S_{11}) \\
conf\_S_{11}' & == var_{\varnothing}(prime(conf\ S_{11}))
\end{array}
$$

$$
\begin{array}{ll}
st\_S_{11} & == var_{\varnothing}(st\ S_{11}) \\
st\_S_{12} & == var_{\varnothing}(st\ S_{12}) \\
st\_S_{21} & == var_{\varnothing}(st\ S_{21}) \\
st\_S_{22} & == var_{\varnothing}(st\ S_{22}) \\
st\_S_{111} & == var_{\varnothing}(st\ S_{111}) \\
st\_S_{112} & == var_{\varnothing}(st\ S_{112})
\end{array}
\qquad
\begin{array}{ll}
x_{lock} & == var_{\varnothing}(lock\ X) \\
y_{lock} & == var_{\varnothing}(lock\ Y) \\
plc_1 & == var_{\varnothing}\ PLC_1 \\
plc_2 & == var_{\varnothing}\ PLC_2
\end{array}
$$

$$
\begin{aligned}
STATEDECL == [STATE ::= & Const(st\ root) \mid Const(st\ S_1) \\
& \mid Const(st\ S_{11}) \mid Const(st\ S_{111}) \\
& \mid Const(st\ S_{112}) \mid Const(st\ S_{12}) \\
& \mid Const(st\ S_2) \mid Const(st\ S_{21}) \\
& \mid Const(st\ S_{22}) \mid true]_{\varnothing}
\end{aligned}
$$

$$
\begin{aligned}
DATADECL == Schema(D, Data, & conf\ S_1 : \{st\_S_{11}, st\_S_{12}\}; \\
& conf\ S_2 : \{st\_S_{21}, st\_S_{22}\}; \\
& conf\ S_{11} : \{st\_S_{111}, st\_S_{112}\}; \\
& lock\ X : \{plc_1, plc_2, var_{\varnothing}\ none\}; \\
& lock\ Y : \{plc_1, var_{\varnothing}\ none\}, \\
& (x_{lock} = var_{\varnothing}\ none \Rightarrow x' = x) \land \\
& (y_{lock} = var_{\varnothing}\ none \Rightarrow y' = y))
\end{aligned}
$$

6.4.3 Semantic Conversion

The abstract syntax of the Statechart shown in Figure 6.9 on the preceding page is defined by *the_root*. It is converted into a state transition relation, given in plain Z. According to the Z convention, the transition relation is defined as a predicate over the state variables representing the pre-state, and their primed counterparts, which represent the post-state. To do this, the configuration[2] of the Statechart has to be modelled. This is done in introducing a configuration variable for each xor-state that holds the currently active substate of the xor-state. A discussion on how the Statechart configuration can be modelled can be found in section 6.1.5 on page 76. In the chosen model, a state is active, if and only if it is the root state or:

- its parent state is active, and
- if its parent state is an xor-state, it is the parent's active sub-state defined by the configuration variable.

The function $conf \in NAME \rightarrowtail special$ maps xor-state names to the names of their respective configuration variables.

The transition relation is built up according to the Statechart hierarchy. For each state, a state transition relation is defined that is used if the respective state is active and

[2] The current state of a running Statechart is usually refered to as *configuration* in the literature, to avoid confusion with the Statechart states (i. e. S_1, S_{22}, etc. in the example). Some authors also use the term *status*.

no transition of an ancestor fires. In brief, an xor-state either fires one of its transitions, or, if this is not possible, behaves as its active sub-state. An and-state behaves as the conjunction of its sub-states, and a basic-state does nothing, except taking care that none of its places is used.

For each transition, two predicates (guard and action) are introduced. The guard is true, if the transition is able to fire, i. e. all source states are active and the transiton's guard is true. Note that only ancestors of the transition's arena state are considered. The complete firing condition is that the guard is true and the arena state is active. The action ensures that all target states are entered and executes the transition's action. The action also warrants that non-written variables are not locked by its places (see section 6.2.3 on page 79 for the concept of locks and places).

$$
\begin{aligned}
guard_1 &== conf\_S_{11} = st\_S_{111} \wedge x = y \\
action_1 &== conf\_S'_{11} = st\_S_{112} \wedge \\
&\quad \neg\,(y_{lock} = var_{\varnothing}\ none) \wedge \\
&\quad (y_{lock} = plc_1 \Rightarrow y' = x) \wedge \\
&\quad \neg\,(x_{lock} = plc_1) \\
guard_2 &== (conf\_S_1 = st\_S_{11}) \wedge (conf\_S_{11} = st\_S_{112}) \wedge (x = y) \\
action_2 &== conf\_S'_1 = st\_S_{12} \wedge \\
&\quad \neg\,(x_{lock} = var_{\varnothing}\ none) \wedge \\
&\quad (x_{lock} = plc_1 \Rightarrow x' = y) \wedge \\
&\quad \neg\,(y_{lock} = plc_1) \\
guard_3 &== (conf\_S_1 = st\_S_{12}) \wedge (x = (y + x)) \\
action_3 &== conf\_S'_1 = st\_S_{11} \wedge \\
&\quad conf\_S'_{11} = st\_S_{111} \wedge \\
&\quad \neg\,(x_{lock} = plc_1 \vee y_{lock} = plc_1) \\
guard_4 &== conf\_S_2 = st\_S_{21} \wedge \neg\,(x = y) \\
action_4 &== conf\_S'_2 = st\_S_{22} \wedge \neg\,(x_{lock} = plc_2) \\
guard_5 &== conf\_S_2 = st\_S_{22} \\
action_5 &== conf\_S'_2 = st\_S_{21} \wedge \neg\,(y_{lock} = var_{\varnothing}\ none) \wedge (x_{lock} = plc_2 \Rightarrow x' = y)
\end{aligned}
$$

The transition relation of a basic state is used only if no transition of its parallel activity can fire. Thus, none of the activity's locks can be used. This is ensured by the state transition relation of the basic states:

$$
\begin{aligned}
trans\_S_{12} &== \neg\,(x_{lock} = plc_1 \vee y_{lock} = plc_2) \\
trans\_S_{111} &== \neg\,(x_{lock} = plc_1 \vee y_{lock} = plc_2) \\
trans\_S_{112} &== \neg\,(x_{lock} = plc_1 \vee y_{lock} = plc_2) \\
trans\_S_{21} &== \neg\,(x_{lock} = plc_2) \\
trans\_S_{22} &== \neg\,(x_{lock} = plc_2)
\end{aligned}
$$

$$
\begin{aligned}
trans\_S_{11} == (&guard_1 \Rightarrow guard_1 \wedge action_1) \wedge \\
&(\neg\ guard_1 \Rightarrow conf\_S'_{11} = conf\_S_{11} \wedge \\
&\qquad (\neg\,(y_{lock} = plc_1)) \wedge \\
&\qquad (conf\_S_{11} = st\_S_{111} \Rightarrow trans\_S_{111}) \wedge \\
&\qquad (conf\_S_{11} = st\_S_{111} \Rightarrow trans\_S_{112}))
\end{aligned}
$$

$$trans\_S_1 == (guard_2 \lor guard_3 \Rightarrow (guard_2 \land action_2 \lor guard_3 \land action_3)) \land (\neg (guard_2 \lor guard_3) \Rightarrow conf\_S_1' = conf\_S_1 \land (conf\_S_1 = st\_S_{11} \Rightarrow trans\_S_{11}) \land (conf\_S_1 = st\_S_{12} \Rightarrow trans\_S_{12}))$$

$$trans\_S_2 == (guard_4 \lor guard_5 \Rightarrow (guard_4 \land action_4) \lor (guard_5 \land action_5)) \land (\neg (guard_4 \lor guard_5) \Rightarrow conf\_S_2' = conf\_S_2 \land (conf\_S_2 = st\_S_{21} \Rightarrow trans\_S_{21}) \land (conf\_S_2 = st\_S_{22} \Rightarrow trans\_S_{22}))$$

$$trans\_root == trans\_S_1 \land trans\_S_2$$

6.4.4 The Resulting Class

$$| result : NAME$$

$$Result == Class(result, STATEDECL; DATADECL; Schema(T, Transition, ex(var_{\varnothing}\ D);\ ex(var_{\varnothing}(prime\ D)), trans\_root))$$

Chapter 7

Z Rewriting

This Chapter describes the Z rewriting needed to prepare Z specifications for model checking. First, the overall rewriting strategy is introduced (section 7.2 on the next page), followed by presenting the target Z sub-language *Simple Z*. The objective of the rewriting is to translate Z into Simple Z (section 7.3 on page 102). Since only a subset of Z can be translated into Simple Z, a typing system is introduced to discriminate this subset and support the rewriting (section 7.4 on page 103). At last, the rewriting is defined (section 7.6 on page 121).

7.1 Introduction

Models that common model checkers can handle, are defined as Kripke structures. In section 4.8 on page 49 it was shown how a $\mu\mathcal{SZ}$ class defines a Kripke structure. Model checkers have their own input language to define the Kripke structures. Therefore, the $\mu\mathcal{SZ}$ class has to be translated into the input language of the model checker.

Alternatively, the $\mu\mathcal{SZ}$ class could be translated directly into BDDs. This has the advantage that specialized optimization can be applied. Some model checkers accept models defined by BDDs. Thus the verification algorithms of the model checker can still be used. However, BDD based model-checking can already fail when the state transition relation BDD is computed. Thus, the translation into BDDs is critical in itself and requires particular attention. This is a problem of its own, addressed well by the available model checkers. In order to use the BDD computations of the model checkers, their input language has to be used.

In Chapter 6 on page 78, it was shown how the Statecharts of an $\mu\mathcal{SZ}$ class can be translated into Z. The remaining problem is to translate Z into the model checker's input language.

The model checker languages are not as powerful as Z. With Z, it is possible to use variables with infinite value domains or formulate predicates that are not computable. Such predicates cannot be translated into a model checker language. Some Z expression that are computable, are still not supported because of the *one-to-one step mapping* (see

section 6.1.3 on page 74). It requires one Statechart step to be represented by exactly one step in the model. Consequently, all expressions have to be evaluated in one step. Therefore, computations such as recursive functions are not supported, either.

Some Z features can be interpreted as solely syntactical. These terms include sets and power sets ($\mathbb{P}$) , membership tests ($\in$), tuples, Cartesian products ($\times$), quantors ($\forall, \exists$), function applications, schemata and bindings. The objective of the Z rewriting is to translate these constructs into simple constructs that can be mapped directly into the model-checker's input language. The resulting Simple Z can then be translated directly into the input language of some model-checker. Expressions using these features can be translated into a model checker language. Providing such a translation has two advantages:

- The language that can be used is much more powerful than a model checker language. This allows a more natural specification of the system. With this, Z can also be used as superior input language for model checking.
- There is a greater chance that $\mu\mathcal{SZ}$ specifications can be model checked without adapting them. It is more likely that adaptations can be done locally. A local adaptation, for example, would be the introduction of bounds for a variable.

Translation of Z into the model checker language is done in two steps. First, Z is rewritten to *Simple Z*. Simple Z is a subset of Z that is designed to be as powerful as a model checker input language. Second, Simple Z is directly translated into the model checker language. Simple Z is defined in section 7.3 on page 102. The translation of Simple Z to SMV can be found in Chapter 8 on page 137.

7.2 Rewriting Strategy

This section gives an informal introduction to the Z rewriting. The rewriting will be formalized in the following sections of this Chapter.

The non-simple terms are translated into simple terms. Sometimes, this will lead to an explosion of the given specification's size. The aim is to support predicates such as $a \in \{ 1, 3 \} \cup \{ x : \mathbb{Z} \mid x \geq 10 \}$, which can be rewritten efficiently ($a = 1 \lor a = 3 \lor a \geq 10$). However, the rewriting also supports inefficient rewriting, such as $\forall x, y : 0 \mathinner{..} 255 \bullet P$, which causes 256^2 replications of the predicate P, since quantors are unfolded. It is the responsibility of the user to avoid such cases. A concrete translator can give some support, in stopping unwinding if some upper bound has been exceeded.

The rewriting is done in two steps. First, abbreviations, function applications, binding, and tuple selections are removed, on the level of expressions. Second, on the level of predicates, the remaining expressions are resolved according to their usage—in an element test or equality.

Abbreviations are replaced by their definitions. Function applications are restricted to functions, given as lambda abstractions. Therefore, application can be resolved by β-reduction. Function application and tuple selection is distributed through conditionals.

The rewriting rules are applied iteratively, until all terms are rewritten to simple terms.

7.2.1 State Variable Declaration

Variables of power-set or Cartesian product types are translated into enumerations in order to represent them. For instance a variable declaration $v : \mathbb{P}\{1, 2\}$ is translated to $v : \{ \mathit{empty}, \mathit{set\_1}, \mathit{set\_2}, \mathit{set\_1\_2} \}$. A variable declaration $v : \{1, 2\} \times \{2, 4\}$ is translated into $v : \{ \mathit{tuple\_1\_2}, \mathit{tuple\_1\_4}, \mathit{tuple\_2\_2}, \mathit{tuple\_2\_4} \}$. This encoding has to be resolved when the variables are applied, as it is done if a variable appears in a set element test. For example $a \in v$ has to be rewritten to:

$$(v = \mathit{set\_1} \Rightarrow a \in \{1\}) \land (v = \mathit{set\_2} \Rightarrow a \in \{2\}) \land \ldots$$

Variables of schema type (bindings) are resolved in flattening the binding rather then enumerating the values: A declaration $v : [a : A;\ b : B]$ is translated to $v\_a : A;\ v\_b : B$. This has the advantage that binding selections ($v.a$) can be supported much more efficiently. On the other hand, variables that are sets of bindings cannot be represented straightforward. Therefore such variables are not supported.

Actually, tuples and bindings are quite similar. Tuples can be represented by bindings with numbered variable names: $[e_1 : A;\ e_2 : B]$ for $A \times B$. In spite of their similarity, tuples and bindings are treated differently in this work. The reason for that is not a technical one; the objective is to evaluate the two approaches.

Variables of basic type $v : e$ (where e is an expression), are supported if e denotes a final set which can be computed at compile time. The declaration is transformed to $v : \mathit{maxset}(e)$ (where $\mathit{maxset}(e)$ denotes the maximal set of values e can take).

7.2.2 Expressions

Three kinds of operators have to be removed:

- *Function Application:* Function application of functions, defined by lambda abstraction, is removed by β-conversion: $(\lambda\, x : \mathbb{N} \bullet f)(e) = f[e/x]$. With this, it is not possible to handle recursive functions[1]
- *Tuple Selection:* Tuples have to be removed, and so has tuple selection, therefore. Tuple selection can be removed by $(a, b).1 = a$, if the selection is applied to a tuple. If it is applied to a tuple variable (e. g. $v : \{1, 2\} \times \{2, 4\}$) things becomes more complicated. In this case, the variable v holds the encoded value pairs, as described above: $v : \{ \mathit{tuple\_1\_2}, \ldots \}$. In a tuple selection, the values have to be decoded: $v.1 = \textbf{if}\ v = \mathit{tuple\_1\_2}\ \textbf{then}\ 1\ \textbf{else} \ldots$.
- *Binding Selection:* Similar to tuples, binding selections have to be removed. If a binding selection is applied to a binding display (e. g. $[a == 4, b == 3]$) it is easy: $[a == 4, b == 3].b = 3$. If the binding is a variable ($v : [a : A;\ b : B]$), this variable is represented by two variables: v_a and v_b (see above). The binding selection can then be resolved: $v.b = v\_b$.

[1] Recursive functions could be supported if the maximum number of recursion steps were finite and known. In this case, the β-conversion could be applied iteratively. However, rather sophisticated static analysis would then be required.

Apart from the the cases described, the operators can also appear in other expressions such as conditionals: $(\mathbf{if}\ a = 4\,\mathbf{then}(a, b)\,\mathbf{else}(a, 5)).1$. The selection (or application) is then pushed into the term: $\mathbf{if}\ a = 4\,\mathbf{then}(a, b).1\,\mathbf{else}(a, 5.1)$.

7.2.3 Element Test

Sets and thus element tests, have to be removed entirely. The applied translation depends on the syntax of the tested set:

- *Set Display:* A predicate $e \in \{ e_1, \ldots e_n \}$ is translated into $e = e_1 \lor \ldots \lor e = e_n$.
- *Set Comprehension:* A predicate $e \in \{ x : E \mid P \}$ is translated into the predicate P, where x is replaced by e: $P[e/x]$. Additionally, e has to be a member of E, the resulting predicate is thus $e \in E \land P[e/x]$.
- *Cartesian Product:* The predicate $e \in E_1 \times \ldots \times E_n$ is translated into $e.1 \in E_1 \land \ldots \land e.n \in E_n$.
- *Power-Set:* The predicate $e \in \mathbb{P}\, E$ is translated into $\forall x : \mathit{maxset}(e) \bullet x \in e \Rightarrow x \in E$ (Note that $e \in \mathbb{P}\, E$ is equal to $e \subseteq E$). Where $\mathit{maxset}(e)$ denotes all values the expression e can take. That means if e is a variable $v : \mathbb{P}\{ 1, 2 \}$ $\mathit{maxset}(e) = \mathit{maxset}(v) = \{ 1, 2 \}$. In contrary to set comprehension, power-set element tests are only supported, if both the tested element and the set are finite and known at compile time. The type system presented in section 7.4 on page 103 supports the computation of *maxset*.
- *Trivial Test:* For some free- or given type T, $e \in T$ is translated to *true* (this is important for sets such as $\{ x : \mathbb{Z} \mid x \geq 0 \}$).
- *Variables:* As already noted, variables of power-set types are represented as enumerations, representing all possible sets. For example a declaration $v : \mathbb{P}\{ 1, 2 \}$ would be translated into $v : \{ \mathit{empty}, \mathit{set\_1}, \mathit{set\_2}, \mathit{set\_1\_2} \}$. The element test of such sets $(e \in v)$ is translated into: $(v = \mathit{empty} \Rightarrow e \in \varnothing) \land (v = \mathit{set\_1} \Rightarrow e \in \{ 1 \}) \land \ldots$.

7.2.4 Equality

- *Simple Expressions:* A predicate $e = e'$ (where e and e' are of simple type: neither schema, product nor power-set) is not changed.
- *Tuples:* The predicate $e = e'$ (where e and e' are of type $\tau_1 \times \ldots \times \tau_n$) is translated to $e.1 = e'.1 \land \ldots \land e'.n$. If both expressions are variables, they can be compared directly.
- *Bindings:* A predicate $e = e'$ (where e and e' are of type $[v_1 : T_1;\ \ldots v_n : T_n]$) is translated to $e.v_1 = e'.v_1 \land \ldots \land e.v_n = e'.v_n$
- *Power-Set:* The predicate $E = E'$ (where E and E' are of type $\mathbb{P}\,\tau$) is translated into $\forall x : \mathit{maxset}(E) \cup \mathit{maxset}(E') \bullet x \in E \Leftrightarrow x \in E'$. If E and E' are variables, the encoded values can be compared directly. No rewriting is required.

7.2.5 Quantors

- *All Quantor:* The predicate $\forall x : e \bullet p$ is translated into $(e_1 \in e \Rightarrow p[e_1/x]) \land \ldots \land (e_n \in e) \Rightarrow p[e_n/x])$, where $maxset(e) = \{ e_1, \ldots, e_n \}$.
- *Exists Quantor:* The predicate $\exists x : e \bullet p$ is translated into $(e_1 \in e \land p[e_1/x]) \lor \ldots \lor (e_n \in e \land p[e_n/x])$, where $maxset(e) = \{ e_1, \ldots, e_n \}$.

7.2.6 Set Union, Intersection, etc.

Set union, intersection, and set-minus do not need special treatment. They only have to be defined as lambda-abstraction:

$$
\begin{array}{|l}
\hline
[X] \\
\_ \cup \_ == (\lambda A, B : \mathbb{P}\, X \bullet \{ x : X \mid x \in A \lor x \in B \} \\
\_ \cap \_ == (\lambda A, B : \mathbb{P}\, X \bullet \{ x : A \mid x \in B \} \\
\_ \setminus \_ == (\lambda A, B : \mathbb{P}\, X \bullet \{ x : A \mid \neg\, x \in B \} \\
\_ \subseteq \_ == \{ A, B : \mathbb{P}\, X \mid A \in \mathbb{P}\, B \} \\
\hline
\end{array}
$$

The different rewriting steps are now illustrated with an example:

	$a \in \{ 1, 3 \} \cup \{ n : \mathbb{Z} \mid n > 10 \}$	
$\equiv$	$a \in \{ x : \mathbb{Z} \mid x \in \{ 1, 3 \} \lor x \in \{ n : \mathbb{Z} \mid n > 10 \} \}$	abbreviation substitution and β-conversion
$\equiv$	$a \in \mathbb{Z} \land (a \in \{ 1, 3 \} \lor a \in \{ n : \mathbb{Z} \mid n > 10 \})$	set-comprehension rewriting
$\equiv$	$a \in \{ 1, 3 \} \lor a \in \{ n : \mathbb{Z} \mid n > 10 \}$	trivial set rewriting
$\equiv$	$a = 1 \lor a = 3 \lor a \in \{ n : \mathbb{Z} \mid n > 10 \}$	set-display rewriting
$\equiv$	$a = 1 \lor a = 3 \lor a > 10$	set-comprehension rewriting

7.2.7 Undefinedness

When the expressions are rewritten, definedness has to be considered. Consider for example the following function application: $(\lambda x : \mathbb{Z} \mid x > 3 \bullet x - 2)(1)$ which is undefined in Z. Undefinedness constitutes a problem, since common symbolic model checkers do not support undefinedness or three-valued-logic.[2] Therefore, atomic predicates (equality and element test) are defined to be false in case one of their expressions is undefined. The Z semantics do not define how undefinedness is handled, but grant tool developers the freedom to choose a definition. Thus, the handling of undefinedness complies with the Z semantics.

