Syntactic and Semantic Modularisation of Modelling Languages

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
akzeptierte Dissertation
von

Dipl.-Inf. Rüdiger Grammes

Datum der wissenschaftlichen Aussprache: 29.06.2007

Dekan: Prof. Dr. Reinhard Gotzhein

Prüfungskommision:
Vorsitzender: Prof. Dr. Klaus Schneider
Berichterstatter: Prof. Dr. Reinhard Gotzhein
Prof. Dr. Arnd Poetzsch-Heffter

Preface

This thesis was created during my time as a post-graduate student in the Distributed Systems Group at the University of Kaiserslautern.

This thesis would not have been possible without the support, helpful criticism and motivation by my tutor, Prof. Dr. Reinhard Gotzhein. I am grateful for his continuous support in spite of a heavy workload, and for convincing me to continue with my work when I was considering to abandon it.

I want to thank Prof. Dr. Arnd Poetzsch-Heffter for agreeing to review this thesis, and for his helpful comments that lead to the acceptance of a paper to a reputable conference.

My thanks also go to Prof. Dr. Andreas Prinz for his hospitality, and for giving me the opportunity to gain research experience abroad during a week in Norway.

Furthermore, I want to thank Rick Reed for his helpful comments, and his help with the proceedings of the SAM'06 conference.

I want to thank my former colleagues, Joachim Thees, Philipp Schaible, Alexander Geraldy, Ingmar Fliege, Thomas Kuhn, Christian Webel, Marc Krämer, and Philipp Becker, for creating a comfortable work environment, and for the occasional gathering after work.

Finally, I want to thank my family for their continuous support over the years.

Kaiserslautern, February 2007 Rüdiger Grammes

Abstract

Modelling languages are important in the process of software development. The suitability of a modelling language for a project depends on its applicability to the target domain. Here, domain-specific languages have an advantage over more general modelling languages. On the other hand, modelling languages like the Unified Modeling Language can be used in a wide range of domains, which supports the reuse of development knowledge between projects. This thesis treats the syntactical and semantical harmonisation of modelling languages and their combined use, and the handling of complexity of modelling languages by providing language subsets - called *language profiles* - with tailor-made formal semantics definitions, generated by a profile tool. We focus on the widely-used modelling languages SDL [35] and UML [52], and formal semantics definitions specified using Abstract State Machines [27].

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Description . . . 1
- 1.3 Focus and Structure of the Thesis . . . 2

2 Foundations: Modelling Languages and Formal Techniques **4**
- 2.1 Specification and Description Language . . . 4
  - 2.1.1 Agents . . . 4
  - 2.1.2 Communication . . . 5
  - 2.1.3 State Machines . . . 6
- 2.2 Unified Modeling Language . . . 7
  - 2.2.1 Overview . . . 7
  - 2.2.2 Meta-model Architecture . . . 7
  - 2.2.3 UML Diagrams . . . 8
- 2.3 Abstract State Machines . . . 9
  - 2.3.1 States . . . 9
  - 2.3.2 Actions . . . 11
  - 2.3.3 Rules . . . 12
  - 2.3.4 Programs . . . 14
- 2.4 Formal Semantis of SDL . . . 16
  - 2.4.1 Static Semantics . . . 16
  - 2.4.2 Dynamic Semantics . . . 18
- 2.5 Formal Semantics of UML . . . 20

3 Harmonisation of Modelling Languages **21**
- 3.1 Motivation . . . 21
- 3.2 Abstract Syntax Representations . . . 22
  - 3.2.1 The Abstract Grammar Approach . . . 22
  - 3.2.2 The Metamodel Approach . . . 23
- 3.3 Mapping Between Meta-models and Abstract Grammars . . . 24
  - 3.3.1 Classes and Enumerations . . . 25
  - 3.3.2 Attributes . . . 27
  - 3.3.3 Associations . . . 28
  - 3.3.4 Multiplicity . . . 31
  - 3.3.5 Specialisation . . . 31
  - 3.3.6 Meta-Model Approach vs. Abstract Grammar Approach . . . 34
- 3.4 Syntactic Harmonisation of SDL and UML . . . 35