Definedness is computed separately for every expression and extra predicates are generated to handle it. An atomic predicate is defined to be false, if one of the contained expressions is not defined. For example the predicate $(\lambda x : \mathbb{Z} \mid x > 3 \bullet x - 2)(a) = 1$ is only true, if $a > 3$. It is thus translated to $a - 2 = 1 \land a > 3$. The definedness predicate of an expression has to be propagated to the predicate that uses the expression. The definedness predicate is then conjuncted with the using predicate.

[2] In principle, it would be possible to extend the BDD by a third value and support undefined predicates.

7.3 Simple Z

section *SimpleZ* **parents** *Syntax*

Simple Z represents the indeed simple logic that is supported by a model checker. It is designed in such a way that a direct translation into the input language of a symbolic model checker is possible. In Chapter, 8 on page 137 the translation to the SMV input language is shown. It has also been verified with other model checkers, such as VIS [2], that a direct translation into the input language is possible. Moreover, with the exception of the built-in functions, the logic corresponds to the capabilities offered by BDDs. Therefore, any BDD based model checker supports this logic. However, some model checkers do not allow the direct definition of the state transition relation as a predicate. These checkers cannot be used.

The built-in functions still raise a problem. The SMV model checker supports enumerations, integers, and integer arithmetic (less, greater, plus, minus). The VIS checker, for example, does not support this functionality. It supports only Boolean variables. Integers have to be encoded as sets of Boolean variables (or bits) and integer arithmetic by corresponding logic. If the checker does not support the built-in functions, their occurrences have to be removed in a second rewriting step. Not removing the built-ins has the advantage of making it much easier to read and to debug the generated code.

The translation of Z to simple Z is described in section 7.6 on page 121.

Simple Z supports equality of expressions, logic operators, built-in relations, conditionals, and built-in functions. The built-in are defined in section 5.1 on page 55. See section 3.3 on page 32 for the mathematical definitions used.

$$
\begin{array}{l}
Pred_f : \mathbb{P}\, Pred;\ Expr_f : \mathbb{P}\, Expr \\
\hline
Pred_f \overset{Pred}{\leftarrow} \{\, true, false \,\} \\
\qquad \| \ \{\, e, e' : Expr_f \bullet e = e' \,\} \\
\qquad \| \ \{\, bt : BinType;\ p, q : Pred_f \bullet bin(bt, p, q) \,\} \\
\qquad \| \ \{\, v : BuiltinRel;\ e : Expr_f \bullet e \in var_{\varnothing}\ v \,\} \quad \oplus \\
Expr_f \overset{Expr}{\leftarrow} var_{\varnothing}(\!| NAME |\!) \\
\qquad \| \ \{\, p : Pred_f;\ e, e' : Expr_f \bullet \textbf{if}\ p\ \textbf{then}\ e\ \textbf{else}\ e' \,\} \\
\qquad \| \ \{\, v : BuiltinFun;\ \vec{e} : \mathrm{seq}_1\, Expr_f \bullet (var_{\varnothing}\ v)\ \omega\ ((,)\ \vec{e}) \,\}
\end{array}
$$

Expressions used in declarations are restricted even more. Basically, only three kinds of declarations are admissible:

- Numbers: $a : 0 \mathinner{..} 255$
- Enumeration of constants: $a : \{\, c_1, c_2, c_3 \,\}$ (if c_1, c_2, and c_3 are branches of a free type).
- Declarations of free types.

Expressions allowed in declarations are defined by *DeclExpr*. The context rules are omitted.

$$
\begin{array}{|l}
Decl_f : \mathbb{P}\, Decl \\
\hline
\exists\, DeclExpr : \mathbb{P}\, Expr \bullet \\
\quad DeclExpr \overset{Expr}{\leftarrow} \{\, e_{min}, e_{max} : var_{\varnothing}(\!|\, Number \,|\!) \bullet e_{min} \mathinner{..} e_{max} \,\} \land \\
\qquad\qquad \| \ \{\, \vec{e} : \mathrm{seq}_1(var_{\varnothing}(\!|\, NAME \,|\!)) \bullet \{,,\}\vec{e} \,\} \\
\quad Decl_f \overset{Decl}{\leftarrow} \{\, n : NAME;\ e : DeclExpr \bullet n{:}\, e \,\} \\
\qquad\qquad \| \ \{\, F : NAME;\ b : Branch \bullet F ::= b \,\} \\
\qquad\qquad \| \ \{\, d_1, d_2 : Decl_f \bullet d_1;\ d_2 \,\}
\end{array}
$$

Only axiomatic definitions and schemata are allowed in specifications. Plain schemata are not allowed, because they have no meaning to the semantics of a $\mu\mathcal{SZ}$ class and cannot be referenced. Schema references have to be resolved in simple Z.

$$
\begin{array}{|l}
Spec_f : \mathbb{P}\, Spec \\
\hline
Spec_f \overset{Spec}{\leftarrow} \{\, d : Decl_f;\ p : Pred_f \bullet [\, d \mid p \,]_{\varnothing} \,\} \\
\qquad\qquad \| \ \{\, n : NAME;\ type : Stype \setminus \{\, Plain \,\};\ d : Decl_f;\ p : Pred_f \bullet \\
\qquad\qquad\quad Schema(n, type, d, p) \,\} \\
\qquad\qquad \| \ \{\, s_1, s_2 : Spec_f \bullet s_1;\ s_2 \,\}
\end{array}
$$

7.4 Annotated Type System for Enumerable Expressions

In this section, the exact supported subset of Z is defined. The decision, whether a Z expression can be supported or not, depends on context information. Therefore, a type system is established. This section defines the supported subset of Z via a type system. It also shows how the *maxset* function used in the previous sections is computed.

7.4.1 The Type System

section *TypeDecl* **parents** *Aux,Syntax,Name*

The *annotated type system of enumerable Z* describes the subset of Z that is supported by the rewriter and the translator. To establish enumerability, basic types are annotated with a set of values (*maxset*). This is the set of values an expression of this type can take. The set is an upper bound of the actual values that the expression can take, considering the complete static and dynamic semantics. (See Table 7.1 on the next page for examples.) This set of values corresponds to the *maxset* that was introduced above (see section 7.2 on page 98). The value set annotation is used to determine whether an expression has a finite and known value domain and to compute this very value domain.

The Z type system consists of basic types and three type constructors. Basic types are introduced by given type declarations (i. e. $[NewType]$) and free type declarations (i. e. $FreeType ::= c_1 \mid c_2$). A type also represents the set of values of this very type (its *carrier set*). For given types, this set is not well defined, which means the specification is *loose* with respect to the carrier set of this type. It can be empty, finite, or infinite. For free types, the set of values is built by a free construction over its branches. If only non-parametrized free types are considered, the carrier set is the set of branches (i. e. $FreeType = \{\, c_1, c_2 \,\}$).

Expression	Values
a	$\{1, 2, 4\}$
$a + 3$	$\mathbb{Z}$
3	$\{3\}$
$\{a, 7\}$	$\mathbb{P}\{1, 2, 4, 7\}$
$(\lambda x : \{1, 2, 4\} \bullet \{x, 4\})\, e$	$\mathbb{P}\{1, 2, 4\}$
$\{x : \{1, 2, 3, 4\} \mid x > 2\}$	$\mathbb{P}\{1, 2, 3, 4\}$
$\{x : \{a, 4\} \mid x > 2\}$	$\mathbb{P}\{1, 2, 4\}$

Table 7.1: Value annotations of expressions. a is assumed to be data variable, declared as $a : \{1, 2, 4\}$.

In Z, variables are not declared with their type but with an expression, also called *type expression*. A type expression must be of a power-set type. It defines all values the variable can have (its *value domain*). The set, defined by the type expression, can be smaller than the carrier set. It may happen (e. g. for numbers: $x : \{1, 2\}$) that the expression denotes a finite set while the carrier set is infinite. Therefore, the type expression of a variable declaration is used, rather then the carrier set of its type, to determine the variable's value domain. For this, the set of values that an expression can have, is needed.

The data space of a model is finite if and only if the value domains of all data variables are finite. Therefore, the value domains of the data variables are already needed for computability analysis.

In order to compute the set of values an expression can have, the type of an expression is annotated with those very values. For a constant, this is the constant itself, and for a given type, it is its carrier set. A variable is annotated with its value domain, according to its declaration.

Because of the not well defined carrier set, support of free types is quite intricate. It can be provided easily for numbers, where the carrier set is known. The type system supports given types. However, since no constants of a type exist, no expression of finite value domain can be constructed. Numbers are added as special cases to the initial environment. Therefore, no variable of a free type (except numbers) can be declared.

More complex types can be constructed from the basic types by the three type constructors. These are: power set, Cartesian product, and schema type. From the set of values of the basic types, the set of values of the constructed types can be computed. A type, that is constructed only from types with finite value domains, has a finite value domain itself. The number of values can grow exponentially, however.

Function applications have to be resolved statically. Therefore, no recursion is permitted in function applications, and function application is only allowed to lambda abstraction. Therefore, function applications can be resolved by the well known rewriting techniques of the lambda calculus. For function application to functions that are not defined as lambda abstraction, auxiliary variables and existential quantification would be

needed. In order to discriminate functions defined by lambda abstractions from sets of tuples, an extra type constructor for function is introduced. A function type is constructed of the types of the parameters and the result type.

A basic type is constructed by the name of the (given or free) type and the set of values. If the set of values is finite, it holds either numbers ($Number$) or constants, introduced by free types. In the case of given types, the set of values is infinite and, for technical reasons, equals the set of all names $NAME$. For a constant c of a free type F, the type is $\mathsf{basic}(F, \{ c \})$. The type of F is $\mathsf{power}(\mathsf{basic}(F, \{ b_1, \ldots, b_n \}))$, if $F ::= b_1 \mid \ldots \mid b_n$.

The type constructors take only a finite number of parameters. This is ensured for the Cartesian product and function type, since Z sequences are always finite. Finiteness of the set of parameters of a schema has to be ensured explicitly.

$$
\begin{aligned}
\mathsf{Type} ::= {} & \mathsf{basic}\langle\!\langle NAME \times \mathbb{P}\, NAME \rangle\!\rangle \\
\mid {} & \mathsf{power}\langle\!\langle \mathsf{Type} \rangle\!\rangle \\
\mid {} & \mathsf{prod}\langle\!\langle \mathrm{seq}_1\, \mathsf{Type} \rangle\!\rangle \\
\mid {} & \mathsf{fun}\langle\!\langle \mathrm{seq}_1\, \mathsf{Type} \times \mathsf{Type} \rangle\!\rangle \\
\mid {} & \mathsf{schema}\langle\!\langle (NAME \pfun \mathsf{Type}) \cap \mathsf{dom}(\#) \rangle\!\rangle
\end{aligned}
$$

The type of numbers without any value constrains is denoted by number.

$$\mid \mathsf{number} == \mathsf{basic}(\mathbb{Z}, NAME)$$

7.4.2 Translating Types to Expressions

The function typeToExpr translates a type back into a Z expression of that type. In order to declare a variable v that has the extended type $\tau \in \mathsf{Type}$, the declaration $v{:}\,\mathsf{typeToExpr}\ \tau$ can be used. This function preserves the enumerability of the type. For this, a basic type with a finite value set annotation is translated into a set display: $\mathsf{typeToExpr}(\mathsf{basic}(Num, \{ 1, 2, 3 \})) = \{ 1, 2, 3 \}$.

$$
\begin{array}{|l}
\mathsf{typeToExpr} : \mathsf{Type} \fun Expr \\
\hline
\forall\, n : NAME;\ N : \mathbb{P}\, NAME \bullet \\
\quad \mathsf{typeToExpr}(\mathsf{basic}(n, N)) = \\
\quad\quad \mathbf{if}\ N \in \mathbb{F}\, NAME\ \mathbf{then} \{ , , \} (var_{\varnothing} \circ order\ \ N)\ \mathbf{else}\ var_{\varnothing}\ n \\
\forall\, \tau : \mathsf{Type} \bullet \mathsf{typeToExpr}(\mathsf{power}\ \tau) = \mathbb{P}(\mathsf{typeToExpr}\ \tau) \\
\forall\, \vec{\tau} : \mathrm{seq}_1\, \mathsf{Type} \bullet \mathsf{typeToExpr}(\mathsf{prod}\ \vec{\tau}) = prod(\mathsf{typeToExpr} \circ \vec{\tau}) \\
\forall\, \vec{\tau} : \mathrm{seq}_1\, \mathsf{Type};\ \tau : \mathsf{Type} \bullet \\
\quad \mathsf{typeToExpr}(\mathsf{fun}(\vec{\tau}, \tau)) = prod\langle prod(\mathsf{typeToExpr} \circ \vec{\tau}), \mathsf{typeToExpr}\ \tau \rangle \\
\forall\, \tau_{sig} : NAME \pfun \mathsf{Type} \bullet \\
\quad \exists\, d == \mathsf{setreduce}\ (\_;\ \_)\ ((\_ : \_)(\lvert \mathsf{typeToExpr} \circ \tau_{sig} \rvert)) \bullet \\
\quad\quad \mathsf{typeToExpr}(\mathsf{schema}(\tau_{sig})) = [d \mid true]
\end{array}
$$

7.4.3 Compatible Types

Because of the annotation, the annotated type system also discriminates type compatible expressions, if they have different value domains. The predicate comp defines the sets of compatible types, i. e. the types that are equal without regarding the value annotation.

Types are compatible if they are basic types of the same given or free type ($\mathsf{basic}(\!|\{ v \} \times \mathbb{P}\, NAME|\!)$ for some given or free type name v) plus all types that can be built with the type constructors from these types.

comp is defined with the fixed point macros (section 3.3 on page 32). Thus, comp is the smallest set that contains the five sets denoted below.

$$\mathsf{comp} : \mathbb{P}(\mathbb{P}\,\mathsf{Type})$$

$$\begin{aligned}
\mathsf{comp} \overset{\mathbb{P}\,\mathsf{Type}}{\longleftarrow} & \{ v : NAME;\ N : \mathbb{P}\, NAME \bullet \mathsf{basic}(\!|\{ v \} \times \mathbb{P}\, N|\!) \} \\
\| & \{ T : \mathsf{comp} \bullet \mathsf{power}(\!|T|\!) \} \\
\| & \{ \vec{T} : \mathrm{seq}_1\, \mathsf{comp} \bullet \mathsf{prod}(\!|\mathsf{explode}\ \vec{T}|\!) \} \\
\| & \{ \vec{T} : \mathrm{seq}_1\, \mathsf{comp};\ T : \mathsf{comp} \bullet \\
& \quad \mathsf{fun}(\!|(\mathsf{explode}\ \vec{T}) \times T|\!) \} \\
\| & \{ T_{sig} : NAME \pfun \mathsf{comp} \bullet \mathsf{schema}(\!|\mathsf{explode}\ T_{sig}|\!) \}
\end{aligned}$$

7.4.4 Unification of Compatible Types

The members of e. g. a set comprehension $\{ a, b \}$, have to be compatible, but may be unequal. In order to determine the type of the set, a unification function is needed. The unification computes from a set $T \in \mathsf{comp}$ of compatible types a unified type $\mathsf{uni}^{\cup}\ T$. The unification function $\mathsf{uni}^{\cup}$ basically computes the union of all possible values of the unified types. Obviously, the unified type is supposed to be compatible to the types it unifies, thus $\{ \mathsf{uni}^{\cup}\ T \} \cup T \in \mathsf{comp}$.

$$\mathsf{uni}^{\cup} : \mathsf{comp} \to \mathsf{Type}$$

$$\begin{aligned}
\forall\, & T : \mathsf{comp} \bullet \\
& (T \subseteq \mathsf{ran}\,\mathsf{basic} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{basic}((\mu\, v : (\mathit{first} \circ \mathsf{basic}^{\sim})(\!|T|\!)), \\
& \qquad \bigcup((\mathit{second} \circ \mathsf{basic}^{\sim})(\!|T|\!))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{power} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{power}(\mathsf{uni}^{\cup}(\mathsf{power}^{\sim}(\!|T|\!)))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{prod} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{prod}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathsf{prod}^{\sim})(\!|T|\!)))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{fun} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{fun}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathit{first} \circ \mathsf{fun}^{\sim})(\!|T|\!)), \wedge \\
& \qquad \mathsf{uni}^{\cup}((\mathit{second} \circ \mathsf{fun}^{\sim})(\!|T|\!)))) \\
& (T \subseteq \mathsf{ran}\,\mathsf{schema} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{schema}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathsf{schema}^{\sim})(\!|T|\!))))
\end{aligned}$$

Whereas $\mathsf{uni}^{\cup}$ computes the maximal set of values, $\mathsf{uni}^{\cap}$ computes the minimal set of values.

$$\mathsf{uni}^{\cap} : \mathsf{comp} \to \mathsf{Type}$$

$$\begin{aligned}
\forall\, & T : \mathsf{comp} \bullet \\
& (T \subseteq \mathsf{ran}\,\mathsf{basic} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{basic}((\mu\, v : (\mathit{first} \circ \mathsf{basic}^{\sim})(\!|T|\!)), \\
& \qquad \bigcap((\mathit{second} \circ \mathsf{basic}^{\sim})(\!|T|\!))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{power} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{power}(\mathsf{uni}^{\cup}(\mathsf{power}^{\sim}(\!|T|\!)))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{prod} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{prod}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathsf{prod}^{\sim})(\!|T|\!)))) \wedge \\
& (T \subseteq \mathsf{ran}\,\mathsf{fun} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{fun}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathit{first} \circ \mathsf{fun}^{\sim})(\!|T|\!)), \wedge \\
& \qquad \mathsf{uni}^{\cup}((\mathit{second} \circ \mathsf{fun}^{\sim})(\!|T|\!)))) \\
& (T \subseteq \mathsf{ran}\,\mathsf{schema} \Rightarrow \mathsf{uni}^{\cup}\ T = \mathsf{schema}(\mathsf{uni}^{\cup} \circ \mathsf{implode}((\mathsf{schema}^{\sim})(\!|T|\!))))
\end{aligned}$$

7.4.5 Enumerable Types and Expressions

For the rewriting, expressions with a finite domain are of interest. The type system describes both expressions with finite and infinite domain. The restricted set of types, describing only expressions of finite domains, is defined by $Type_c$. These types (or expressions) are called *enumerable* in the following. Enumerable types are basic types, annotated with a finite set of values and all types that can be constructed from these types by the type constructors (the power set and Cartesian product). Function types are defined to be not enumerable. The objective of the typing system it to preserve enumerability and to compute all enumerable expressions. Because of the special treatment of schema variables (see section 7.2 on page 98), schema types are only allowed at outermost position. That is: Sets of schemata are defined to be not enumerable.

$$
\begin{array}{|l}
Type_c : \mathbb{P}\,\mathsf{Type} \\
\hline
\exists\, type : \mathbb{P}\,\mathsf{Type} \bullet \\
\quad type \overset{\mathsf{Type}}{\leftarrow} \mathsf{basic}(\!| NAME \times \mathbb{F}_1\, NAME |\!) \\
\qquad \| \;\; \mathsf{power}(\!| type |\!) \\
\qquad \| \;\; \mathsf{prod}(\!| \mathrm{seq}_1\, type |\!) \\
\quad \land \\
\quad Type_c = type \cup \mathsf{schema}(\!|(NAME \pfun type) \cap \mathsf{dom}\,\#|\!)
\end{array}
$$

7.4.6 Characteristic Tuples and Signatures

section *Type* **parents** *Aux, Syntax, Environment, TypeDecl, Statecharts, ModelChecking*

The Z type system introduces the notion of a *characteristic tuple* and a *signature*.

The type of a set that is defined by a set comprehension $\{ d \mid p \}$, is the characteristic tuple, defined by the declaration d. For example, the characteristic tuple of the declaration $a : A;\ b : B$ is $A \times B$. Or, more precisely: $type(\{ a : A;\ b : B \}) = \mathsf{power}(\mathsf{prod}\langle type\ A, type\ B\rangle)$.

The type of a schema is the signature, defined by the declaration. This is: $type([a : A;\ b : B]) = \mathsf{power}(\mathsf{schema}\{ a \mapsto type\ A, b \mapsto type\ B \})$.

Note that, depending on whether the characteristic tuple or the signature is needed, the declarations are treated differently. In the first case, the order of the declaration is considered and the names of the variables are ignored, whereas in the second case, the order does not matter, but the names do. Moreover, schema expressions are treated differently, since only in signatures, schema expressions are flattened: $\mathsf{sig}^\tau([d \mid p]) = \mathsf{sig}^\tau(d)$ while $\mathsf{ct}^\tau([d \mid p]) = \mathsf{schema}(\mathsf{sig}^\tau(d))$.

The type of a set comprehension is defined by the characteristic tuple of its declaration part. In order to evaluate its predicate, the environment has to be changed according the signatures of the declarations. This is a somewhat confusing concept of Z, since in order to determine the type of the set, the characteristic tuple is used, and in order to determine which variables are declared, the signature is used.

The function ct^τ computes the characteristic tuple of a declaration as a sequence of types. For a declaration that contains variable declarations, this sequence is not empty.

For the type of a set comprehension with declaration part d is $\mathsf{power}(\mathsf{prod}(\mathsf{ct}^\tau\ d))$.

$$
\begin{array}{l}
\mathsf{ct}^\tau : \mathit{Env} \rightarrow \mathit{Decl} \rightarrow \mathsf{seq}\,\mathsf{Type} \\
\hline
\forall\, \mathcal{E} : \mathit{Env};\ n : \mathit{NAME};\ e : \mathit{Expr};\ b : \mathit{Branch};\ d, d' : \mathit{Decl} \bullet \\
\quad \mathsf{ct}^\tau(\mathcal{E})(n == e) = \varnothing \land \\
\quad \mathsf{ct}^\tau(\mathcal{E})(n ::= b) = \varnothing \land \\
\quad \mathsf{ct}^\tau(\mathcal{E})(\mathit{Given}\ n) = \varnothing \land \\
\quad (\exists \tau : \mathsf{Type} \mid \mathcal{E} \vdash_E e : \mathsf{power}\,\tau \bullet \mathsf{ct}^\tau(\mathcal{E})(n{:}\ e) = \langle \tau \rangle) \land \\
\quad \mathsf{ct}^\tau(\mathcal{E})(d;\ d') = (\mathsf{ct}^\tau\ \mathcal{E}\ d) \frown (\mathsf{ct}^\tau\ \mathcal{E}\ d') \land \\
\quad \mathsf{ct}^\tau(\mathcal{E})(\mathit{ex}(e)) = \langle \mathsf{schema}(\mathsf{sig}^\tau\ \mathcal{E}\ d) \rangle
\end{array}
$$

The function sig^τ computes the signature of a declaration as a mapping from variable names to their types. $\mathsf{power}(\mathsf{schema}(\mathsf{sig}^\tau\ d))$ is the type of a schema expression $[d \mid p]$.

$$
\begin{array}{l}
\mathsf{sig}^\tau : \mathit{Env} \rightarrow \mathit{Decl} \rightarrow (\mathit{NAME} \pfun \mathsf{Type}) \\
\hline
\forall\, \mathcal{E} : \mathit{Env};\ n : \mathit{NAME};\ e, e' : \mathit{Expr};\ b : \mathit{Branch};\ d, d' : \mathit{Decl};\ p : \mathit{Pred} \bullet \\
\quad \mathsf{sig}^\tau(\mathcal{E})(n == e) = \varnothing \land \\
\quad \mathsf{sig}^\tau(\mathcal{E})(n ::= b) = \varnothing \land \\
\quad \mathsf{sig}^\tau(\mathcal{E})(\mathit{Given}\ n) = \varnothing \land \\
\quad (\exists \tau : \mathsf{Type} \mid \mathcal{E} \vdash_E e : \mathsf{power}\,\tau \bullet \mathsf{sig}^\tau(\mathcal{E})(n{:}\ e) = \{ n \mapsto \tau \}) \land \\
\quad \mathsf{sig}^\tau(\mathcal{E})(d;\ d') = (\mathsf{sig}^\tau\ \mathcal{E}\ d) \oplus (\mathsf{sig}^\tau\ \mathcal{E}\ d') \land \\
\quad (\exists \tau_{sig} : \mathit{NAME} \pfun \mathsf{Type} \mid \mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig}) \bullet \mathsf{sig}^\tau(\mathcal{E})(\mathit{ex}\ e) = \tau_{sig})
\end{array}
$$

7.4.7 Reverting Types

In schema negation, all value domain restriction gets lost. This will be explained later. The function revert removes all restrictions from the value domain annotations of a given type. The formal definition of the function's domain is omitted. For some environment $\mathcal{E}$ all types τ are in the domain of revert if and only if all free and given types occuring in τ are declared in $\mathcal{E}$.

$$
\begin{array}{l}
\mathsf{revert} : \mathit{Env} \rightarrow \mathsf{Type} \pfun \mathsf{Type} \\
\hline
\forall\, \mathcal{E} : \mathit{Env} \bullet \\
\quad (\forall\, n : \mathit{NAME};\ N : \mathbb{P}\, \mathit{NAME} \bullet \\
\qquad \mathsf{revert}\ \mathcal{E}(\mathsf{basic}(n, N)) = \mathsf{basic}(n, ((\mathsf{basic}^\sim)(\mathit{getType}\ \mathcal{E}\ (n, \varnothing))).2)) \land \\
\quad (\forall \tau : \mathsf{Type} \bullet \mathsf{revert}\ \mathcal{E}\ (\mathsf{power}\,\tau) = \mathsf{power}(\mathsf{revert}\ \mathcal{E}\ (\mathsf{revert}\ \mathcal{E}\ \tau))) \land \\
\quad (\forall \vec{\tau} : \mathsf{seq}_1\ \mathsf{Type} \bullet \mathsf{revert}\ \mathcal{E}\ (\mathsf{prod}\ \vec{\tau}) = \mathsf{prod}((\mathsf{revert}\ \mathcal{E}) \circ \vec{\tau})) \land \\
\quad (\forall \vec{\tau} : \mathsf{seq}_1\ \mathsf{Type};\ \tau : \mathsf{Type} \bullet \\
\qquad \mathsf{revert}\ \mathcal{E}\ (\mathsf{fun}(\vec{\tau}, \tau)) = \mathsf{fun}((\mathsf{revert}\ \mathcal{E}) \circ \vec{\tau}, \mathsf{revert}\ \mathcal{E}\ \tau)) \land \\
\quad (\forall \tau_{sig} : \mathit{NAME} \pfun \mathsf{Type} \bullet \mathsf{revert}\ \mathcal{E}\ (\mathsf{schema}\ \tau_{sig}) = \mathsf{schema}((\mathsf{revert}\ \mathcal{E}) \circ \tau_{sig}))
\end{array}
$$

7.4.8 Type System Relations

The type system is defined via two relations. For some environment, $\vdash_P$ defines the supported predicates, and $\vdash_E$ assigns types to expressions.

relation $(\_ \vdash_E \_ : \_)$
relation $(\_ \vdash_P \_)$

$$\_\vdash_E \_ : \_ : \mathbb{P}(Env \times Expr \times \mathsf{Type})$$
$$\_\vdash_P \_ : \mathbb{P}(Env \times Pred)$$

With these relations, the sets of computable expressions and predicates can be defined for a given environment. The functions $Expr_T$ and $Pred_T$ compute all expressions and predicates that are computable in a given environment.

$$\begin{array}{l} Expr_T : Env \to \mathbb{P}\, Expr \\ Pred_T : Env \to \mathbb{P}\, Pred \\ \hline \forall \mathcal{E} : Env \bullet \\ \quad Expr_T\ \mathcal{E} = \{ e : Expr \mid \exists \tau : \mathsf{Type} \bullet \mathcal{E} \vdash_E e : \tau \} \land \\ \qquad Pred_T\ \mathcal{E} = \{ p : Pred \mid \mathcal{E} \vdash_P p \} \end{array}$$

7.4.9 Expressions

7.4.9.1 Numbers

In order to be supported, numbers get a special treatment. In particular, a possibility to specify number ranges is needed. For this, the Z upto function $(..)$ is used. Otherwise, users would have to define number ranges by set displays (e. g. $\{0, 1, 2, 3, 4, 5, 6, 7\}$ instead of $0..7$), which is somewhat cumbersome for large sets. The upto operator is only supported for constant ranges.

$$\frac{\begin{array}{c} \forall \mathcal{E}{:}Env;\ e_{min}, e_{max}{:}Expr;\ min, max{:}\mathbb{Z} \bullet \\ \mathcal{E} \vdash_E e_{min} : \mathsf{basic}(\mathbb{Z}, \{ Num\ min \}) \land \\ \mathcal{E} \vdash_E e_{max} : \mathsf{basic}(\mathbb{Z}, \{ Num\ max \}) \land min < max \end{array}}{\mathcal{E} \vdash_E e_{min} .. e_{max} : \mathsf{power}(\mathsf{basic}(\mathbb{Z}, \{ i : \mathbb{Z} \mid min \le i \le max \bullet Num\ i \}))} \tag{7.4.1}$$

7.4.9.2 Set Displays

Membership tests of set displays ($e \in \{e_1, e_2\}$) are resolved into a disjunction of equalities ($e = e_1 \lor e = e_2$). Therefore, all elements in a set display have to be enumerable. The types of the elements have to be compatible. The resulting type is the power-type of the unification of the element types. For instance, the type of the expression $\{1, 3\}$ is $\mathsf{power}\ \mathsf{basic}(\mathbb{Z}, \{1, 3\})$, and that of the expression $\{\{1, 3\}, \{2, 3, 4\}\}$ is $\mathsf{power}\ \mathsf{power}\ \mathsf{basic}(\mathbb{Z}, \{1, 2, 3, 4\})$.

$$\frac{\begin{array}{c} \forall \mathcal{E}{:}Env;\ \vec{e}{:}\mathrm{seq}_1\ Expr;\ \vec{\tau}{:}\mathrm{seq}_1\ \mathsf{Type} \mid \mathsf{dom}\ \vec{e} = \mathsf{dom}\ \vec{\tau} \bullet \\ \mathsf{ran}\ \vec{\tau} \subseteq Type_c \land \mathsf{ran}\ \vec{\tau} \in \mathsf{comp} \land (\forall i : \mathsf{dom}\ \vec{\tau} \bullet \mathcal{E} \vdash_E \vec{e}\ i : \vec{\tau}\ i) \end{array}}{\mathcal{E} \vdash_E \{,,\}(\vec{e}) : \mathsf{power}(\mathsf{uni}^{\cup}(\mathsf{ran}\ \vec{\tau}))} \tag{7.4.2}$$

7.4.9.3 Set Comprehension

As mentioned in section 7.4.6 on page 107, a type of a set comprehension is determined by its declaration's characteristic tuple. For the evaluation of the predicate, the environment is augmented according to the declaration's signature. For this, the signature is added to the current environment. The predicate and the predicate induced by the declaration have to be computable.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ d{:}\mathit{Decl};\ p{:}\mathit{Pred} \bullet \quad (\mathit{addDecls}(\mathcal{E})(\varnothing, \mathsf{sig}^{\tau}\ \mathcal{E}\ d)) \vdash_P p \land \mathcal{E} \vdash_P \mathit{declPred}\ d}{\mathcal{E} \vdash_E \{ d \mid p \} : \mathsf{power}(\mathsf{prod}(\mathsf{ct}^{\tau}\ \mathcal{E}\ d))} \tag{7.4.3}$$

The predicate p has to be computable for the local declarations in d already. In fact, this restricts the computability more than necessary. Consider for example the predicate $\{ 1, 3 \} \in \{ X : \mathbb{P}\,\mathbb{N} \mid 3 \in X \}$. With the above definition, this predicate is not computable, since $3 \in X$ for an unbound X is not computable. However, taking an actualization of X with $\{ 1, 3 \}$ into consideration, the predicate is computable. Nevertheless, this problem occurs only with sets of sets and is therefore not regarded as very serious. Because it would make the type system much more complex, it is not supported.

7.4.9.4 Power Set

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ e{:}\mathit{Expr};\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \mathsf{power}\,\tau}{\mathcal{E} \vdash_E \mathbb{P}\, e : \mathsf{power}(\mathsf{power}\,\tau)} \tag{7.4.4}$$

7.4.9.5 Variables

The type of a variable is simply its type as defined by the environment, with given actualization of the generic parameters. The type of abbreviations is the type of their definition with the given actualization.

In the definition, τ denotes the type of the variable and $\vec{\tau}$ the types of the actual parameters. There are two definitions, the first is for variables and the second for abbreviations.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ v{:}\mathit{NAME};\ \mathit{actuals}{:}\mathsf{seq}\ \mathit{Expr};\ \tau{:}\mathsf{Type};\ \vec{\tau}{:}\mathsf{seq}\ \mathsf{Type} \mid \#\vec{\tau} = \#\mathit{actuals} \bullet \quad (v, \vec{\tau}) \in \mathsf{dom}(\mathit{getType}\ \mathcal{E}) \land \mathit{getType}(\mathcal{E})(v, \vec{\tau}) = \tau \land (\forall\, i : \mathsf{dom}\ \vec{\tau} \bullet \mathcal{E} \vdash_E \mathit{actuals}\ i : \vec{\tau}\ i)}{\mathcal{E} \vdash_E \mathit{var}(v, \mathit{actuals}) : \tau} \tag{7.4.5}$$

$$\forall \mathcal{E}:Env;\ v:NAME;\ actuals:\mathrm{seq}\ Expr;\ \tau:\mathsf{Type} \bullet \frac{(v, actuals) \in \mathsf{dom}(getDef\ \mathcal{E}) \land \mathcal{E} \vdash_E getDef\,(\mathcal{E})(v, actuals) : \tau}{\mathcal{E} \vdash_E var(v, actuals) : \tau} \tag{7.4.6}$$

7.4.9.6 Cartesian Product

$$\forall \mathcal{E}:Env;\ \vec{\tau}:\mathrm{seq}_1\ \mathsf{Type};\ \vec{e}:\mathrm{seq}_1\ Expr \mid \mathsf{dom}\ \vec{\tau} = \mathsf{dom}\ \vec{e} \bullet \frac{(\forall i : \mathsf{dom}\ \vec{\tau} \bullet \mathcal{E} \vdash_E \vec{e}\ i : \mathsf{power}(\vec{\tau}\ i) \land \vec{\tau}\ i \in Type_c)}{\mathcal{E} \vdash_E prod\ \vec{e} : \mathsf{power}(\mathsf{prod}\ \vec{\tau})} \tag{7.4.7}$$

7.4.9.7 Tuple

$$\forall \mathcal{E}:Env;\ \vec{e}:\mathrm{seq}_1\ Expr;\ \vec{\tau}:\mathrm{seq}_1\ \mathsf{Type} \mid \mathsf{dom}\ \vec{\tau} = \mathsf{dom}\ \vec{e} \bullet \frac{\forall i : \mathsf{dom}\ \vec{\tau} \bullet \mathcal{E} \vdash_E \vec{e}\ i : (\vec{\tau}\ i)}{\mathcal{E} \vdash_E (,,)(\vec{e}) : \mathsf{prod}\ \vec{\tau}} \tag{7.4.8}$$

7.4.9.8 Tuple Selection

$$\forall \mathcal{E}:Env;\ e:Expr;\ \vec{\tau}:\mathrm{seq}_1\ \mathsf{Type};\ i:\mathbb{Z} \bullet \frac{\mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau} \land i \in \mathsf{dom}\ \vec{\tau}}{\mathcal{E} \vdash_E e\ .\ i : \vec{\tau}\ i} \tag{7.4.9}$$

7.4.9.9 Binding Selection

$$\forall \mathcal{E}:Env;\ e:Expr;\ \tau_{sig}:NAME \pfun \mathsf{Type};\ n:NAME \bullet \frac{\mathcal{E} \vdash_E e : \mathsf{schema}\ \tau_{sig} \land n \in \mathsf{dom}\ \tau_{sig}}{\mathcal{E} \vdash_E e\ .\ n : \tau_{sig}\ n} \tag{7.4.10}$$

7.4.9.10 Lambda Abstraction

Lambda abstraction is the only applicable expression supported. Moreover, lambda abstraction is only supported in function application. It cannot be used to denote a set of tuples. A function is computable, if the expression (and the predicate) resulting from an

application of the function is computable. In order to decide this, the types of the formal parameters are needed. The types of the formal parameters could be taken from their declaration in the lambda expression, as it is done for set comprehension.

However, these types are usually not restrictive enough. Taking these types would limit computability too much. Consider for example the definition of a set union operator for numbers: $\_ \cup \_ == (\lambda\, A, B : \mathbb{P}\,\mathbb{Z} \bullet \{\, x : \mathbb{Z} \mid x \in A \lor x \in B \,\})$. In order to define a useful operator, the value domains of A and B cannot be restricted. With $\mathbb{P}\,\mathbb{Z}$ as type for A and B, however, the predicate $x \in A \lor x \in B$ is not computable. Therefore, the types of the actual parameters in a function application are used. Doing this, a function application of the above defined set union is computable, if the actual parameters allow element test. For example the predicate $v \in \{\, 1, 2 \,\} \cup \{\, x : \mathbb{Z} \mid x > 5 \,\}$ is computable. It will be rewritten to: $v = 1 \lor v = 2 \lor v > 5$.

Because of this, computability of lambda abstraction can only be determined if the lambda abstraction is applied. Since lambda abstraction is only allowed to be used in application, this is always possible.

The type $\mathsf{fun}(\vec{\tau}, \tau)$ of a lambda abstraction means that an expression of type $\mathsf{prod}\,\vec{\tau}$ can be applied and that the resulting expression is of type τ.

In order to define the type of a lambda abstraction, a declaration d_a is assumed. d_a declares the variables of the lambda abstraction's declaration d. The types assigned by d_a are compatible to those defined by d, but are assumed to be the types of the actual parameters rather than the formal one's.

The lambda abstraction's declaration predicate $\mathit{declPred}\ d$, the predicate p, and the expression e have to be computable under the declaration d_a.

$$\frac{\begin{array}{c}\forall\, \mathcal{E}{:}\mathit{Env};\ d, d_a{:}\mathit{Decl};\ p{:}\mathit{Pred};\ f{:}\mathit{Expr};\ \tau{:}\mathsf{Type} \bullet \\ \mathit{addDecls}\ (\mathcal{E})(\varnothing, \mathsf{sig}^{\tau}\ \mathcal{E}\ d_a) \vdash_P \mathit{declPred}\ d \land \\ \mathit{addDecls}(\mathcal{E})(\varnothing, \mathsf{sig}^{\tau}\ \mathcal{E}\ d_a) \vdash_P p \land \\ \mathit{addDecls}(\mathcal{E})(\varnothing, \mathsf{sig}^{\tau}\ \mathcal{E}\ d_a) \vdash_E f : \tau \land \\ \{\, \mathsf{schema}(\mathsf{sig}^{\tau}\ \mathcal{E}\ d_a), \mathsf{schema}(\mathsf{sig}^{\tau}\ \mathcal{E}\ d) \,\} \in \mathsf{comp}\end{array}}{\mathcal{E} \vdash_E \lambda\, d \mid p \bullet f : \mathsf{fun}(\mathsf{ct}^{\tau}\ \mathcal{E}\ d_a, \tau)} \tag{7.4.11}$$

This definition for lambda abstractions gives each lambda abstraction several types, one for each possible application.

7.4.9.11 Function Application

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ f, e{:}\mathit{Expr};\ \vec{\tau}{:}\mathsf{seq}_1\ \mathsf{Type};\ \tau{:}\mathsf{Type} \bullet \\ \mathcal{E} \vdash_E f : \mathsf{fun}(\vec{\tau}, \tau) \land \mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau}}{\mathcal{E} \vdash_E f\ \omega\ e : \tau} \tag{7.4.12}$$

7.4.9.12 Condition

$$\frac{\mathcal{E} \vdash_P p \land \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E e' : \tau' \land \{ \tau, \tau' \} \in \mathsf{comp}}{\mathcal{E} \vdash_E \mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e' : \mathsf{uni}^{\cup} \{ \tau, \tau' \}} \quad (7.4.13)$$

7.4.9.13 Schema Expressions

A schema expression denotes a set of bindings and it is thus the power type of a schema type. The type of a schema display is determined by the signature of its declaration part.