3.5 Semantic Comparison of SDL and UML . . . 41
3.6 The UML Profile Approach . . . 47
3.6.1 Overview . . . 47
3.6.2 UML Profiles . . . 47
3.6.3 The UML Profile for SDL . . . 49
3.6.4 Formalisation of the UML Profile for SDL . . . 50
3.6.5 Survey of an SDL-style Formalisation Approach . . . 51
3.6.6 Conclusions . . . 60
3.7 Related Work . . . 60
3.8 Summary and Conclusions . . . 61

4 Profiling of Modelling Languages 62
4.1 Language Profiles . . . 62
4.2 Consistency of Profiles . . . 63
4.2.1 Consistency for Sequential Abstract State Machines . . . 64
4.2.2 Consistency for Distributed Abstract State Machines . . . 65
4.2.3 Verifying Consistency for Distributed Abstract State Machines . . . 67
4.2.4 Using Structural Information for Verifying Consistency . . . 71
4.3 SDL Profiles . . . 71
4.3.1 Core, Static and Dynamic . . . 71
4.3.2 Cmicro and Cadvanced . . . 72
4.3.3 SDL+/Safire . . . 73
4.3.4 UML Profile for SDL . . . 73
4.3.5 Hierarchy of SDL Profiles . . . 73
4.4 Related Work. . . . 74
4.5 Summary and Conclusions . . . 75

5 Extraction of Language Profiles 76
5.1 Motivation . . . 76
5.2 Language Profile Definition . . . 76
5.3 Approach for Defining Semantics for Language Profiles . . . 77
5.3.1 The Composition Approach . . . 77
5.3.2 The Extraction Approach . . . 78
5.4 Static Semantics . . . 79
5.5 Reduction Profile . . . 82
5.6 Formalisation Signature . . . 84
5.7 Formal Reduction of ASMs . . . 85
5.7.1 Formal Reduction of ASM Definitions . . . 85
5.7.2 Macros, Functions and Parameters . . . 86
5.7.3 Formal Reduction of ASM Rules . . . 88
5.7.4 Formal Reduction of ASM Domains . . . 90
5.7.5 Formal Reduction of ASM Expressions . . . 91
5.8 Verifying Correctness of the Extraction . . . 97
5.8.1 Proof Obligations . . . 97
5.8.2 Proofs over Distributed Abstract State Machines . . . 99

5.8.3 Case Study: Proving Correctness of Extraction . . . 102
5.9 Related Work. . . . 106
5.10 Summary and Conclusions . . . 107
5.11 Future Work . . . 108

6 The SDL-Profile Tool **111**
6.1 Sequence of Steps of the SDL-profile Tool . . . 111
6.1.1 Parse . . . 111
6.1.2 Normalise . . . 112
6.1.3 Remove . . . 112
6.1.4 Clean . . . 112
6.1.5 Iterate . . . 112
6.1.6 Unparse . . . 113
6.2 The Term Processor Kimwitu . . . 113
6.2.1 Defining the Abstract Syntax. . . . 113
6.2.2 Rewriting Abstract Syntax Trees. . . . 114
6.2.3 Unparsing Abstract Syntax Trees. . . . 114
6.2.4 Kimwitu Control Structures. . . . 115
6.3 Implementation Decisions . . . 115
6.3.1 ASM Syntax Definition . . . 115
6.3.2 Implementing the Predicates . . . 116
6.3.3 Implementing the Remove Function . . . 117
6.3.4 Referencing Definitions . . . 117
6.3.5 Transforming Trivial Rules and Expressions . . . 119
6.3.6 Generating Proof Obligations . . . 119
6.3.7 Output of the Reduced Abstract Syntax Tree . . . 120
6.4 Implementation of the SDL-Profile Tool . . . 123
6.4.1 Executing the SDL-Profile Tool . . . 123
6.4.2 Parsing the Reduction Profile . . . 124
6.4.3 Processing the Semantics Definition . . . 124
6.5 Application of the SDL-Profile Tool . . . 126
6.5.1 Extracting a Profile without Timers . . . 127
6.5.2 Extracting a Profile without Inheritance . . . 129
6.5.3 Size of Extracted SDL Profiles . . . 134
6.6 Summary and Conclusions . . . 135
6.7 Future Work . . . 136

7 Conclusions **138**
7.1 State of the Art . . . 138
7.1.1 Combined Use of Modelling Languages . . . 138
7.1.2 Language Profiles and Modular Language Definition . . . 139
7.2 Future Work . . . 140

A UML Kernel Abstract Grammar **141**

B Agent Moves for Ping-Pong System **147**

C Complete Formal Definition of Extraction **149**
C.1 Definitions . . . 149
C.2 Macros, Functions and Parameters . . . 149
C.3 Rules . . . 150
C.4 Domains . . . 151
C.5 Expressions . . . 153

D Reduction Profiles for SDL Features **160**

E Syntax of Abstract State Machines **163**

List of Tables

3.1 Rules and their meanings 23
3.2 Mapping of Multiplicities 31
3.3 Common syntax of packages 36
3.4 Common syntax of agent types and classes 37
3.5 Common syntax of signals 37
3.6 Common syntax of channels and connectors 38
3.7 Common syntax of gates and ports 38
3.8 Common syntax of agents and properties 39
3.9 Common syntax of parameters 39
3.10 Common syntax of composite states and state machines 40
3.11 Common syntax of states 40

5.1 Truth table for negation 92
5.2 Truth table for disjunction 92
5.3 Derived truth table for conjunction 93
5.4 Truth table for element-of operator 93
5.5 Truth table for universal quantification 94
5.6 Truth table for existential quantification 95

6.1 Abstract syntax nodes and corresponding definitions 118
6.2 Semantically equivalent transformations 119
6.3 Important command line options of the SDL-profile tool 124
6.4 Size of extracted SDL profiles 134
6.5 Size of extracted SDL profiles 135

B.1 Moves performed by ping-pong ASM agents 148
B.2 SDL entities and corresponding ASM agents 148

List of Figures

2.1 SDL agent PingPong with substructure 5
2.2 Agent type Ping 6
2.3 Meta-model hierarchy of UML 7
2.4 Static semantics of SDL 17
2.5 Agent mode *execution* (level 2) and submodes 19

3.1 Excerpt of the abstract syntax of states 24
3.2 Submachine (composite) states and entry/exit-points in SDL and UML 42
3.3 Priority of consume and defer 43
3.4 State with two enabled transitions (UML) 44
3.5 Transitions in SDL and UML 44
3.6 Continuous signal, connect node and completion transition 45
3.7 Profile package with stereotype 48
3.8 Stereotyped class 49
3.9 Mapping from UML meta-model to SDL abstract syntax 51

4.1 m:n refinement relation 64
4.2 Consistency of language profiles (1:1 refinement) 64
4.3 Consistency with stuttering steps (m:1 refinement) 65
4.4 Partially ordered run 66
4.5 Possible executions of a partially-ordered run 66
4.6 Two agents performing causally unrelated moves 67
4.7 Possible executions for causally unrelated moves 68
4.8 Relating update sets of two DASMs 71
4.9 Superset relationship between SDL profiles 74

5.1 Reduced production rule of the SDL abstract syntax 79
5.2 Well-formedness condition for state names 80
5.3 Mapping states from AS0 to AS1 81
5.4 Concept of the extraction process (for SDL) 82
5.5 Reduction Profile for 'Save' Feature 84
5.6 Removed part of semantics definition 98
5.7 Reduction Profile for save Feature 102
5.8 Reduction profile for 'Timer' feature 104
5.9 Reduction profile for 'Exception' feature 105

6.1 Sequence of steps of the SDL-profile tool 111

6.2 Reduction profile for save feature (tool syntsx) 124
6.3 Excerpt of reduction profile for timer feature 127
6.4 Excerpt of reduction profile for inheritance feature 129

B.1 SDL specification of ping-pong system 147
B.2 Partially-ordered run for ping-pong system 148

List of Listings

2.1 The program for the philosopher example 15
2.2 Behaviour primitive for setting timers 18
2.3 Compilation of *Assignment* node to behaviour primitive TASK 20