Note the treatment of schema calculus expressions. In the schema calculus, the same variables can be declared multiple times: $[a : \{1, 3\}] \land [a : \mathbb{Z};\ b : B]$. If a variable is declared twice, its value domain has to be computed from the two declarations. If the declarations are combined by $\land$, the value domain is computed by intersection. If the declarations are combined by $\lor$, the value domain is computed by union. This is done by $\mathsf{uni}^{\cap}$ and $\mathsf{uni}^{\cup}$, respectively.

$$\frac{addDecls\ (\mathcal{E})(\varnothing, (\mathsf{sig}^{\tau}\ \mathcal{E}\ d)) \vdash_P p \land \mathcal{E} \vdash_P declPred\ d}{\mathcal{E} \vdash_E [d \mid p] : \mathsf{power}(\mathsf{schema}(\mathsf{sig}^{\tau}\ \mathcal{E}\ d))} \quad (7.4.14)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig}) \land \mathcal{E} \vdash_E e' : \mathsf{power}(\mathsf{schema}\ \tau'_{sig}) \land \\ \mathsf{ran}(\mathsf{implode}\{ \tau_{sig}, \tau'_{sig} \}) \subseteq \mathsf{comp}\end{array}}{\mathcal{E} \vdash_E bin_E(and, e, e') : \mathsf{power}(\mathsf{schema}(\mathsf{uni}^{\cap} \circ (\mathsf{implode}\{ \tau_{sig}, \tau'_{sig} \})))} \quad (7.4.15)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig}) \land \mathcal{E} \vdash_E e' : \mathsf{power}(\mathsf{schema}\ \tau'_{sig}) \land \\ \mathsf{ran}(\mathsf{implode}\{ \tau_{sig}, \tau'_{sig} \}) \subseteq \mathsf{comp}\end{array}}{\mathcal{E} \vdash_E bin_E(or, e, e') : \mathsf{power}(\mathsf{schema}(\mathsf{uni}^{\cup} \circ (\mathsf{implode}\{ \tau_{sig}, \tau'_{sig} \})))} \quad (7.4.16)$$

At this point, it becomes apparent why the logical operators $\neg$, $\Rightarrow$, and $\Leftrightarrow$ are problematic in schema calculus. Negation of a schema is interpreted not only as negation of the predicate, but also as the negation of the predicate induced by the declaration. The problem is that this predicate is used to get the value domain of a variable. Consider the following example:

$$
\begin{aligned}
& \neg\,[\,a : \{\,1, 3\,\} \mid a > 2\,] \\
= & \neg\,[\,a : \mathbb{Z} \mid a \in \{\,1, 3\,\} \land a > 2\,] \\
= & \quad [\,a : \mathbb{Z} \mid \neg\,(a \in \{\,1, 3\,\} \land a > 2)\,] \\
= & \quad [\,a : \mathbb{Z} \mid a \notin \{\,1, 3\,\} \lor a \leq 2\,]
\end{aligned}
$$

Note that with the negation of the schema, any restriction of the value domain gets lost. The variable a can still have the value 2: $\neg\,[\,a : \{\,1, 3\,\} \mid a > 2] \neq [\,a : \mathbb{Z} \setminus \{\,1, 3\,\} \mid \neg\,(a > 2)\,]$.

Thus, any restriction of the value domain gets lost with schema negation. In praxis, however, this is not so bad, since schema negation usually gets along with non-negated schemata. This can be seen in the definition of the schema *xOdd*:

D	$xEven$	$xOdd$
$x : 0 \,..\, 255$	D $\exists\, i : \mathbb{Z} \bullet x = 2 * i$	$D \land \neg\, xEven$

Even though the restriction of the value domain gets lost in the negation of the schema *xEven*, it is restored when conjuncted with D. This is because the conjunction takes the intersection of the value domains. With the presented definition, this is fully exploited.

$$
\frac{\forall\, \mathcal{E} : Env;\ e : Expr;\ \tau_{sig} : NAME \pfun \mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig})}{\mathcal{E} \vdash_E not_E(e) : \mathsf{power}(\mathsf{schema}((\mathsf{revert}\ \mathcal{E}) \circ \tau_{sig}))} \tag{7.4.17}
$$

The same problem as for negation occurs with implication and equivalence, since they imply negation. Implication and equivalence can be handled in transforming them: $A \Rightarrow B \equiv \neg\, A \lor B$ and $A \Leftrightarrow B \equiv (A \land B) \lor (\neg\, A \land \neg\, B)$.

7.4.9.14 Schema Binding

A valid binding b ($b \in \mathsf{dom}\ b_{res}$) defines an assignment from variables to expressions ($as = b_{res}\ \ b$). The type of a binding is a schema type. It assigns each variable of the binding the type of the expression assigned to the variable.

$$
\frac{\forall\, \mathcal{E} : Env;\ b : Expr;\ as : NAME \pfun Expr;\ \tau_{sig} : NAME \pfun \mathsf{Type} \bullet \quad b \in \mathsf{dom}\ b_{res} \land as = (b_{res}\ \ b) \land \mathsf{dom}\ as = \mathsf{dom}\ \tau_{sig} \land (\forall\, v : \mathsf{dom}\ as \bullet \mathcal{E} \vdash_E as\ v : \tau_{sig}\ v)}{\mathcal{E} \vdash_E b : \mathsf{schema}\ \tau_{sig}} \tag{7.4.18}
$$

7.4.10 Predicates

7.4.10.1 Equality

Equality of two expressions is only supported if both expressions are enumerable. The types of the expressions have to be compatible.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ e,e'{:}\mathit{Expr};\ \tau,\tau'{:}\mathsf{Type} \bullet \quad \tau \in \mathit{Type}_c \land \tau' \in \mathit{Type}_c \land \{\tau,\tau'\} \in \mathsf{comp} \land \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E e' : \tau'}{\mathcal{E} \vdash_P e = e'} \tag{7.4.19}$$

7.4.10.2 Set Membership

Set membership is computable if both expressions are computable and type compatible.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ e,E{:}\mathit{Expr};\ \tau,\tau'{:}\mathsf{Type} \bullet \quad \tau \in \mathit{Type}_c \land \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E E : \mathsf{power}\,\tau' \land \{\tau,\tau'\} \in \mathsf{comp}}{\mathcal{E} \vdash_P e \in E} \tag{7.4.20}$$

7.4.10.3 Logical Operators

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ p{:}\mathit{Pred} \bullet \quad \mathcal{E} \vdash_P p}{\mathcal{E} \vdash_P \neg\, p} \tag{7.4.21}$$

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ bt{:}\mathit{BinType};\ p,p'{:}\mathit{Pred} \bullet \quad \mathcal{E} \vdash_P p \land \mathcal{E} \vdash_P p'}{\mathcal{E} \vdash_P \mathit{bin}(bt,p,p')} \tag{7.4.22}$$

7.4.10.4 Quantors

Quantors are resolved in enumerating all possible values of the quantified variables, i. e. $\forall\, x{:}\{e_1,e_2\} \bullet p$, is resolved to $p[e_1/x] \land p[e_2/x]$. Therefore, all type expressions have to be enumerable and p and q have to be computable under an environment, where the quantified variables are declared according to d.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ qt{:}\mathit{QuantType};\ d{:}\mathit{Decl};\ p,q{:}\mathit{Pred} \bullet \quad \mathsf{schema}(\mathsf{sig}^\tau\ \mathcal{E}\ d) \in \mathit{Type}_c \land \mathit{addDecls}(\mathcal{E})(\varnothing, \mathsf{sig}^\tau\ \mathcal{E}\ d) \vdash_P p \Rightarrow q}{\mathcal{E} \vdash_P \mathcal{Q}_{qt}\ d \mid p \bullet q} \tag{7.4.23}$$

7.4.10.5 Facts

$$\frac{\forall \mathcal{E} : Env \bullet \quad true}{\mathcal{E} \vdash_P true} \tag{7.4.24}$$

$$\frac{\forall \mathcal{E} : Env \bullet \quad true}{\mathcal{E} \vdash_P false} \tag{7.4.25}$$

7.4.11 Declarations

function $400(\llbracket \_ \rrbracket^{\mathcal{D}}_{\_})$

The interpretation function $\llbracket \_ \rrbracket^{\mathcal{D}}_{\_}$ translates a declaration in an environment, containing nothing but the given declarations. The set of branches of a free type is computed by *branches*: $branches(b_1 \mid \ldots \mid b_n) = \{ b_1, \ldots, b_n \}$.

The supported declarations are computed by the function $Decl_T$. These are all given and free types and all variable declarations, where the type expression is enumerable.

$$\begin{array}{|l} Decl_T : Env \to \mathbb{P}\, Decl \\ \hline \forall \mathcal{E} : Env \bullet \exists\, decl == Decl_T\ \mathcal{E} \bullet \\ \quad decl \overset{Decl}{\leftarrow} \{ e : Expr;\ n : NAME;\ \tau : Type_c \mid \mathcal{E} \vdash_E e : \mathsf{power}\, \tau \bullet n : e \} \\ \qquad \| \{ n : NAME;\ b : Branch \bullet n ::= b \} \\ \qquad \| \{ n : NAME \bullet Given n \} \\ \qquad \| \{ d, d' : decl \bullet d;\ d' \} \end{array}$$

$$\begin{array}{|l} \llbracket \_ \rrbracket^{\mathcal{D}}_{\_} : Env \times Decl \pfun Env \\ branches : Branch \to \mathbb{P}\, NAME \\ \hline \mathsf{dom}(\llbracket \_ \rrbracket^{\mathcal{D}}_{\_}) = \{ \mathcal{E} : Env;\ d : Decl \mid d \in Decl_T\ \mathcal{E} \} \\ \forall v : NAME \bullet branches(Const\ v) = \{ v \} \\ \forall b_1, b_2 : Branch \bullet branches(b_1 \mid b_2) = branches\ b_1 \cup branches\ b_2 \end{array}$$

7.4.11.1 Variable Declaration

A simple variable declaration creates an environment containing only the declared variable. The type of the variable has to be enumerable. Generic variables are not supported. Therefore, there must not be any current formal generic parameter.

According to the Z type rules, the type expression e has to be a power type.

$$\frac{\forall \mathcal{E} : Env;\ v : NAME;\ e : Expr;\ \tau : \mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \mathsf{power}\, \tau \land \tau \in Type_c}{\llbracket v : e \rrbracket^{\mathcal{D}}_{\mathcal{E}} = (addDecls\ \varnothing_{Env})(\varnothing, \{ v \mapsto \tau \})} \tag{7.4.26}$$

Schema types (bindings) are only allowed at top-level and therefore need a special treatment.

$$\forall \mathcal{E}{:}Env;\ v{:}NAME;\ e{:}Expr;\ \tau_{sig}{:}NAME \pfun \mathsf{Type} \bullet \frac{\mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig}) \land \mathsf{ran}\ \tau_{sig} \subseteq Type_c}{[\![v{:}\ e]\!]^{\mathcal{D}}_{\mathcal{E}} = (addDecls\ \varnothing_{Env})(\varnothing, \{ v \mapsto \mathsf{schema}\ \tau_{sig} \})} \tag{7.4.27}$$

7.4.11.2 Abbreviation

An abbreviation declaration creates a new environment, containing this very abbreviation.

The declaration is added with the current formal generic parameters that are stored in the environment: $getParams\ \mathcal{E}$. In a generic axiomatic definition or a generic schema, the current generic parameters are set to the generic parameters of the respective box. Otherwise, the list of generic parameters is empty.

$$\forall \mathcal{E}{:}Env;\ v{:}NAME;\ e{:}Expr;\ \tau{:}\mathsf{Type} \bullet \frac{\mathcal{E} \vdash_E e : \tau}{[\![v == e]\!]^{\mathcal{D}}_{\mathcal{E}} = addDefs(\varnothing_{Env})(getParams\ \mathcal{E}, \{ v \mapsto e \})} \tag{7.4.28}$$

7.4.11.3 Continuation

$$\forall \mathcal{E}{:}Env;\ d, d'{:}Decl \bullet \frac{true}{[\![d;\ d']\!]^{\mathcal{D}}_{\mathcal{E}} = joinEnv([\![d]\!]^{\mathcal{D}}_{\mathcal{E}}, [\![d']\!]^{\mathcal{D}}_{\mathcal{E}})} \tag{7.4.29}$$

7.4.11.4 Free Type

A free type declaration declares the type and its values as constants in the environment. The set of values of the given type is restricted to the values, ensuring that it is enumerable.

$$\forall \mathcal{E}{:}Env;\ v{:}NAME;\ b{:}Branch \bullet \frac{true}{\begin{aligned}[\![v ::= b]\!]^{\mathcal{D}}_{\mathcal{E}} = addDecls(\varnothing_{Env})(&getParams\ \mathcal{E},\\ &\{ v \mapsto \mathsf{power}(\mathsf{basic}(v, branches\ b)) \} \cup\\ &(\lambda c : branches\ b \bullet \mathsf{basic}(v, \{ c \})))\end{aligned}} \tag{7.4.30}$$

7.4.11.5 Given Type

A given type declaration adds the type to the environment. The set of values is set to $NAME$. This set is actually too large, since it includes also identifiers of other types. Nevertheless, the actual set of values of a given type is not known. Therefore, the maximal set is taken.

$$\forall \mathcal{E}{:}Env;\ v{:}NAME \bullet \frac{true}{[\![Given\ v]\!]^{\mathcal{D}}_{\mathcal{E}} = addDecls(\varnothing_{Env})(getParams\ \mathcal{E}, \{\, v \mapsto \mathsf{power}(\mathsf{basic}(v, NAME)) \,\})} \tag{7.4.31}$$

7.4.11.6 Schema Expression

$$\forall \mathcal{E}{:}Env;\ e{:}Expr;\ \tau_{sig}{:}NAME \pfun \mathsf{Type} \bullet \frac{\mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\ \tau_{sig})}{[\![ex\ e]\!]^{\mathcal{D}}_{\mathcal{E}} = (addDecls\ \varnothing_{Env})(\varnothing, \tau_{sig})} \tag{7.4.32}$$

7.4.12 Specification

Now, computability for a complete specification can be defined:
relation $(\_ \vdash_S \_)$ **function** $1000([\![\_]\!]^{\mathcal{SP}}_{\_})$

$$\frac{[\![\_]\!]^{\mathcal{SP}}_{\_} : Env \times Spec \to Env \quad \_ \vdash_S \_ : Env \leftrightarrow Spec}{\mathsf{dom}([\![\_]\!]^{\mathcal{SP}}_{\_}) = (\_ \vdash_S \_)}$$

7.4.12.1 Axiomatic Definitions

An axiomatic definition with formal generic parameters G, variable declaration d, and predicate p $[d \mid p]_G$ adds the variables declared in d as constants to the environment.

$$\forall \mathcal{E}, \mathcal{E}'{:}Env;\ d{:}Decl;\ p{:}Pred;\ G{:}\mathsf{seq}\ NAME \bullet \frac{\mathcal{E}' = addConst(\mathcal{E})([\![d]\!]^{\mathcal{D}}_{(addParams\ \mathcal{E}\ G)}) \land \mathcal{E}' \vdash_P p \land \mathcal{E} \vdash_P declPred\ d}{\mathcal{E} \vdash_S [d \mid p]_G \land [\![[d \mid p]_G]\!]^{\mathcal{SP}}_{\mathcal{E}} = \mathcal{E}'} \tag{7.4.33}$$

7.4.12.2 Data Schemata

The predicate of data schemata has to be computable. The declaration of the schema is added to the environment for computability check. Only the names of the data variables

are then added to the environment with $addData$. Variables that are declared are not automitally visible in a class.

$$\frac{\forall \mathcal{E}, \mathcal{E}' : Env;\ n : NAME;\ d : Decl;\ p : Pred \bullet \\ \mathcal{E}' = joinEnv(\mathcal{E}, [\![d]\!]^{\mathcal{D}}_{\mathcal{E}}) \land \mathcal{E}' \vdash_P p \land d \in Decl_T\ \mathcal{E}}{\mathcal{E} \vdash_S Schema(n, Data, d, p) \land \\ [\![Schema(n, Data, d, p)]\!]^{\mathcal{SP}}_{\mathcal{E}} = addData\ \mathcal{E}\ \mathcal{E}'} \tag{7.4.34}$$

7.4.12.3 Port Schemata

Port schemata are treated as data schemata with the only exception that variables are added as port variables to the environment.

$$\frac{\forall \mathcal{E}, \mathcal{E}' : Env;\ n : NAME;\ d : Decl;\ p : Pred \bullet \\ \mathcal{E}' = joinEnv(\mathcal{E}, [\![d]\!]^{\mathcal{D}}_{\mathcal{E}}) \land \mathcal{E}' \vdash_P p \land d \in Decl_T\ \mathcal{E}}{\mathcal{E} \vdash_S Schema(n, Port, d, p) \land \\ [\![Schema(n, Port, d, p)]\!]^{\mathcal{SP}}_{\mathcal{E}} = addPort\ \mathcal{E}\ \mathcal{E}'} \tag{7.4.35}$$

7.4.12.4 Init Schemata

The predicates of init schemata have to be checked for computability. The environment is not augmented.

$$\frac{\forall \mathcal{E}, \mathcal{E}' : Env;\ n : NAME;\ d : Decl;\ p : Pred \bullet \\ \mathcal{E}' = joinEnv(\mathcal{E}, [\![d]\!]^{\mathcal{D}}_{\mathcal{E}}) \land \mathcal{E}' \vdash_P p \land \mathcal{E} \vdash_P declPred\ d}{\mathcal{E} \vdash_S Schema(n, Init, d, p) \land \\ [\![Schema(n, Init, d, p)]\!]^{\mathcal{SP}}_{\mathcal{E}} = \mathcal{E}} \tag{7.4.36}$$

7.4.12.5 Property Schemata

In property schemata, only CTL formulae are allowed. The non-modal predicates in the CTL formulae have to be computable. The environment is not altered.

$$\frac{\forall \mathcal{E} : Env;\ T : \mathrm{CTL}[Pred];\ n : NAME;\ d : Decl \bullet \\ T \in \mathrm{CTL}[\{ p : Pred \mid \mathcal{E} \vdash_P p \}] \land T \in \mathsf{ran}\ \mathsf{ctl}^{meta} \land \mathcal{E} \vdash_P declPred\ d}{\mathcal{E} \vdash_S Schema(n, Property, d, (\mathsf{ctl}^{meta\sim})\ T) \land \\ [\![Schema(n, Property, d, (\mathsf{ctl}^{meta\sim})\ T)]\!]^{\mathcal{SP}}_{\mathcal{E}} = \mathcal{E}} \tag{7.4.37}$$

7.4.12.6 Transition Relation Schema

The predicate of the transition relation schema has to be computable. The environment is not changed.

$$\frac{\forall \mathcal{E}, \mathcal{E}' : Env;\ n : NAME;\ d : Decl;\ p : Pred \bullet \\ \mathcal{E}' = joinEnv(\mathcal{E}, [\![d]\!]^{\mathcal{D}}_{\mathcal{E}}) \land \mathcal{E}' \vdash_P p \land \mathcal{E} \vdash_P declPred\ d}{\mathcal{E} \vdash_S Schema(n, Transition, d, p) \land \\ [\![Schema(n, Transition, d, p)]\!]^{\mathcal{SP}}_{\mathcal{E}} = \mathcal{E}} \tag{7.4.38}$$

7.4.12.7 Statechart

The state transition relation computed from the Statechart has to be computable as a transition schema.

$$\frac{\forall \mathcal{E} : Env;\ n : NAME;\ s : State \bullet \\ \mathcal{E} \vdash_S Schema(n, Transition, \Delta \mathrm{DATA}\, \mathcal{E}, stateTrans\ \mathcal{E}\ s)}{\mathcal{E} \vdash_S Statechart\ s \land [\![Statechart\ s]\!]^{\mathcal{SP}}_{\mathcal{E}} = addStatechart\ \mathcal{E}\ s} \tag{7.4.39}$$

7.4.12.8 Item Concatination

Specification items (axiomatic definitions, schemata, etc.) are subsequently added to the environment.

$$\frac{\forall \mathcal{E}, \mathcal{E}' : Env;\ S_1, S_2 : Spec \bullet \\ \mathcal{E} \vdash_S S_1 \land \mathcal{E}' = [\![S_1]\!]^{\mathcal{SP}}_{\mathcal{E}} \land \mathcal{E}' \vdash_S S_2}{\mathcal{E} \vdash_S S_1;\ S_2 \land [\![S_1;\ S_2]\!]^{\mathcal{SP}}_{\mathcal{E}} = [\![S_2]\!]^{\mathcal{SP}}_{\mathcal{E}'}} \tag{7.4.40}$$

7.5 Values

section *Values* **parents** *Type*

Enumerable types can be assigend a set of simple expressions. Simple expressions are variables as well as power set and tuples constructed from simple expressions.

$$\frac{Expr_c : \mathbb{P}\, Expr}{Expr_c \overset{Expr}{\leftarrow} var_{\varnothing}(\!| NAME |\!) \\ \parallel \mathbb{P}(\!| Expr_c |\!) \\ \parallel (,)(\!| \mathrm{seq}_1\ Expr_c |\!)}$$

$$
\begin{array}{l}
values : Type_c \to \mathbb{F}\, Expr_c \\
\hline
\forall\, v : NAME;\ V : \mathbb{F}\, NAME \bullet values(\mathsf{basic}(v, V)) = var(\!|\, V \times \varnothing \,|\!) \\
\forall\, \tau : Type_c \bullet values(\mathsf{power}\, \tau) = \mathbb{P}(\!|\, values\, \tau \,|\!) \\
\forall\, \vec{\tau} : \mathrm{seq}_1\, Type_c \bullet values(\mathsf{prod}\, \vec{\tau}) = (,,)(\!|\mathsf{explode}(values \circ \vec{\tau})|\!) \\
\forall\, \tau_{sig} : (NAME \nrightarrow Type_c) \bullet values(\mathsf{schema}\, \tau_{sig}) = b^{Inv}_{res}\ (\!|\mathsf{explode}(values \circ \tau_{sig})|\!)
\end{array}
$$

In order to support variables of power or product type, such values have to be encoded as variables. This is done by the *code* function. The *code* function uses the *special* ($special \subseteq NAME$) name space.

$$
\begin{array}{l}
code : Expr_c \rightarrowtail special \\
decode : special \nrightarrow Expr_c \\
\hline
code^{\sim} = decode
\end{array}
$$

Consider, for example, a declaration $v : \mathbb{P}\{ a, b, c \}$. The annotated type of v is $\mathsf{power}(\mathsf{basic}(F, \{ a, b, c \}))$ (if a, b, and c are constants of the free type F). Clearly, the type is enumerable. The values v can take are:

$$
\begin{array}{l}
values(\mathsf{power}(\mathsf{basic}(F, \{ a, b, c \}))) = \\
\quad \{ \varnothing, \{ a \}, \{ b \}, \{ c \}, \{ a, b \}, \{ a, c \}, \{ b, c \}, \{ a, b, c \} \}
\end{array}
$$

In order to rewrite v to a flat type, these values are now encoded by constants. This could be for example:

$$
\begin{array}{l}
(code \circ values)(\mathsf{power}(\mathsf{basic}(F, \{ a, b, c \}))) = \\
\quad \{ set\_empty, set\_a, set\_b, set\_c, set\_a\_b, set\_a\_c, set\_b\_c, set\_a\_b\_c \}
\end{array}
$$

7.6 Rewriting to Simple Z

This section describes how Z is translated to Simple Z. The domain of the translation are Z expressions and predicates that are computable as defined by the type system presented in section 7.4 on page 103. A predicate $p \in Pred$ can be translated (in an environment $\mathcal{E}$) if and only if $\mathcal{E} \vdash_P p$. The translation concepts are explained in section 7.2 on page 98.

section *Rewrite* **parents** *SimpleZ, Type, Values, CTLTransform*

function $100([\![\_]\!]^{\mathsf{P}}_{\_})$
function $1000([\![\_]\!]^{\mathsf{E}}_{\_})$
function $1000([\![\_]\!]^{\mathsf{S}}_{\_})$
function $1000([\![\_]\!]^{\mathsf{D}}_{\_})$
function $1000([\![\_]\!]^{\mathsf{Spec}}_{\_})$
function $1000([\![\_ \leftarrow \_]\!]_{\_})$
function $1000([\![\_ \equiv \_]\!]_{\_})$
function $1000([\![\_ \,.\, \_]\!]^{\mathsf{Sel}}_{\_})$
function $1000([\![\_ \sim \_]\!]^{\mathsf{App}}_{\_})$
function $1000([\![\_ \sim \_]\!]^{\mathsf{App_s}}_{\_})$
function $1000([\![\_ \,.\, \_]\!]^{\mathsf{Bind}}_{\_})$

7.6.1 Tuple and Binding Selection

Tuple and binding selection are syntactically resolved. The problem of handling tuple and binding selection is that it can appear at every expression that is of tuple or schema type, respectively. Besides tuples and bindings themselves, also variables, conditionals, and applications of product or schema type can be of tuple or schema type. It can also appear at selection, i. e. $t.1.3$ or $b.a.4$. In reduced Z, it is possible to push selection down

to variables and bindings or tuples and resolve them. This is done by $[\![\_ \,.\, \_]\!]^{\mathsf{Sel}}_{\_}$ for tuple selection and $[\![\_ \,.\, \_]\!]^{\mathsf{Bind}}_{\_}$ for binding selection. Note that selection at lambda abstractions is not possible. It only occurs during the resolution.

$$
\begin{array}{l}
[\![\_ \,.\, \_]\!]^{\mathsf{Sel}}_{\_} : \mathit{Env} \times \mathit{Expr} \times \mathbb{N} \pfun \mathit{Expr} \\
\hline
\mathsf{dom}([\![\_ \,.\, \_]\!]^{\mathsf{Sel}}_{\_}) = \{\, \mathcal{E} : \mathit{Env};\ e : \mathit{Expr};\ i : \mathbb{N} \mid \\
\qquad (\exists \vec{\tau} : \mathsf{seq}_1\ \mathsf{Type} \bullet \mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau} \land i \in \mathsf{dom}\ \vec{\tau}) \,\} \\
\forall \mathcal{E} : \mathit{Env};\ v, n : \mathit{NAME};\ e, e', f : \mathit{Expr};\ p : \mathit{Pred};\ i, i' : \mathbb{N};\ \vec{e} : \mathsf{seq}\ \mathit{Expr} \bullet \\
\qquad [\![f\ \omega\ e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = ([\![f \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}})\ \omega\ e \land \\
\qquad [\![e\ .\ n \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = [\![([\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}) \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} \land \\
\qquad [\![\mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e' \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = \mathbf{if}\ p\ \mathbf{then} [\![e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}\ \mathbf{else} [\![e' \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} \land \\
\qquad (\exists \vec{\tau} : \mathsf{seq}_1\ \mathsf{Type} \mid \mathcal{E} \vdash_E \mathit{var}_{\varnothing}\ v : \mathsf{prod}\ \vec{\tau} \bullet \\
\qquad\qquad [\![\mathit{var}_{\varnothing}\ v \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = \mathit{case\_reduce}\{\, \vec{e} : \mathsf{explode}(\mathit{values} \circ \vec{\tau}) \bullet \\
\qquad\qquad\qquad\qquad \mathit{var}_{\varnothing}\ v = \mathit{var}_{\varnothing}(\mathit{code}((,,)\vec{e})) \mapsto \vec{e}\ i \,\}) \land \\
\qquad (\exists\, e' == [\![e \,.\, i']\!]^{\mathsf{Sel}}_{\mathcal{E}} \bullet [\![e\ .\ i' \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = [\![e' \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}) \land \\
\qquad [\![(,,)\vec{e} \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} = \vec{e}\ i
\end{array}
$$

$$
\begin{array}{l}
[\![\_ \,.\, \_]\!]^{\mathsf{Bind}}_{\_} : \mathit{Env} \times \mathit{Expr} \times \mathit{NAME} \pfun \mathit{Expr} \\
\hline
\mathsf{dom}([\![\_ \,.\, \_]\!]^{\mathsf{Bind}}_{\_}) = \\
\qquad \{\, \mathcal{E} : \mathit{Env};\ e : \mathit{Expr};\ n : \mathit{NAME} \mid \\
\qquad\ (\exists \tau_{sig} : \mathit{NAME} \pfun \mathsf{Type} \bullet \mathcal{E} \vdash_E e : \mathsf{schema}\ \tau_{sig} \land n \in \mathsf{dom}\ \tau_{sig}) \,\} \\
\forall \mathcal{E} : \mathit{Env};\ v, n, n' : \mathit{NAME};\ e, e', f : \mathit{Expr};\ p : \mathit{Pred};\ d : \mathit{Decl};\ i : \mathbb{N} \bullet \\
\qquad [\![(\lambda\, d \mid p \bullet e) \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = [\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} \land \\
\qquad [\![f\ \omega\ e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = ([\![f \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}})\ \omega\ e \land \\
\qquad [\![e\ .\ i \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = [\![([\![e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}) \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} \land \\
\qquad [\![\mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e' \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = \mathbf{if}\ p\ \mathbf{then} [\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}\ \mathbf{else} [\![e' \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} \land \\
\qquad [\![\mathit{var}_{\varnothing}\ v \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = \mathit{var}_{\varnothing}(\mathit{conc}(v, n)) \land \\
\qquad (\exists\, e' == [\![e \,.\, n']\!]^{\mathsf{Bind}}_{\mathcal{E}} \bullet [\![e\ .\ n' \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = [\![e' \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}) \land \\
\qquad (e \in \mathsf{ran}([,,]) \Rightarrow [\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} = (b_{res}\ \ e)(n))
\end{array}
$$

7.6.2 Expressions

The function $[\![\_]\!]^{\mathsf{E}}_{\_}$ rewrites an expression under a given environment. $[\![\_]\!]^{\mathsf{S}}_{\_}$ computes the predicate that has to be hold for the expression to be defined. This is also called the *definedness condition*. For example, for a function application $f\ \omega\ e$ it is: $[\![f\ \omega\ e]\!]^{\mathsf{S}}_{\mathcal{E}} = e \in \mathsf{dom}\ f$.

$$
\begin{array}{l}
[\![\_]\!]^{\mathsf{E}}_{\_} : \mathit{Env} \times \mathit{Expr} \pfun \mathit{Expr} \\
[\![\_]\!]^{\mathsf{S}}_{\_} : \mathit{Env} \times \mathit{Expr} \pfun \mathit{Pred} \\
\hline
\mathsf{dom}([\![\_]\!]^{\mathsf{E}}_{\_}) = \mathsf{dom}([\![\_]\!]^{\mathsf{S}}_{\_}) = \{\, t : (\_ \vdash_E \_ : \_) \bullet (t.1, t.2) \,\}
\end{array}
$$

A curried version of the expression rewriting function $[\![\bullet]\!]^{\mathsf{E}}_{\_}$ is defined, to ease the translation of sequences.

function $1000(\llbracket \_ \rrbracket^{\mathsf{E}}_{\_})$

$$\mid \llbracket \_ \rrbracket^{\mathsf{E}}_{\_} == curryFstOf2(\llbracket \_ \rrbracket^{\mathsf{E}}_{\_})$$

Expression rewriting removes function applications and by that also lambda expressions, tuple selection, and abbreviation. It ensures that

- Expressions of *basic type* are either variables, application of built-ins, or conditionals. Note that built-in functions always return values of basic type.
- Expression of *power-set type* are variables, set comprehensions, set displays, power-sets, Cartesian products, or conditionals.
- Expressions of *product type* are variables, tuples, or conditionals.
- Expressions of *function type* are lambda abstractions. Note that the type system ensures that lambda abstractions appear only in applications. Therefore, an expression like $(x, y) \in (\lambda\, i : \mathbb{N} \bullet i + i)$ is not admissible.

7.6.2.1 Set Comprehension

The sub-terms of set comprehensions are simplified when simplifying membership test predicates. A set comprehension is defined if and only if the type expressions of its declarations are defined.

$$\frac{\forall\, \mathcal{E}{:}Env;\ d{:}Decl;\ p{:}Pred;\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E \{d \mid p\} : \tau}{\begin{array}{l}\llbracket \{d \mid p\} \rrbracket^{\mathsf{E}}_{\mathcal{E}} = \{d \mid p\} \land \\ \llbracket \{d \mid p\} \rrbracket^{\mathsf{S}}_{\mathcal{E}} = \bigwedge \{\, e : \mathsf{ran}(second \circ (ct\ d)) \bullet \llbracket e \rrbracket^{\mathsf{S}}_{\mathcal{E}} \,\}\end{array}} \tag{7.6.1}$$

7.6.2.2 Set Display

Rewriting of set displays descends to the set members. The set display is defined, if all its members are defined. Note that the set $\{\, e_1, \ldots, e_n \,\}$ is denoted by $\{,,\}\vec{e}$ where $\vec{e} = \langle\, e_1, \ldots, e_n \,\rangle$.

$$\frac{\forall\, \mathcal{E}{:}Env;\ \vec{e}{:}\mathsf{seq}_1\ Expr;\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E \{,,\}\vec{e} : \tau}{\begin{array}{l}\llbracket \{,,\}\vec{e} \rrbracket^{\mathsf{E}}_{\mathcal{E}} = \{,,\}(\llbracket \_ \rrbracket^{\mathsf{E}}_{\mathcal{E}} \circ \vec{e}) \land \\ \llbracket \{,,\}\vec{e} \rrbracket^{\mathsf{S}}_{\mathcal{E}} = \bigwedge \{\, e : \mathsf{ran}\ \vec{e} \bullet \llbracket e \rrbracket^{\mathsf{S}}_{\mathcal{E}} \,\}\end{array}} \tag{7.6.2}$$

7.6.2.3 Power Set

The power set construction itself is not changed. Rewriting is pushed to the included expression. If this expression is defined, then the power set construction is defined as well.

$$\frac{\forall \mathcal{E}:Env;\ e:Expr;\ \tau:\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E \mathbb{P}\, e : \tau}{[\![\mathbb{P}\, e]\!]^{\mathsf{E}}_{\mathcal{E}} = \mathbb{P}[\![e]\!]^{\mathsf{E}}_{\mathcal{E}} \land [\![\mathbb{P}\, e]\!]^{\mathsf{S}}_{\mathcal{E}} = [\![e]\!]^{\mathsf{S}}_{\mathcal{E}}} \tag{7.6.3}$$

7.6.2.4 Abbreviations

Abbreviations are replaced by their definitions. Actually, the defining expression has to be interpreted under the environment, where the abbreviation was defined, to avoid name clashes with locally bound variables. Consider the following example:

$$\frac{X == \{\, x : \mathbb{Z} \mid x > 3 \,\}}{\exists \mathbb{Z} : \mathbb{N} \bullet 5 \in X}$$

If X would be replaced by simple textual replacement, the resulting expression would not be well typed anymore. To avoid this problem, α-conversion has to be applied. However, this is omitted here. It is assumed that name clashes are already removed by some other translation step.

$$\frac{\forall \mathcal{E}:Env;\ v:NAME;\ a:\mathsf{seq}\ Expr;\ e:Expr \bullet \quad (v,a) \in \mathsf{dom}(getDef\ \mathcal{E}) \land e = getDef(\mathcal{E})(v,a)}{\begin{array}{c}[\![var(v,a)]\!]^{\mathsf{E}}_{\mathcal{E}} = [\![e]\!]^{\mathsf{E}}_{\mathcal{E}} \land \\ [\![var(v,a)]\!]^{\mathsf{S}}_{\mathcal{E}} = [\![e]\!]^{\mathsf{S}}_{\mathcal{E}}\end{array}} \tag{7.6.4}$$

7.6.2.5 Variables

Variables are left as they are. They may be changed by the predicate rewriter, however. Variables are always defined.

$$\frac{\forall \mathcal{E}:Env;\ v:NAME \bullet \quad (v, \varnothing) \in \mathsf{dom}(getType\ \mathcal{E})}{[\![var_{\varnothing}\ v]\!]^{\mathsf{E}}_{\mathcal{E}} = var_{\varnothing}\ v \land [\![var_{\varnothing}\ v]\!]^{\mathsf{S}}_{\mathcal{E}} = true} \tag{7.6.5}$$

7.6.2.6 Cartesian Product

Rewriting of Cartesian product descends to the product members. The product is defined if all its members are defined.

$$\frac{\forall \mathcal{E}:Env;\ \vec{e}:\mathrm{seq}_1\ Expr;\ \tau:\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E prod\ \vec{e} : \tau}{\begin{array}{c}[\![prod\ \vec{e}]\!]^{\mathsf{E}}_{\mathcal{E}} = prod([\![\_]\!]^{\mathsf{E}}_{\mathcal{E}} \circ \vec{e}) \land \\ [\![prod\ \vec{e}]\!]^{\mathsf{S}}_{\mathcal{E}} = \bigwedge \{\, e : \mathsf{ran}\ \vec{e} \bullet [\![e]\!]^{\mathsf{S}}_{\mathcal{E}} \,\}\end{array}} \tag{7.6.6}$$

7.6.2.7 Tuple

Rewriting of tuples descends to the tuple members. The tuple is defined, if all its members are defined.

$$\frac{\forall \mathcal{E}:Env;\ \vec{e}:\mathrm{seq}_1\ Expr;\ \tau:\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E (,,)\vec{e} : \tau}{\begin{array}{c}[\![(,,)\vec{e}]\!]^{\mathsf{E}}_{\mathcal{E}} = (,,)([\![\_]\!]^{\mathsf{E}}_{\mathcal{E}} \circ \vec{e}) \land \\ [\![(,,)\vec{e}]\!]^{\mathsf{S}}_{\mathcal{E}} = \bigwedge \{\, e : \mathsf{ran}\ \vec{e} \bullet [\![e]\!]^{\mathsf{S}}_{\mathcal{E}} \,\}\end{array}} \tag{7.6.7}$$

7.6.2.8 Tuple Selection

Tuple selection is handled by the tuple selection rewriter ($[\![\_ \,.\, \_]\!]^{\mathsf{Sel}}_{\_}$), introduced in section 7.6.1 on page 121.

$$\frac{\forall \mathcal{E}:Env;\ e:Expr;\ i:\mathbb{N};\ \vec{\tau}:\mathrm{seq}_1\ \mathsf{Type} \bullet \quad i \in \mathsf{dom}\ \vec{\tau} \land \mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau}}{\begin{array}{c}[\![e\ .\ i]\!]^{\mathsf{E}}_{\mathcal{E}} = [\![[\![e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}]\!]^{\mathsf{E}}_{\mathcal{E}} \land \\ [\![e\ .\ i]\!]^{\mathsf{S}}_{\mathcal{E}} = [\![[\![e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}]\!]^{\mathsf{S}}_{\mathcal{E}}\end{array}} \tag{7.6.8}$$

7.6.2.9 Binding Selection

Binding selection is handled by the binding selection rewriter ($[\![\_ \,.\, \_]\!]^{\mathsf{Bind}}_{\_}$), introduced in section 7.6.1 on page 121.

$$\frac{\forall \mathcal{E}:Env;\ e:Expr;\ n:NAME;\ \tau_{sig}:NAME \pfun \mathsf{Type} \bullet \quad n \in \mathsf{dom}\ \tau_{sig} \land \mathcal{E} \vdash_E e : \mathsf{schema}\ \tau_{sig}}{\begin{array}{c}[\![e\ .\ n]\!]^{\mathsf{E}}_{\mathcal{E}} = [\![[\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}]\!]^{\mathsf{E}}_{\mathcal{E}} \land \\ [\![e\ .\ n]\!]^{\mathsf{S}}_{\mathcal{E}} = [\![[\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}]\!]^{\mathsf{S}}_{\mathcal{E}}\end{array}} \tag{7.6.9}$$

7.6.2.10 Lambda Abstraction

Lambda abstractions are handled with the function applications. The definedness of lambda abstractions is also handled in the applications.

$$\frac{\forall \mathcal{E}{:}Env;\ d{:}Decl;\ p{:}Pred;\ f{:}Expr;\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E \lambda\, d \mid p \bullet f : \tau}{\begin{array}{c}[\![(\lambda\, d \mid p \bullet f)]\!]^{\mathsf{E}}_{\mathcal{E}} = \lambda\, d \mid p \bullet f \wedge \\ [\![(\lambda\, d \mid p \bullet f)]\!]^{\mathsf{S}}_{\mathcal{E}} = true\end{array}} \tag{7.6.10}$$

7.6.2.11 Application

Similar to the functions to remove binding and tuple selection, a special function to remove function applications is defined. The function and its argument are rewritten first, to remove binding and tuple selections.

$$\begin{array}{l}[\![\_ \sim \_]\!]^{\mathsf{App}}_{\_} : Env \times Expr \times Expr \pfun Expr \\ [\![\_ \sim \_]\!]^{\mathsf{App_s}}_{\_} : Env \times Expr \times Expr \pfun Pred\end{array}$$

$$\frac{\forall \mathcal{E}{:}Env;\ e,f{:}Expr;\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E f\ \omega\ e : \tau}{\begin{array}{c}[\![f\ \omega\ e]\!]^{\mathsf{E}}_{\mathcal{E}} = [\![[\![f]\!]^{\mathsf{E}}_{\mathcal{E}} \sim [\![e]\!]^{\mathsf{E}}_{\mathcal{E}}]\!]^{\mathsf{App}}_{\mathcal{E}} \wedge \\ [\![f\ \omega\ e]\!]^{\mathsf{S}}_{\mathcal{E}} = [\![[\![f]\!]^{\mathsf{E}}_{\mathcal{E}} \sim [\![e]\!]^{\mathsf{E}}_{\mathcal{E}}]\!]^{\mathsf{App_s}}_{\mathcal{E}}\end{array}} \tag{7.6.11}$$

Function application to lambda abstraction is resolved by β-reduction. This is done by adding the variables of the declaration as abbreviations to the environment. The actual parameters of the application are taken as values.

$$\frac{\begin{array}{c}\forall \mathcal{E},\overline{\mathcal{E}}{:}Env;\ d{:}Decl;\ p{:}Pred;\ e,f{:}Expr;\ \vec{\tau}{:}\mathrm{seq}_1\ \mathsf{Type};\ \tau{:}\mathsf{Type} \bullet \\ \mathcal{E} \vdash_E e : \mathsf{prod}\,\vec{\tau} \wedge \mathcal{E} \vdash_E \lambda\, d \mid p \bullet f : \mathsf{fun}(\vec{\tau}, \tau) \wedge \\ \overline{\mathcal{E}} = addVarsWithDefs(\mathcal{E})(d, \vec{\tau}, (\lambda\, i : \mathsf{dom}\,\vec{\tau} \bullet [\![e \mathbin{.} i]\!]^{\mathsf{Sel}}_{\mathcal{E}}))\end{array}}{\begin{array}{c}[\![(\lambda\, d \mid p \bullet f) \sim e]\!]^{\mathsf{App}}_{\mathcal{E}} = [\![f]\!]^{\mathsf{E}}_{\overline{\mathcal{E}}} \wedge \\ [\![(\lambda\, d \mid p \bullet f) \sim e]\!]^{\mathsf{App_s}}_{\mathcal{E}} = [\![(declPred\ d) \wedge p]\!]^{\mathsf{P}}_{\overline{\mathcal{E}}}\end{array}} \tag{7.6.12}$$