4.1 Replacing choose constructs 68
4.2 Application of the choose replacement 69
4.3 Structurally related ASMs 71

5.1 Abstract Syntax of Reduction Profiles 83
5.2 Functions of the SVM 84
5.3 Modular definition of timer instruction set 109
5.4 Modular definition of entry procedures 109

6.1 Excerpt from the abstract syntax definition of ASMs 113
6.2 Definition of list types 114
6.3 Defining attributes for node types 114
6.4 Transforming expressions with rewrite rules 114
6.5 Unparsing the addition operator 114
6.6 Computation using unparse rules 115
6.7 Abstract syntax for ASM rules 116
6.8 Interface of the predicates 116
6.9 Implementation of predicate *true* for conjunction 116
6.10 General definition of remove functions 117
6.11 Abstract syntax for rules with proof obligation 120
6.12 Macro SELECTTRANSITIONSTARTPHASE before reduction 127
6.13 Macro SELECTTRANSITIONSTARTPHASE after reduction 127
6.14 Rule macros for setting timers 127
6.15 Extracted semantics definition 129
6.16 Derived functions for inheritance and refinement 129
6.17 Extracted derived functions for inheritance and refinement 133
6.18 Excerpt from an ASM semantics for UML statemachines 136

1 Introduction

1.1 Motivation

In order to support a wide range of applications, system modelling languages are often complex and expressive. This leads to language definitions that are long and hard to understand, and can limit language applicability in domains for which specialised, tailor-made languages are preferred. Another drawback is that tool support for complex languages usually covers only parts of the languages. For example, there is no tool that supports all features of SDL-96 [13, 31, 32], and only a few of the language features introduced in SDL-2000 [35, 37, 36] are supported. This raises questions how the complexity of language definitions can be coped with, and how language definitions - both formal and informal - can be structured to support the definition of domain-specific sublanguages.

For specific problems and project phases, different modelling languages are suitable. For example, UML [50, 52] is often used for the early phases of software development. Further aspects in the choice of modelling languages are the preferences of the developer, existing specifications in a certain language, and tool support. The combined use of modelling languages enables a developer to utilise the advantages of the individual languages, and to combine existing specifications with new specifications in a different language. The combined use of modelling languages requires techniques for the exchange and integration of models between the languages.

1.2 Problem Description

The goal of this thesis is to provide and apply sound approaches for the syntactical and semantical modularisation of modelling languages. This goal can be split up into a number of problem areas that are further characterised below.

- **Horizontal modularity** is concerned with the modular structure within a language. The goal is to achieve a modular language definition that splits a language into a language core and a hierarchy of language modules that extend this core. Language modules encapsulate language features and can be added to the language core, yielding sublanguages targeted at specific application areas. We call these tailor-made sublanguages *language profiles*.

  Horizontal modularity is also concerned with the harmonisation of modelling languages. Modelling languages like SDL and UML have a lot of features in common. These features can be defined in a common language core. On top of this core, a hierarchy of language modules for each langauge can be based.

An object of research is the nature and size of the language core. The language core should contain the most important language constructs, and be small enough to apply to different problem areas. The core must be extensible, both syntactically and semantically. For language modules, the object of research is how language features can be encapsulated, and how the extension of the core can be achieved.

- **Vertical modularity** is concerned with the modular structure within language families like SDL and MSC, or between different views in modelling languages, like statecharts and interactions in UML. Object of research is the common semantic core of the related modelling languages. The common core should contain all language features that are common to the languages based on the core. These languages must have a sufficient amount of common features, otherwise the resulting core would be too small and be irrelevant. Given sound techniques for horizontal modularity in addition, the common language core can be extended with language modules for each language based on this core.
- **Optional modularity** is concerned with *semantic variation points* and incomplete semantics. These can be expressed in a language definition by non-determinism. Formalisms like ASMs provide constructs for implicit and explicit non-determinism.
- **Temporal modularity** is concerned with language extensions and modifications that come with new language versions. Given a modular language definition, conservative language extensions can be introduced as new language modules.
- **Hybrid modularity** denotes any combination of the forms of modularity mentioned above, for example, a common semantic core, with language modules for different languages and language versions based on the core.