If the function is a conditional, the application is pushed into the sub-terms. The definedness condition is non-strict.

$$\frac{\forall \mathcal{E}:Env;\ e,f,f':Expr;\ p:Pred;\ \vec{\tau}:\mathrm{seq}_1\ \mathsf{Type};\ \tau:\mathsf{Type} \bullet \\ \mathcal{E} \vdash_E (\mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ f') : \mathsf{fun}(\vec{\tau},\tau) \land \mathcal{E} \vdash_E e : \mathsf{prod}\,\vec{\tau}}{\begin{array}{c}[\![\mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ f' \sim e]\!]^{\mathsf{App}}_{\mathcal{E}} = [\![\mathbf{if}\ p\ \mathbf{then}\ f\ \omega\ e\ \mathbf{else}\ f'\ \omega\ e]\!]^{\mathsf{E}}_{\mathcal{E}} \land \\ [\![\mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ f' \sim e]\!]^{\mathsf{App_s}}_{\mathcal{E}} = [\![\mathbf{if}\ p\ \mathbf{then}\ f\ \omega\ e\ \mathbf{else}\ f'\ \omega\ e]\!]^{\mathsf{S}}_{\mathcal{E}}\end{array}} \tag{7.6.13}$$

Built-in function applications are not changed.

$$\frac{\forall \mathcal{E}:Env;\ e:Expr;\ v:NAME;\ a:\mathrm{seq}\ Expr;\ \vec{\tau},\vec{\tau}_a:\mathrm{seq}_1\ \mathsf{Type};\ \tau:\mathsf{Type} \mid \mathsf{dom}\ a = \mathsf{dom}\ \vec{\tau}_a \bullet \\ v \in getBuiltins\,\mathcal{E} \land getType(\mathcal{E})(v,\vec{\tau}_a) = \mathsf{fun}(\vec{\tau},\tau) \land \\ (\forall i : \mathsf{dom}\ a \bullet \mathcal{E} \vdash_E a\ i : \vec{\tau}_a\ i) \land \mathcal{E} \vdash_E e : \mathsf{prod}\,\vec{\tau}}{\begin{array}{c}[\![(var(v,a)) \sim e]\!]^{\mathsf{App}}_{\mathcal{E}} = (var(v,a))\ \omega\ e \land \\ [\![(var(v,a)) \sim e]\!]^{\mathsf{App_s}}_{\mathcal{E}} = [\![e]\!]^{\mathsf{S}}_{\mathcal{E}}\end{array}} \tag{7.6.14}$$

7.6.2.12 Conditional

Conditionals are not rewritten. Rewriting is distributed through these expressions.

$$\frac{\forall \mathcal{E}:Env;\ p:Pred;\ e,e':Expr;\ \tau:\mathsf{Type} \bullet \\ \mathcal{E} \vdash_E \mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e' : \tau}{\begin{array}{c}[\![\mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e']\!]^{\mathsf{E}}_{\mathcal{E}} = \mathbf{if}\,[\![p]\!]^{\mathsf{P}}_{\mathcal{E}}\ \mathbf{then}\,[\![e]\!]^{\mathsf{E}}_{\mathcal{E}}\ \mathbf{else}\,[\![e']\!]^{\mathsf{E}}_{\mathcal{E}} \land \\ [\![\mathbf{if}\ p\ \mathbf{then}\ e\ \mathbf{else}\ e']\!]^{\mathsf{S}}_{\mathcal{E}} = ([\![p]\!]^{\mathsf{P}}_{\mathcal{E}} \Rightarrow [\![e]\!]^{\mathsf{S}}_{\mathcal{E}}) \land (\neg\,([\![p]\!]^{\mathsf{P}}_{\mathcal{E}}) \Rightarrow [\![e']\!]^{\mathsf{S}}_{\mathcal{E}})\end{array}} \tag{7.6.15}$$

7.6.3 Predicates

The function $[\![\_]\!]^{\mathsf{P}}_{\_}$ rewrites predicates. All predicates p that are computable with respect to the type system ($\mathcal{E} \vdash_P p$), are supported. Most rewriting takes place for equality and membership predicates. For these predicates, special rewriting functions are introduced: $[\![\_ \equiv \_]\!]_{\_}$ and $[\![\_ \leftarrow \_]\!]_{\_}$, respectively.

$$\frac{[\![\_]\!]^{\mathsf{P}}_{\_} : Env \times Pred \nrightarrow Pred}{\mathsf{dom}([\![\_]\!]^{\mathsf{P}}_{\_}) = (\_ \vdash_P \_)}$$

A curried version of the predicate rewriting function $[\![.]\!]^{\mathsf{P}}_{\_}$ is defined, to make the translation easier.

function $1000([\![.]\!]^{\mathsf{P}}_{\_})$

$$[\![.]\!]^{\mathsf{P}}_{\_} == curryFstOf2([\![\_]\!]^{\mathsf{P}}_{\_})$$

7.6.3.1 Equality

$$[\![\_ \equiv \_]\!]_\_ : Env \times Expr \times Expr \pfun Pred$$

Equality is handled depending on the type of the expressions. The objective of the translation is that equality tests remain only between simple expressions (of basic type). For an equality test to be true, both expressions have to be defined.

$$\frac{\forall \mathcal{E}: Env;\ e, e': Expr;\ \tau: \mathsf{Type} \bullet \quad \tau \in Type_c \land \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E e : \tau}{[\![e = e']\!]^{\mathsf{P}}_{\mathcal{E}} = [\![[\![e]\!]^{\mathsf{E}}_{\mathcal{E}} \equiv [\![e']\!]^{\mathsf{E}}_{\mathcal{E}}]\!]_{\mathcal{E}} \land [\![e]\!]^{\mathsf{S}}_{\mathcal{E}} \land [\![e']\!]^{\mathsf{S}}_{\mathcal{E}}} \tag{7.6.16}$$

Equality of variables can be left as it is, since if the variables are of power or product type, their values are encoded, and they can thus be compared directly.

$$\frac{\forall \mathcal{E}: Env;\ v, v': NAME;\ \tau, \tau': \mathsf{Type} \bullet \quad \mathcal{E} \vdash_E (var_\varnothing\ v) : \tau \land \mathcal{E} \vdash_E (var_\varnothing\ v') : \tau' \land \{ \tau, \tau' \} \in \mathsf{comp}}{[\![(var_\varnothing\ v) \equiv (var_\varnothing\ v')]\!]_{\mathcal{E}} = ((var_\varnothing\ v) = (var_\varnothing\ v'))} \tag{7.6.17}$$

Equality tests between simple expressions need not be rewritten.

$$\frac{\forall \mathcal{E}: Env;\ e, e': Expr;\ F: NAME;\ V, V': \mathbb{P}\, NAME \bullet \quad \mathcal{E} \vdash_E e : \mathsf{basic}(F, V) \land \mathcal{E} \vdash_E e' : \mathsf{basic}(F, V')}{[\![e \equiv e']\!]_{\mathcal{E}} = e = e'} \tag{7.6.18}$$

An equality test for two sets $E = E'$ is resolved in $\forall\, e : values \bullet (e \in E \Leftrightarrow e \in E')$, where *values* denotes all possible members of E and E'.

$$\frac{\forall \mathcal{E}: Env;\ E, E': Expr;\ \tau, \tau': \mathsf{Type} \bullet \quad \mathcal{E} \vdash_E E : \mathsf{power}\, \tau \land \mathcal{E} \vdash_E E' : \mathsf{power}\, \tau' \land \{ \tau, \tau' \} \in \mathsf{comp} \land \{ \tau, \tau' \} \subseteq Type_c}{[\![E \equiv E']\!]_{\mathcal{E}} = \bigwedge \{ e : values\, \tau \cup values\, \tau' \bullet [\![E \leftarrow e]\!]_{\mathcal{E}} \Leftrightarrow [\![E' \leftarrow e]\!]_{\mathcal{E}} \}} \tag{7.6.19}$$

An equality test of two n-tuples $e = e'$ is resolved by comparing the tuple members, i. e. $e.1 = e'.1 \land \ldots\ldots e.n = e'.n$.

$$\forall \mathcal{E}{:}Env;\ e,e'{:}Expr;\ \vec{\tau}{:}\mathrm{seq}_1\ \mathsf{Type} \bullet$$
$$\frac{\mathcal{E} \vdash_E e : \mathsf{prod}\,\vec{\tau} \land \mathcal{E} \vdash_E e' : \mathsf{prod}\,\vec{\tau}}{[\![e \equiv e']\!]_{\mathcal{E}} = \bigwedge \{\, i : \mathsf{dom}\,\vec{\tau} \bullet [\![[\![e \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}} \equiv [\![e' \,.\, i]\!]^{\mathsf{Sel}}_{\mathcal{E}}]\!]_{\mathcal{E}} \,\} \land [\![e]\!]^{\mathsf{S}}_{\mathcal{E}} \land [\![e']\!]^{\mathsf{S}}_{\mathcal{E}}} \tag{7.6.20}$$

An equality test of binding expression is resolved by comparing the members of the binding.

$$\forall \mathcal{E}{:}Env;\ e,e'{:}Expr;\ \tau_{sig}{:}NAME \pfun \mathsf{Type} \bullet$$
$$\frac{\mathcal{E} \vdash_E e : \mathsf{schema}\,\tau_{sig} \land \mathcal{E} \vdash_E e' : \mathsf{schema}\,\tau_{sig}}{[\![e \equiv e']\!]_{\mathcal{E}} = \bigwedge \{\, n : \mathsf{dom}\,\tau_{sig} \bullet [\![[\![e \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}} \equiv [\![e' \,.\, n]\!]^{\mathsf{Bind}}_{\mathcal{E}}]\!]_{\mathcal{E}} \,\} \land [\![e]\!]^{\mathsf{S}}_{\mathcal{E}} \land [\![e']\!]^{\mathsf{S}}_{\mathcal{E}}} \tag{7.6.21}$$

7.6.3.2 Set Membership

$$[\![\_ \leftarrow \_]\!]_{\_} : Env \times Expr \times Expr \pfun Pred$$
$$\mathsf{dom}([\![\_ \leftarrow \_]\!]_{\_}) = \{\, \mathcal{E} : Env;\ E : Expr;\ e : Expr \mid (\exists \tau, \tau' : \mathsf{Type} \bullet \{\tau, \tau'\} \in \mathsf{comp} \land \mathcal{E} \vdash_E E : \mathsf{power}\,\tau \land \mathcal{E} \vdash_E e : \tau')\}$$

$$\forall \mathcal{E}{:}Env;\ e,E{:}Expr;\ \tau{:}\mathsf{Type} \bullet$$
$$\frac{\mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E E : \mathsf{power}\,\tau}{[\![e \in E]\!]^{\mathsf{P}}_{\mathcal{E}} = [\![[\![E]\!]^{\mathsf{E}}_{\mathcal{E}} \leftarrow e]\!]_{\mathcal{E}} \land [\![E]\!]^{\mathsf{S}}_{\mathcal{E}} \land [\![e]\!]^{\mathsf{S}}_{\mathcal{E}}} \tag{7.6.22}$$

A membership test to number ranges $e \in e_1 \,..\, e_2$ is resolved by using built-in relation, i. e. $e \geq e_1 \land e \leq e_2$.

$$\forall \mathcal{E}{:}Env;\ e,e_1,e_2{:}Expr;\ V,V_1,V_2{:}\mathbb{F}\ NAME \bullet$$
$$\frac{\mathcal{E} \vdash_E e : \mathsf{basic}(\mathbb{Z}, V) \land \mathcal{E} \vdash_E e_1 : \mathsf{basic}(\mathbb{Z}, V_1) \land \mathcal{E} \vdash_E e_2 : \mathsf{basic}(\mathbb{Z}, V_2)}{[\![e_1 \,..\, e_2 \leftarrow e]\!]_{\mathcal{E}} = e \geq e_1 \land e \leq e_2} \tag{7.6.23}$$

A membership test to set comprehension ($e \in \{\, d \mid p \,\}$) is resolved, similar to function application, by β-reduction. The formal variables declared in d are bound to the actual values, defined by e. The predicate is evaluated with this valuation of the parameters. This is done by adding the parameters to the environment.

It addition, the predicate, induced by the declaration, has to be evaluated. A predicate $e \in \{\, x : X \mid p \,\}$ has to be rewritten to $e \in X \land p[x/e]$.

$$\forall \mathcal{E}{:}Env;\ e{:}Expr;\ d{:}Decl;\ p{:}Pred;\ \vec{\tau},\vec{\tau}'{:}\mathrm{seq}_1\ \mathsf{Type} \bullet$$
$$\frac{\{\,\mathsf{prod}\ \vec{\tau}, \mathsf{prod}\ \vec{\tau}'\,\} \in \mathsf{comp} \land \mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau}' \land \mathcal{E} \vdash_E \{d \mid p\} : \mathsf{power}(\mathsf{prod}\ \vec{\tau})}{[\![(\{d \mid p\}) \leftarrow e]\!]_{\mathcal{E}} = [\![p]\!]^{\mathsf{P}}_{addVarsWithDefs(\mathcal{E})(d,\vec{\tau},(\lambda\, i{:}\mathsf{dom}\ \vec{\tau} \bullet [\![e.i]\!]^{\mathsf{Sel}}_{\mathcal{E}}))} \land [\![prod(second \circ (ct\ d)) \leftarrow e]\!]_{\mathcal{E}}} \tag{7.6.24}$$

Schema expressions are treated similar to set comprehensions. In order to preserve the declaration predicate, the function *declPred* is used that computes the predicate induced by a declaration.

$$\forall \mathcal{E}{:}Env;\ e{:}Expr;\ d{:}Decl;\ p{:}Pred;\ \tau{:}\mathsf{Type};\ \tau_{sig}{:}NAME \pfun \mathsf{Type} \bullet$$
$$\frac{\{\,\mathsf{schema}\ \tau_{sig}, \tau\,\} \in \mathsf{comp} \land \mathcal{E} \vdash_E e : \mathsf{schema}\ \tau_{sig} \land \mathcal{E} \vdash_E [d \mid p] : \mathsf{power}\ \tau}{[\![([d \mid p]) \leftarrow e]\!]_{\mathcal{E}} = [\![(declPred\ d) \land p]\!]^{\mathsf{P}}_{addDefs(\mathcal{E})(\varnothing,(\lambda\, n{:}\mathsf{dom}\ \tau_{sig} \bullet [\![e.n]\!]^{\mathsf{Bind}}_{\mathcal{E}}))}} \tag{7.6.25}$$

Set displays are rewritten to disjunctions. A predicate $e \in \{\, e_1, \ldots, e_n \,\}$ is rewritten to $e = e_1 \lor \ldots \lor e_n$.

$$\forall \mathcal{E}{:}Env;\ e{:}Expr;\ \vec{e}{:}\mathrm{seq}_1\ Expr;\ \tau{:}\mathsf{Type} \bullet$$
$$\frac{\tau \in Type_c \land \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E \{,,\}(\vec{e}) : \mathsf{power}\ \tau}{[\![\{,,\}(\vec{e}) \leftarrow e]\!]_{\mathcal{E}} = \bigvee \{\, e_i : \mathsf{ran}\ \vec{e} \bullet [\![e = e_i]\!]^{\mathsf{P}}_{\mathcal{E}} \,\}} \tag{7.6.26}$$

Membership tests to power-sets ($e \in \mathbb{P}\, E$) are equivalent to subset relations: $e \subseteq E$. Each possible value of e has to be a member of E: $\forall\, e_i : values \bullet e_i \in e \Rightarrow e_i \in E$.

$$\forall \mathcal{E}{:}Env;\ e,E{:}Expr;\ \tau{:}\mathsf{Type} \bullet$$
$$\frac{\tau \in Type_c \land \mathcal{E} \vdash_E e : \mathsf{power}\ \tau \land \mathcal{E} \vdash_E \mathbb{P}\ E : \mathsf{power}(\mathsf{power}\ \tau)}{[\![\mathbb{P}\ E \leftarrow e]\!]_{\mathcal{E}} = \bigwedge \{\, e_i : values\ \tau \bullet [\![e \leftarrow e_i]\!]_{\mathcal{E}} \Rightarrow [\![E \leftarrow e_i]\!]_{\mathcal{E}} \,\}} \tag{7.6.27}$$

Membership tests to given types ($e \in F$) are always true, since e can only have values from F.

$$\forall \mathcal{E}{:}Env;\ e{:}Expr;\ F{:}NAME;\ V, V'{:}\mathbb{P}\ NAME \bullet$$
$$\frac{\mathcal{E} \vdash_E e : \mathsf{basic}(F, V) \land \mathcal{E} \vdash_E var(F, \varnothing) : \mathsf{basic}(F, V')}{[\![var_{\varnothing}\ F \leftarrow e]\!]_{\mathcal{E}} = true} \tag{7.6.28}$$

In order to handle variables of power-set type, the sets are encoded in values. In order to handle membership tests, the variables are decoded.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ e{:}\mathit{Expr};\ v{:}\mathit{NAME};\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \tau \land \mathcal{E} \vdash_E \mathit{var}_{\varnothing}\ v : \mathsf{power}\ \tau}{\begin{array}{l}[\![\mathit{var}_{\varnothing}\ v \leftarrow e]\!]_{\mathcal{E}} = \bigwedge \{\, E : \mathit{values}(\mathsf{power}\ \tau) \bullet \\ \quad \mathit{var}_{\varnothing}(\mathit{code}\ E) = \mathit{var}_{\varnothing}\ v \Rightarrow [\![E \leftarrow e]\!]_{\mathcal{E}}\, \}\end{array}} \tag{7.6.29}$$

A membership test to tuples ($e \in A \times B$) is resolved by testing the single tuple elements: $e.1 \in A$ and $e.2 \in B$.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ e{:}\mathit{Expr};\ \vec{e}{:}\mathrm{seq}_1\ \mathit{Expr};\ \vec{\tau}{:}\mathrm{seq}_1\ \mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \mathsf{prod}\ \vec{\tau} \land \mathcal{E} \vdash_E \mathit{prod}\ \vec{e} : \mathsf{power}(\mathsf{prod}\ \vec{\tau})}{[\![\mathit{prod}\ \vec{e} \leftarrow e]\!]_{\mathcal{E}} = \bigwedge \{\, i : \mathsf{dom}\ \vec{\tau} \bullet [\![\vec{e}\ i \leftarrow [\![[\![e]\!]^{\mathsf{E}}_{\mathcal{E}} \mathbf{.} i]\!]^{\mathsf{Sel}}_{\mathcal{E}}]\!]_{\mathcal{E}}\, \}} \tag{7.6.30}$$

Membership tests to sets, defined by conditionals ($e \in \mathbf{if}\ p\ \mathbf{then}\ E\ \mathbf{else}\ E'$), are pushed into the conditional: $(p \Rightarrow e \in E) \land (\neg\, p \Rightarrow e \in E')$.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ p{:}\mathit{Pred};\ e,E,E'{:}\mathit{Expr};\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E \mathbf{if}\ p\ \mathbf{then}\ E\ \mathbf{else}\ E' : \mathsf{power}\ \tau \land \mathcal{E} \vdash_E e : \tau}{\begin{array}{l}[\![\mathbf{if}\ p\ \mathbf{then}\ E\ \mathbf{else}\ E' \leftarrow e]\!]_{\mathcal{E}} = ([\![p]\!]^{\mathsf{P}}_{\mathcal{E}} \Rightarrow [\![E \leftarrow e]\!]_{\mathcal{E}}) \land \\ \qquad\qquad ((\neg\, ([\![p]\!]^{\mathsf{P}}_{\mathcal{E}})) \Rightarrow [\![E' \leftarrow e]\!]_{\mathcal{E}})\end{array}} \tag{7.6.31}$$

7.6.3.3 Logical Operators

Logical operators are left as they are. Only the included predicates are rewritten.

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ p{:}\mathit{Pred} \bullet \quad \mathcal{E} \vdash_P \neg\, p}{[\![\neg\, p]\!]^{\mathsf{P}}_{\mathcal{E}} = \neg\, ([\![p]\!]^{\mathsf{P}}_{\mathcal{E}})} \tag{7.6.32}$$

$$\frac{\forall\, \mathcal{E}{:}\mathit{Env};\ bt{:}\mathit{BinType};\ p,q{:}\mathit{Pred} \bullet \quad \mathcal{E} \vdash_P \mathit{bin}(bt,p,q)}{[\![\mathit{bin}(bt,p,q)]\!]^{\mathsf{P}}_{\mathcal{E}} = \mathit{bin}(bt, [\![p]\!]^{\mathsf{P}}_{\mathcal{E}}, [\![q]\!]^{\mathsf{P}}_{\mathcal{E}})} \tag{7.6.33}$$

7.6.3.4 *Quantors*

Quantors are conjunctions or disjunctions over the value domain of the quantified variables. For example $\forall x : \{1, 2, 3\} \bullet x > y$ is rewritten to $1 > y \land 2 > y \land 3 > y$. The definition for existence quantors is omitted.

$$\frac{\begin{array}{l}\forall\, \mathcal{E}{:}Env;\ d{:}Decl;\ p, q{:}Pred;\ \vec{\tau}{:}\mathrm{seq}_1\ \mathsf{Type} \bullet \\ \forall\, i : \mathsf{dom}\ \vec{\tau} \bullet \mathcal{E} \vdash_E second((ct\ d)\ i) : \vec{\tau}\ i \land \mathcal{E} \vdash_P \mathcal{Q}_{forall}\ d \mid p \bullet q\end{array}}{\begin{array}{l}[\![\mathcal{Q}_{forall}\ d \mid p \bullet q]\!]^{\mathsf{P}}_{\mathcal{E}} = \bigwedge \{\, \vec{e} : \mathsf{explode}(values \circ \vec{\tau}) \bullet \\ [\![((,,)\ \vec{e}) \in prod(second \circ (ct\ d)) \land p \Rightarrow q]\!]^{\mathsf{P}}_{addVarsWithDefs(\mathcal{E})(d,\vec{\tau},\vec{e})}\,\}\end{array}} \tag{7.6.34}$$

7.6.3.5 *Computation Tree Logic*

Computation Tree Logic formulae can be left as their are, except for the plain properties they contain. These properties have to be rewritten, too. CTL formulae are defined over Z expressions, where the expressions have to be sets of bindings. Moreover, as defined by the type system, the expressions have to be given by schema displays. Rewriting is done using the CTL mapping function map_{CTL}.

$$\frac{\begin{array}{l}\forall\, \mathcal{E}{:}Env;\ T{:}\mathrm{CTL}[Pred];\ d{:}Decl \bullet \\ T \in \mathrm{CTL}[\{\, p : Pred \mid \mathcal{E} \vdash_P p \,\}] \land T \in \mathsf{ran}\ \mathsf{ctl}^{meta}\end{array}}{[\![(\mathsf{ctl}^{meta\sim})\ T]\!]^{\mathsf{P}}_{\mathcal{E}} = (\mathsf{ctl}^{meta\sim})(\mathsf{map}_{CTL}\ [\![\_]\!]^{\mathsf{P}}_{\mathcal{E}}\ T)} \tag{7.6.35}$$

7.6.4 Declarations

In Simple Z, only set displays and number ranges $(0 \ldots 255)$ over constants are allowed. Other expressions have to be rewritten to these kinds. This is done by using the *values* of the expressions. The *values* function, introduced in section 7.5 on page 120, assigns an enumerable expression all possible values. For constructed types (i. e. product and schema), the values (tuples and bindings, respectively) have to be encoded as constants. For this, section 7.5 on page 120 also introduces the *code* function.

Considering this, a type expression e of type τ can be rewritten to $code(\!|\, values\tau \,|\!)$.

There is a special treatment for bindings such as $b : [x, y : 0 \ldots 255]$. They are translated to a set of variable declaration: $b\_x, b\_y : 0 \ldots 255$. With this, it is easier to reference these variables.

Abbreviations and free type declarations are left as they are. In fact, they are not needed anymore in Simple Z.

$$\frac{[\![\_]\!]^{\mathsf{D}}_{\_} : Env \times Decl \nrightarrow Decl_f}{\mathsf{dom}([\![\_]\!]^{\mathsf{D}}_{\_}) = \mathsf{dom}([\![\_]\!]^{\mathcal{D}}_{\_})}$$

$$\frac{\forall \mathcal{E}{:}\mathit{Env};\ e{:}\mathit{Expr};\ v{:}\mathit{NAME};\ \tau{:}\mathsf{Type} \bullet \quad \tau \in \mathit{Type}_c \land \mathcal{E} \vdash_E e : \mathsf{power}\,\tau}{[\![v{:}\ e]\!]^{\mathsf{D}}_{\mathcal{E}} = v{:}\,\{,,\}(\mathit{var}_{\varnothing} \circ \mathit{order}\ (\mathit{code}(\!|\mathit{values}\,\tau|\!)))} \tag{7.6.36}$$

$$\frac{\forall \mathcal{E}{:}\mathit{Env};\ v{:}\mathit{NAME};\ e{:}\mathit{Expr};\ \tau_{sig}{:}\mathit{NAME} \pfun \mathsf{Type} \bullet \quad \mathsf{ran}\,\tau_{sig} \subseteq \mathit{Type}_c \land \mathcal{E} \vdash_E e : \mathsf{power}(\mathsf{schema}\,\tau_{sig})}{\begin{array}{l}[\![v{:}\ e]\!]^{\mathsf{D}}_{\mathcal{E}} = \mathsf{setreduce}(\_;\ \_) \\ \qquad (\{\, n : \mathsf{dom}\,\tau_{sig} \bullet \mathit{conc}(v, n){:} \\ \qquad\qquad \{,,\}(\mathit{var}_{\varnothing} \circ \mathit{order}\ (\mathit{code}(\!|\mathit{values}(\tau_{sig}\ n)|\!)))\,\})\end{array}} \tag{7.6.37}$$

$$\frac{\forall \mathcal{E}{:}\mathit{Env};\ d,d'{:}\mathit{Decl} \bullet \quad \mathit{true}}{[\![d;\ d']\!]^{\mathsf{D}}_{\mathcal{E}} = [\![d]\!]^{\mathsf{D}}_{\mathcal{E}};\ [\![d']\!]^{\mathsf{D}}_{\mathcal{E}}} \tag{7.6.38}$$

$$\frac{\forall \mathcal{E}{:}\mathit{Env};\ v{:}\mathit{NAME};\ e{:}\mathit{Expr};\ \tau{:}\mathsf{Type} \bullet \quad \mathcal{E} \vdash_E e : \tau}{[\![v == e]\!]^{\mathsf{D}}_{\mathcal{E}} = v == e} \tag{7.6.39}$$

$$\frac{\forall \mathcal{E}{:}\mathit{Env};\ v{:}\mathit{NAME};\ b{:}\mathit{Branch} \bullet \quad \mathit{true}}{[\![v ::= b]\!]^{\mathsf{D}}_{\mathcal{E}} = v ::= b} \tag{7.6.40}$$

7.6.5 Specification

The function $[\![\_]\!]^{\mathsf{Spec}}_{\_}$ rewrites a complete $\mu\mathcal{SZ}$ class. Here, the special treatment of the different components (axiomatic definition, schemata with different roles, Statechart) of a class are applied.

Note that Simple Z still supports the different schema roles. Only *plain* schemata have to be removed. This can be done, since references to these schemata are resolved and they are thus not needed anymore.

$$\frac{[\![\_]\!]^{\mathsf{Spec}}_{\_} : \mathit{Env} \times \mathit{Spec} \pfun \mathit{Spec}}{\mathsf{dom}([\![\_]\!]^{\mathsf{Spec}}_{\_}) = (\_ \vdash_S \_)}$$

7.6.5.1 Axiomatic Definitions

Axiomatic definitions are used for the declaration of abbreviations ($A == e$), given types ($[TYPE]$), free types ($F ::= a \mid b$), and constant variables. Axiomatic definitions may also contain predicates over the variables, declared in axiomatic definitions.

Only the constant variables and predicates are subject to rewriting. Constant variables may by *loose*, i. e. they can have different values. The value of a constant variable is only restricted by the predicate. According to Spivey [48], specifications with loose variables can be interpreted as families of specifications. The family has one member for each value the variable can take.

Properties have to be verified for each member of these families of specification, thus they have to be verified for each possible value. The value of the variable must not change dynamically.

For this, the constant variables are added as data variables to the data space. Their value is set initially, as defined by the predicate. During the transition execution, the value is kept constant. Note that the predicate has to hold during all steps and not only the first one. However, since the predicate uses only constant variables, it is sufficient that it holds in the beginning. It cannot be violated during the transition execution.

An axiomatic definition is translated into three schemata. A data-, init- and a transition-schema. The names of these schemata are chosen at random.

$$| \; persistend : Env \times Decl_f \rightarrow Pred_f$$

$$\frac{\begin{array}{l}\forall \mathcal{E}{:}Env;\ d,d'{:}Decl;\ p,constant{:}Pred \bullet \\ \qquad \mathcal{E} \mapsto d \in \mathsf{dom}(\llbracket \_ \rrbracket^{\mathsf{D}}_{\_}) \land \mathcal{E} \vdash_P p \land d' = \llbracket d \rrbracket^{\mathsf{D}}_{\mathcal{E}} \land \\ \qquad constant = \bigwedge \{\, n : \mathsf{dom}(\mathsf{ran}(ct\ d)) \bullet var_{\varnothing}(prime\ n) = var_{\varnothing}\ n \,\}\end{array}}{\begin{array}{l}\llbracket [d \mid p]_{\varnothing} \rrbracket^{\mathsf{Spec}}_{\mathcal{E}} = Schema(newName\ \mathcal{E}, Data, d', true); \\ \qquad\qquad Schema(newName\ \mathcal{E}, Init, d', \llbracket p \rrbracket^{\mathsf{P}}_{\mathcal{E}}); \\ \qquad\qquad Schema(newName\ \mathcal{E}, Transition, \Delta\mathrm{DATA}\ \mathcal{E}, constant)\end{array}} \tag{7.6.41}$$

7.6.5.2 Schemata

Schemata with roles (*Data*, *Port*, *Init*, *Property* and *Transition*) are left as they are. Plain schemata are removed.

$$\frac{\begin{array}{l}\forall \mathcal{E}{:}Env;\ n{:}NAME;\ type{:}Stype;\ d{:}Decl;\ p{:}Pred \bullet \\ \qquad \mathcal{E} \vdash_P p \land type \neq Plain\end{array}}{\llbracket Schema(n, type, d, p) \rrbracket^{\mathsf{Spec}}_{\mathcal{E}} = Schema(n, type, \llbracket d \rrbracket^{\mathsf{D}}_{\mathcal{E}}, \llbracket p \rrbracket^{\mathsf{P}}_{\mathcal{E}})} \tag{7.6.42}$$

7.6.5.3 Statecharts

Statecharts are translated as described in section 6.3 on page 86. The Statechart is translated into a state transition relation (*stateTrans Env state*) and replaced by this relation. A Statechart implies the introduction of the racing and persistency semantics.

The Statechart of a class implies the introduction of the racing and persistency semantics (see section 6 on page 78) for the class' variables. Thus, for all data-variables, the required lock variables and persistency predicates are generated, if a Statechart is found in a class.

Note that for racing, all transition predicates have to obey to the locks, which means they may only write to a variable if they hold its lock. This does not apply for the

transition predicates generated for constant variables and invariants, as described in the previous sections, since these predicates do not perform *writing* actions but apply invariants. Invariants are not subject to racing, they have to hold independently of any other writing actions. For the constant variables, no lock variables are generated, since they are superfluous.

If, on the other hand, transitions are supposed to perform writing action, they have to be assigned places and writing has to observe the locks. An example for such transitions are static reactions, known from Statemate. Such additional places would have to be considered during the computation of the writers (see section 6.2.3 on page 79), which is not done. Therefore, other writing transitions are not supported in the presented solution. Nevertheless, in general, this is possible.

$$\frac{\forall \mathcal{E}:Env;\ state:State \bullet \quad true}{\begin{array}{l} [\![Statechart(state)]\!]^{\mathsf{Spec}}_{\mathcal{E}} = \\ \quad Schema(newName\,\mathcal{E}, Transition, \Delta\mathrm{DATA}\,\mathcal{E}, stateTrans\ \ \mathcal{E}\ \ state); \\ \quad Schema(newName\,\mathcal{E}, Data, statechartDecls\,\mathcal{E}, true); \\ \quad Schema(newName\,\mathcal{E}, Transition, \Delta\mathrm{DATA}\,\mathcal{E}, persistencyPred\,\mathcal{E}) \end{array}} \tag{7.6.43}$$

Chapter 8

Translating Simple Z to SMV

section *SMV* **parents** *Name, Syntax, ModelChecking, SimpleZ, Environment, Type, CTLTransform*

The objective of *simple Z* is its translation into the input language of a model checker. In this section, the translation into the input language of McMillan's [38] SMV model checker is shown.

8.1 SMV Syntax

8.1.1 Introduction

In order to describe the translation of simple Z to SMV, firstly the SMV syntax is defined as the Z free type SMV. A SMV model specification contains the following items:

- `DEFINE`: Definitions of abbreviations. The translation does not use abbreviations.
- `VAR`: Declarations of data variables.
- `ASSIGN`: SMV offers two ways of defining the initial state and the state transition relation. The first way is to define a set of assignments that define the initial state and the next state for each variable. The second way is to define the initial state and the transition relation directly by predicates. These predicates are defined by `INIT` and `TRANS` items, as described below. The translation uses the second method, because the initial state and the transition relation are already given by predicates.
- `INIT`: Definition of the initial state by a predicate.
- `TRANS`: Definition of the state transition relation by a predicate.
- `FAIRNESS`: Definition of a fairness constraint. The fairness constraint is given by a predicate. If a fairness constraint is given, the applicable traces are limited to those traces where the fairness constraint is indefinitly often true.
- `SPEC`: Definition of a CTL predicate that is to be shown.

8.1.2 Module

$$\mathsf{SMV} ::= \texttt{MODULE}\langle\!\langle NAME \times \mathsf{item} \rangle\!\rangle$$

8.1.3 Items

function $20(\texttt{init}\ \_\ \texttt{(}\ \_\ \texttt{:=}\ \_)$

function $20(\texttt{next}\ \texttt{(}\ \_\ \texttt{)}\ \texttt{:=}\ \_)$

function $20(\{,,\})$

function 21 leftassoc $(\_\ \texttt{..}\ \_)$

function $20(\_\ \texttt{:}\ \_)$

function $20(\_\ \texttt{:}\ \_)$

function $20(\_\ \texttt{;})$

function 20 leftassoc $(\_\ \texttt{;}\ \_)$

function 4 leftassoc $(\_\ \mathsf{SEP}\ \_)$

function $20(\_\ \texttt{;}\ \_)$

$$\begin{array}{lll}
\mathsf{item} & ::= & \texttt{DEFINE}\langle\!\langle \mathsf{defines} \rangle\!\rangle \\
& | & \texttt{VAR}\langle\!\langle \mathsf{decls} \rangle\!\rangle \\
& | & \texttt{ASSIGN}\langle\!\langle \mathsf{assigns} \rangle\!\rangle \\
& | & \texttt{INIT}\langle\!\langle \mathsf{Expr}_{SMV} \rangle\!\rangle \\
& | & \texttt{TRANS}\langle\!\langle \mathsf{Expr}_{SMV} \rangle\!\rangle \\
& | & \texttt{FAIRNESS}\langle\!\langle \mathsf{Expr}_{SMV} \rangle\!\rangle \\
& | & \texttt{SPEC}\langle\!\langle \mathsf{CTL} \rangle\!\rangle \\
& | & (_\ \mathsf{SEP}\ _)\langle\!\langle \mathsf{item} \times \mathsf{item} \rangle\!\rangle \\
\mathsf{define} & ::= & (_\ \texttt{:}\ _)\langle\!\langle NAME \times \mathsf{Expr}_{SMV} \rangle\!\rangle \\
\mathsf{defines} & ::= & (_\ \texttt{;})\langle\!\langle \mathsf{define} \rangle\!\rangle \\
& | & (_\ \texttt{;}\ _)\langle\!\langle \mathsf{define} \times \mathsf{defines} \rangle\!\rangle \\
\mathsf{assign} & ::= & (\texttt{init}\ \texttt{(}\ _\ \texttt{)}\ \texttt{:=}\ _)\langle\!\langle NAME \times \mathsf{Expr}_{SMV} \rangle\!\rangle \\
& | & (\texttt{next}\ \texttt{(}\ _\ \texttt{)}\ \texttt{:=}\ _)\langle\!\langle NAME \times \mathsf{Expr}_{SMV} \rangle\!\rangle \\
\mathsf{assigns} & ::= & (_\ \texttt{;})\langle\!\langle \mathsf{assign} \rangle\!\rangle \\
& | & (_\ \texttt{;}\ _)\langle\!\langle \mathsf{assigns} \times \mathsf{assigns} \rangle\!\rangle \\
\mathsf{decl} & ::= & (_\ \texttt{:}\ _)\langle\!\langle NAME \times \mathsf{type} \rangle\!\rangle \\
\mathsf{decls} & ::= & (_\ \texttt{;})\langle\!\langle \mathsf{decl} \rangle\!\rangle \\
& | & (_\ \texttt{;}\ _)\langle\!\langle \mathsf{decl} \times \mathsf{decls} \rangle\!\rangle \\
\mathsf{type} & ::= & \texttt{boolean} \\
& | & (\{,,\})\langle\!\langle \mathrm{seq}_1\ NAME \rangle\!\rangle \\
& | & (_\ \texttt{..}\ _)\langle\!\langle \mathbb{Z} \times \mathbb{Z} \rangle\!\rangle
\end{array}$$

8.1.4 Expressions

function $30(\{,,\})$ **function** 20 leftassoc $(\_ * \_)$ **function** 20 leftassoc $(\_ / \_)$
function 21 leftassoc $(\_ + \_)$ **function** 21 leftassoc $(\_ - \_)$ **function** 22 leftassoc $(\_ \texttt{mod} \_)$
function 23 leftassoc $(\_ = \_)$ **function** 23 leftassoc $(\_ < \_)$ **function** 23 leftassoc $(\_ <= \_)$
function 23 leftassoc $(\_ > \_)$ **function** 23 leftassoc $(\_ \texttt{in} \_)$ **function** 23 leftassoc $(\_ >= \_)$
function 25 leftassoc $(\_ \& \_)$ **function** 26 leftassoc $(\_ \mid \_)$ **function** 27 leftassoc $(\_ \texttt{->} \_)$
function 27 leftassoc $(\_ \texttt{<->} \_)$ **function** $20(\texttt{next}\ (\_))$ **function** 25 leftassoc $(\_ : \_)$
function 24 rightassoc $(\_ \texttt{;} \_)$ **function** 23 leftassoc $(\_ \texttt{union} \_)$ **function** $14(\texttt{case}\ \_)$

$$
\begin{array}{lll}
\mathsf{Expr}_{SMV} & ::= & \mathsf{num}\langle\langle \mathbb{Z} \rangle\rangle \\
& \mid & \mathsf{var}\langle\langle NAME \rangle\rangle \\
& \mid & \texttt{!}\langle\langle \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{\&} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{|} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{->} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{<->} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{=} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{<} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{<=} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{>} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{>=} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{+} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{-} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{*} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{/} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{mod} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{in} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\_ \texttt{union} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\texttt{next}\ (\_))\langle\langle NAME \rangle\rangle \\
& \mid & (\{,,\})\langle\langle \mathrm{seq}_1\ \mathsf{Expr}_{SMV} \rangle\rangle \\
& \mid & (\texttt{case}\ \_)\langle\langle \mathsf{Cases}_{SMV} \rangle\rangle \\
\mathsf{Cases}_{SMV} & ::= & (\_ \texttt{;} \_)\langle\langle \mathsf{CaseExpr}_{SMV} \times \mathsf{Cases}_{SMV} \rangle\rangle \\
& \mid & \texttt{esac} \\
\mathsf{CaseExpr}_{SMV} & ::= & (\_ \texttt{:} \_)\langle\langle \mathsf{Expr}_{SMV} \times \mathsf{Expr}_{SMV} \rangle\rangle
\end{array}
$$

8.1.5 Computation Tree Logic

SMV CTL predicates are ordinary CTL formulae, with Expr_{SMV} as properties.

$$\mathsf{CTL} == \mathrm{CTL}[\mathsf{Expr}_{SMV}]$$

True and false are defined as one and zero, respectively:

$$1 == \mathsf{num}\,1$$
$$0 == \mathsf{num}\,0$$

8.2 Translation

8.2.1 Predicates

function $200([\![\_]\!]^{\mathcal{P}})$

$$\mid [\![\_]\!]^{\mathcal{P}} : Pred_f \rightarrow \mathsf{Expr}_{SMV}$$

$\forall \bullet$

$$\frac{true}{[\![true]\!]^{\mathcal{P}} = \mathsf{1}} \quad (8.2.1)$$

$\forall \bullet$

$$\frac{true}{[\![false]\!]^{\mathcal{P}} = \mathsf{0}} \quad (8.2.2)$$

$\forall p : Pred_f \bullet$

$$\frac{true}{[\![\neg\ p]\!]^{\mathcal{P}} = \mathsf{!}([\![p]\!]^{\mathcal{P}})} \quad (8.2.3)$$

$\forall p, p' : Pred_f \bullet$

$$\frac{true}{[\![p \wedge p']\!]^{\mathcal{P}} = [\![p]\!]^{\mathcal{P}}\ \mathsf{\&}\ [\![p']\!]^{\mathcal{P}}} \quad (8.2.4)$$

$\forall p, p' : Pred_f \bullet$

$$\frac{true}{[\![p \vee p']\!]^{\mathcal{P}} = [\![p]\!]^{\mathcal{P}}\ \mathsf{|}\ [\![p']\!]^{\mathcal{P}}} \quad (8.2.5)$$

$\forall p, p' : Pred_f \bullet$

$$\frac{true}{[\![p \Rightarrow p']\!]^{\mathcal{P}} = [\![p]\!]^{\mathcal{P}}\ \mathsf{->}\ [\![p']\!]^{\mathcal{P}}} \quad (8.2.6)$$

$\forall p, p' : Pred_f \bullet$

$$\frac{true}{[\![p \Leftrightarrow p']\!]^{\mathcal{P}} = [\![p]\!]^{\mathcal{P}}\ \mathsf{<->}\ [\![p']\!]^{\mathcal{P}}} \quad (8.2.7)$$

$\forall e, e' : Expr_f \bullet$

$$\frac{true}{[\![e = e']\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{E}}\ \mathsf{=}\ [\![e']\!]^{\mathcal{E}}} \quad (8.2.8)$$

$\forall e, e' : Expr_f \bullet$

$$\frac{true}{[\![e < e']\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{E}}\ \mathsf{<}\ [\![e']\!]^{\mathcal{E}}} \quad (8.2.9)$$

$\forall e, e' : Expr_f \bullet$

$$\frac{true}{[\![e \leq e']\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{E}}\ \mathsf{<=}\ [\![e']\!]^{\mathcal{E}}} \quad (8.2.10)$$

$\forall e, e' : Expr_f \bullet$

$$\frac{true}{[\![e > e']\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{E}}\ \mathsf{>}\ [\![e']\!]^{\mathcal{E}}} \quad (8.2.11)$$

$\forall e, e' : Expr_f \bullet$

$$\frac{true}{[\![e \geq e']\!]^{\mathcal{P}} = [\![e]\!]^{\mathcal{E}}\ \mathsf{>=}\ [\![e']\!]^{\mathcal{E}}} \quad (8.2.12)$$

$\forall e, e', E : Expr_f \bullet$

$$\frac{true}{[\![e \neq [\ e'\ ]\ E]\!]^{\mathcal{P}} = \mathsf{!}([\![e]\!]^{\mathcal{E}}\ \mathsf{=}\ [\![e']\!]^{\mathcal{E}})} \quad (8.2.13)$$

8.2.2 Expressions

function $200([\![\_]\!]^{\mathcal{E}})$

$$\mid [\![\_]\!]^{\mathcal{E}} : Expr_f \rightarrow \mathsf{Expr}_{SMV}$$

$\forall v : NAME \bullet$

$$\frac{v \notin \mathsf{dom}\ prime}{[\![var_{\varnothing}\ v]\!]^{\mathcal{E}} = \mathsf{var}\ v} \quad (8.2.14)$$

$\forall\, v{:}NAME \bullet$

$$\frac{v \in \mathrm{dom}\ prime}{(\llbracket (var_{\varnothing}\ v) \rrbracket^{\mathcal{E}} = \texttt{next}\ (((prime^{\sim})v)\,))} \tag{8.2.15}$$

$\forall\, p{:}Pred_f;\ e,e'{:}Expr_f \bullet$

$$\frac{true}{\begin{array}{l}\llbracket \textbf{if}\ p\ \textbf{then}\ e\ \textbf{else}\ e' \rrbracket^{\mathcal{E}} = \texttt{case}\ (\llbracket p \rrbracket^{\mathcal{P}}) : (\llbracket e \rrbracket^{\mathcal{E}})\ ; \\ \qquad\qquad\qquad\qquad 1 : (\llbracket e' \rrbracket^{\mathcal{E}})\ ; \\ \qquad\qquad\quad \texttt{esac}\end{array}} \tag{8.2.16}$$

$\forall\, e,e'{:}Expr_f \bullet$

$$\frac{true}{\llbracket e + e' \rrbracket^{\mathcal{E}} = (\llbracket e \rrbracket^{\mathcal{E}})\ \texttt{+}\ (\llbracket e' \rrbracket^{\mathcal{E}})} \tag{8.2.17}$$

$\forall\, e,e'{:}Expr_f \bullet$

$$\frac{true}{\llbracket e - e' \rrbracket^{\mathcal{E}} = (\llbracket e \rrbracket^{\mathcal{E}})\ \texttt{-}\ (\llbracket e' \rrbracket^{\mathcal{E}})} \tag{8.2.18}$$

8.2.3 Declarations

function $200(\llbracket \_ \rrbracket^{\mathcal{DD}})$
function $200(\llbracket \_ \rrbracket^{\mathcal{D}})$

$$\left|\begin{array}{l}\llbracket \_ \rrbracket^{\mathcal{D}} : Decl_f \nrightarrow \mathsf{decl} \\ \llbracket \_ \rrbracket^{\mathcal{DD}} : Decl_f \nrightarrow \mathsf{decls}\end{array}\right.$$

$\forall\, \mathcal{E}{:}Env;\ n{:}NAME;\ e_{min},e_{max}{:}var_{\varnothing}(\!|Number|\!);\ min,max{:}\mathbb{Z} \bullet$

$$\frac{var_{\varnothing}(Num\ min) = e_{min} \land var_{\varnothing}(Num\ max) = e_{max}}{\llbracket n{:}\ e_{min}\ ..\ e_{max} \rrbracket^{\mathcal{D}} = n : min\ ..\ max} \tag{8.2.19}$$

$\forall\, \mathcal{E}{:}Env;\ n{:}NAME;\ \vec{e}{:}\mathrm{seq}_1\ var_{\varnothing}(\!|NAME|\!) \bullet$

$$\frac{true}{\llbracket n{:}\ \{,,\}\ \vec{e} \rrbracket^{\mathcal{D}} = n : (\{,,\})(var_{\varnothing}{}^{\sim} \circ \vec{e})} \tag{8.2.20}$$

$\forall\, \mathcal{E}{:}Env;\ n{:}NAME;\ e{:}Expr;\ d{:}Decl_f \bullet$

$$\frac{n{:}\ e \in Decl_f}{\llbracket n{:}\ e;\ \ d \rrbracket^{\mathcal{DD}} = (\llbracket n{:}\ e \rrbracket^{\mathcal{D}})\ ;(\llbracket d \rrbracket^{\mathcal{DD}})} \tag{8.2.21}$$

8.2.4 Specifications

function $200(\llbracket \_ \rrbracket^{\mathcal{S}})$

$$\left|\ \llbracket \_ \rrbracket^{\mathcal{S}} : Spec_f \nrightarrow \mathsf{item}\right.$$

The *next* function translates a predicate by translating all variables into their primed version. It translates a predicate into a predicate that holds in the *next*-state: $next(v{=}4) = v' = 4$.

$$\mid next : Pred \to Pred$$

$\forall\, n{:}NAME;\ dataport{:}Stype;\ d{:}Decl_f;\ p{:}Pred_f \bullet$

$$\frac{dataport \in \{\, Data, Port \,\}}{[\![Schema(n, dataport, d, p)]\!]^{\mathcal{S}} = \mathtt{VAR}([\![d]\!]^{\mathcal{DD}})\ \mathsf{SEP}\ \mathtt{TRANS}([\![next(p)]\!]^{\mathcal{P}})} \tag{8.2.22}$$

$\forall\, d{:}Decl_f;\ p{:}Pred_f \bullet$

$$\frac{true}{[\![([d \mid p]_{\varnothing})]\!]^{\mathcal{S}} = \mathtt{VAR}([\![d]\!]^{\mathcal{DD}})\ \mathsf{SEP}\ \mathtt{INIT}([\![p]\!]^{\mathcal{P}})} \tag{8.2.23}$$

$\forall\, n{:}NAME;\ d{:}Decl_f;\ p{:}Pred_f \bullet$

$$\frac{true}{[\![(Schema(n, Init, d, p))]\!]^{\mathcal{S}} = \mathtt{INIT}([\![p]\!]^{\mathcal{P}})} \tag{8.2.24}$$

$\forall\, n{:}NAME;\ d{:}Decl_f;\ p{:}Pred_f \bullet$

$$\frac{true}{[\![Schema(n, Transition, d, p)]\!]^{\mathcal{S}} = \mathtt{TRANS}([\![p]\!]^{\mathcal{P}})} \tag{8.2.25}$$

$\forall\, n{:}NAME;\ d{:}Decl_f;\ p{:}Pred_f \bullet$

$$\frac{true}{[\![Schema(n, Fairness, d, p)]\!]^{\mathcal{S}} = \mathtt{FAIRNESS}([\![p]\!]^{\mathcal{P}})} \tag{8.2.26}$$

$\forall\, \mathcal{E}{:}Env;\ n{:}NAME;\ d{:}Decl_f;\ T{:}\mathrm{CTL}[Expr] \bullet$

$$\frac{T \in \mathrm{CTL}[\{\, p : Pred \mid \mathcal{E} \vdash_P p \,\}] \land T \in \mathsf{ran}\ \mathsf{ctl}^{meta}}{[\![Schema(n, Property, d, (\mathsf{ctl}^{meta\sim})\ T)]\!]^{\mathcal{S}} = \mathtt{SPEC}(\mathsf{map}_{CTL}([\![\_]\!]^{\mathcal{P}})(T))} \tag{8.2.27}$$

Chapter 9

Conclusion

In this work an approach for model checking $\mu\mathcal{SZ}$ specifications is presented. The model checking is done by translating the specification into the input language of the SMV model checker. Other model checkers can be integrated easily, too. The $\mu\mathcal{SZ}$ language is supported to a large extent. Limitations are "model checking intrinsic", such as finiteness of the model.

As a side result, pure Z or pure Statemate specifications can be checked as well. Comparing the performance of Statemate checking with other approaches (see section 1.6 on page 15) shows that the translation scheme is very efficient and that the integration with Z does not cause any performance penalties.

Z offers a much richer expression language than Statemate. This advantage of Z can be fully exploited for model checking. This is true for both: specifying the properties to be verified as well as specifying guards and actions in the Statecharts. This shows that the combination of Statemate/Statecharts and Z is very well suited for the application of verification techniques.

9.1 Limitations

For small and medium sized control specifications, the results that can be achieved with model checking are very good. However, due to the exponential growth of the state space, model checking can not be used for large specifications. Nevertheless, it is possible to verify single modules of large specifications.

Moreover, integer arithmetic (integer and floating point) is not efficiently supported by BDD based symbolic model checking. Other model checking techniques are needed to support this better.

Model checking has been used successfully for hardware specifications (e. g. in processor design). Hardware problems have the advantage that the number of registers etc. is finite and known. They are therefore finite by nature. Usually, this number is also rather small. Thus, hardware problems are a perfect target for model checking.

In software, the number of variables is usually quite high and not bound. Even if it is

possible to restrict the state space to be finite, it is likely too huge to be subject of model checking. Thus, considering today's model checking technology, software problems can not be tackled with model checking, yet. However, model checking is a quickly developing area, and there are a number of promising approaches that try to cope with these problems (see section 4.7 on page 46).

Fortunately, control problems such as the Espress project's reference case studies, the intelligent cruise control and traffic light systems, are not as hard to handle as general software problems. They have a bounded state space and do not contain much integer arithmetic.

9.2 Implementation

With the exception of most of the Z rewriting, the presented translations (Statemate to abstract Statemate syntax, abstract Statemate syntax to a Z state transition relation, and Z to SMV) have been implemented by the author. The implementation was done using the Java dialect *Pizza* ([44]) and the JavaCC compiler-compiler. It runs under Sun's Java Standard Edition Runtime Environments 1.1.8 and 1.2 under Linux, Solaris, and Windows NT. For integration and type checking, Grieskamp's Zeta Z type checker was used. The translators where integrated using the Zeta system ([14]).

The Zeta system has proven to be very effective in:

- managing the chain of translators,
- modularizing the translation,
- offering a uniform (graphical) user interface, and
- integrating third party translators.

9.3 The Meta Theory

The mathematical theory of this work was done using Z as meta language. It was syntax and type checked with the Zeta type checker. Using a formal language like Z is sometimes a little bit long winded. However, the checker reveals flaws in the specification very quickly and thoroughly. Finding these bugs without automatization would call for extensive proof readings. Due to the size of the theory (approx. 350 function and variable declarations), this would be particularly hard. Also reading the theory is easier if one can rely on syntax and type correctness.

Some of the Z deficiencies can be circumvented in using TEX macros together with the Zeta `%%macro` directive. In defining TEX macros for functions, the printed representation can look quite different from what the type checker sees. Zeta macros are replaced by the type checker, before the checking is done.

With this mechanism it is, for example, possible to "overload" symbols. This is not possible in Z. For instance, to overload +, one could define a second operator `\myPlus` and define: `\def\myPlus{+}`. For the reader of the printed representation, the specification is

not type correct, because he cannot see the difference. Therefore, this technique has to be used with care. In this work, such tricks are explained whenever they are used.

9.4 Implementation

As already mentioned, the Z rewriting is not completely implemented. Most of its implementation was done for experimental purposes. In order to implement it, one could consider the Isabelle theory Holz for Z, presented by Kolyang, Santen and Wolff [35]. This has the advantage of guaranteeing the correctness of the rewriting. There is an Isabelle adaptor for Zeta that allows the integration of Isabelle/Holz translations into the Zeta tool chain. With this, it would be easy to integrate Isabelle/Holz rewriting into the translation process. However, the Isabelle adaptor has not been in the state of development that it could be used for this purpose, when this work was conducted.

9.5 Processing the SMV Output

The SMV model checker outputs the results of the verification. This comprises the verified formulae and, for failed formulae, the error backtrace. The backtrace shows a sequence of states which violate the formula. These backtraces are very important for understanding, why a formula does not hold. For the time being, the SMV output is presented as it is, with only marginal post-processing.

It would be desirable to present the backtraces as Statemate simulation runs. With this, the user could reproduce the trace in an environment he is used to. Statemate presents the active states and firing transitions graphically and offers elaborated support for monitoring variables.

Unfortunately, the Statemate interface that could be used to feed the backtrace in Statemate, is too limited. It does not support non-deterministic specification sufficiently. Therefore, this approach was abandoned.

An alternative could be to write a new tool to visualize the backtrace. This tool could use the graphical information found in the Statemate model, to visualize the Statemate Statecharts. It could also be used to present errors found in the Statemate model by a translator or the type checker. Currently, such errors can only be presented textual, e. g. “undeclared variable I in transition from state A to state B”. The Zeta systems would support such an alternative way to present errors.

In order to make the tool user-friendly and to increase the acceptance, post-processing of the SMV output is indispensable.

9.6 Statemate Semantics

The Statemate translation presented here is based on the abstract syntax presented in section 5.1 on page 55. A translation from Statemate textual model representation format

into the abstract syntax is presented by Li [36]. Any graphical information (coordinates of states etc.) found in the Statemate model is ignored.

This approach is not necessarily correct. Moreover, it is not convincing for the reader, since he knows only the graphical representation and not the Statemate file format.

In order to describe the semantics of a graphical language, a visual grammar is needed, and the semantics have to be described based on this grammar. Recent work on visual languages and visual grammars also provide support for Statechart. See for example Costagliola et. al. [19].

Bibliography

[1] Rajeev Alur. Timed automata. In *11th Intl. Conf. on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[2] Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. Mocha: Modularity in model checking. In *Proc. of the Tenth Intl. Conf. on Computer-aided Verification (CAV 1998)*, volume 1427 of *LNCS*, pages 521–525. Springer-Verlag, 1998.

[3] G. Berry, S. Moisan, and J.-P. Rigault. ESTEREL: Towards a synchronous and semantically sound high-level language for real-time applications. In *Proc. IEEE Real-Time Systems Symposium*, volume IEEE catalog 83CH1941-4, pages 30–40, 1983.

[4] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19 no. 2:87–152, 1992.

[5] U. Brockmeyer and G. Wittich. Tamagotchis need not die – verification of statemate designs. In Bernhard Steffen, editor, *Proc. of the 4th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems – TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, 1998.

[6] Udo Brockmeyer and Gunnar Wittich. Real-time verification of Statemate designs. In Alan Hu, editor, *Computer Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[7] Randale E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] R. Büssow, R. Geisler, and M. Klar. Spezifikation eingebetteter Steuerungssysteme mit Z und Statecharts. In *Tagungsband zur 5. Fachtagung: Entwurf komplexer Automatisierungssysteme*. Technische Universität Braunschweig, 1997.

[9] R. Büssow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: A case study. In Egidio Astesiano, editor, *Proc. of the 1st Intl. Conf. on Fundemantal Approaches to Software Engineering – FASE'98*, volume 1382 of *LNCS*, pages 71–87. Springer-Verlag, 1998.

[10] R. Büssow, R. Geisler, M. Klar, and S. Mann. Spezifikation einer Lichtsignalanlagen-Steuerung mit $\mu\mathcal{SZ}$. Technical Report 97–13, Technische Universität Berlin, Fachbereich Informatik, 1997.

[11] Robert Büssow, Robert Geisler, Wolfgang Grieskamp, and Marcus Klar. The $\mu\mathcal{SZ}$ notation version 1.0. Technical Report 97–26, Technische Universität Berlin, Fachbereich Informatik, December 1997.

[12] Robert Büssow, Robert Geisler, Wolfgang Grieskamp, and Marcus Klar. Integrating Z with dynamic modeling techniques for the specification of reactive systems. 1998.

[13] Robert Büssow and Wolfgang Grieskamp. Combining Z and temporal interval logics for the formalization of properties and behaviors of embedded systems. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science – Asian '97*, volume 1345 of *LNCS*, pages 46–56. Springer-Verlag, 1997.

[14] Robert Büssow and Wolfgang Grieskamp. A modular framework for the integration of heterogenous notations and tools. In Keijiro Araki, Andy Galloway, and Kenji Taguchi, editors, *Proc. of the 1st Intl. Conference on Integrated Formal Methods—IFM'99*. Springer-Verlag, London, June 1999.

[15] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Conf. on Principles of Programming Languages*, 1987.

[16] Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[17] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, number 131 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

[18] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge Massachusetts, 1999.

[19] G. Costagliola, F. Ferrucci, G. Polese, and G. Vitiello. Supporting hybrid and hierarchical visual language definition. In *Proc. of the 1999 IEEE Symposium on Visual Languages*, pages 236–245, September 1999.

[20] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[21] Nancy Day. A model checker for statecharts. Technical Report TR-93-35, UBC, October 1993.

[22] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 995–1072. MIT press, 1990.

[23] E. A. Emerson, A. K. Mok, and A. P. Sistla. Quantative temporal reasoning. In E. M. Clarke and R. P. Kurshan, editors, *Workshop on Computer-Aided Verification. 2nd Intl. Conf. CAV'90, Proceedings*, volume 531 of *LNCS*, pages 136–145. Springer-Verlag, 1990.

[24] Robert Geisler. *Formal Specification for the Integration of Statecharts and Z in a Metamodel-Based Framework*. PhD thesis, Technische Universität Berlin, 1999.

[25] Wolfgang Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.

[26] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[27] David Harel and Amnon Naamad. The statemate semantics of statecharts. Technical report, The Weizmann Institute of Science, October 1995.

[28] Jan-Juan Hiemer. *Statecharts in CSP – Ein Prozeßmodell in CSP zur Analyse von Statemate-Statecharts*. PhD thesis, Technische Universität Berlin, 1998.

[29] C. A. R. Hoare. *Communicating Sequential Processes*. Printice Hall, 1985.

[30] G. J. Holzmann and D. Peled. The state of Spin. In T. A. Henzinger and R. Alur, editors, *CAV '96: 8th Intl. Conf. on Computer Aided Verification*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.

[31] G. E. Hughes and M. J. Cresswell. *A new Introduction to Modal Logic*. Routledge, London, 1996.

[32] C. Huizing, R. Gerth, and W. P. de Roever. Modelling statecharts behaviour in a fully abstract way. In *Proc. 13th CAAP*, volume 299 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[33] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Transactions on Programming Languages and Systems*, 20(2):302–343, March 1998.

[34] Peter Kelb. *Abstraktionstechniken für automatische Verifikationsmethoden*. PhD thesis, Universität Oldenburg, 1995.

[35] Kolyang, T. Santen, and B. Wolff. Towards structure preserving encoding of Z in HOL. Technical Report 986, Arbeitspapiere der GMD, April 1996.

[36] Ye Li. Übersetzung von Statemate Modellen nach MSZ. Diplomarbeit, Technische Universität Berlin, 2000.

[37] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer-Verlag, 1992.

[38] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[39] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *Lecture Notes in Computer Science*. Springer Verlag, December 1997.

[40] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in Promela/SPIN. In *Proc. of WIFT98*, 1998.

[41] Erich Mikk. The MOCES user's guide. http://www.informatik.uni-kiel.de/~erm/MOCES/, 1998.

[42] Erich Mikk. *Semantics and Verification of Statecharts*. PhD thesis, Christian Albrechts Universität Kiel, 2000.

[43] Robin Milner. *Communcating and Concurrency*. Printice Hall, 1989.

[44] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[45] International Standardization Organization. Information technology – Z formal specification language notation – syntax, type system and semantics. International Standard 13568, June 2002.

[46] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264. Springer-Verlag, September 1991.

[47] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. of the Fifth Intl. Symposium on Programming*, number 137 in Lecture Notes in Computer Science. Springer-Verlag, 1982.

[48] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[49] I. Toyn. Innovations in standard Z notation. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 192–213. Springer-Verlag, 1998.

[50] Michael von der Beeck. A comparison of statecharts variants. In Langmaak, de Roever, and Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, 1994.

Index