1.3 Focus and Structure of the Thesis

The focus of this thesis lies on the harmonisation between modelling languages, and on the definition of language profiles (horizontal modularity). Harmonisation between modelling languages in concerned with the integration of languages taking into account their underlying similarities and differences. Language profiles define subsets of a language, from which tailor-made formal semantics definitions can be derived.

For the harmonisation of modelling languages, we have selected the widely-used Unified Modelling Language (UML) [52] and the Specification and Description Language (SDL) [35]. Harmonisation of these languages has been an ongoing topic in research and industry over the last couple of years. Both languages have a lot of features in common. However, the languages come from different backgrounds, have a different focus and differently structured language definitions. UML aims to be a universal modelling language for a large number of domains, with incomplete and informal semantics, aimed at the early development phases. SDL is a language with formal semantics, which has a more specific application domain, but is more suited for

detailed design, simulation and code generation. We contribute a mapping between the abstract syntax representations of UML (meta-models) and SDL (abstract grammars), perform a syntactic and semantic comparison, and derive a common abstract syntax of UML and SDL. We propose and evaluate an approach for the formalisation of the UML Profile for SDL. Work in this area was published in the proceedings of the FORTE 2004 conference [24], and in technical reports [23, 22].

For profiling languages, the focus lies on the behaviour specification, specifically behaviour specified by abstract transition systems. We define consistency between the dynamic semantics of language profiles, and outline its verification. We examine the effect of defining language profiles on the static semantics of a language, and introduce an approach to extract a reduced formal semantics definition for a given language profile. For the practical application of our approach, we focus on SDL, which has a complete formal semantics defined using Abstract State Machines (ASMs) [27, 28, 8]. However, the results apply to other formal semantics defined using ASMs as well. Work in this area was published in the proceedings of the SAM 2006 workshop [21, 18], the FASE 2007 conference [26], and as a technical report [22].

This thesis is structured as follows: We introduce modelling languages and formal techniques covered in this thesis in Chapter 2. Chapter 3 treats abstract syntax representations and the harmonisation of modelling languages SDL and UML. Chapter 4 introduces language profiles, and makes general observations on the consistency of formal language definitions. Chapter 5 formalises an extraction-based approach for the generation of formal semantics definitions for language profiles, with tool support and results discussed in Chapter 6. We draw conclusions and give an outlook on future work in Chapter 7.

2 Foundations: Modelling Languages and Formal Techniques

This chapter introduces modelling languages and formal techniques covered in this thesis. Modelling languages covered are the Specification and Description Language (SDL) and the Unified Modeling Language (UML). We introduce the formal technique Abstract State Machines (ASM), and outline the formal definitions of SDL and UML.

2.1 Specification and Description Language

The Specification and Description Language (SDL) [13, 35, 37, 36] is a formal language standardised by the International Telecommunications Union (ITU), widely used both in industry[1] and academia. It is based on the concept of asynchronously communicating extended finite state machines, running concurrently or in parallel. SDL provides language constructs for the specification of nested *system structure*, *communication* using channels, signals and signal queues, *behaviour* using extended finite state machines, and *data*.

In 1988, the semantics of SDL was formally defined, upgrading the language to a formal description technique. In 1999, a new version of the language, referred to as SDL-2000, was introduced. Since the formal definition of the old semantics was assessed as being too difficult to extend and maintain, a new formal semantics, based on Abstract State Machines, was defined from scratch [19].

2.1.1 Agents

The structure of an SDL system is described using *SDL agents*. An SDL agent can contain variables, procedures, a state machine, and a substructure consisting of SDL agent sets and channels. An *SDL agent set* contains SDL agents of a certain type. A specified number of SDL agents are created initially for each SDL agent set. SDL agents can be created and stopped dynamically.

The contained instances of an SDL agent are either interpreted concurrently (agent kind BLOCK) or in an interleaving manner (agent kind PROCESS). The outermost agent of an SDL specification is a special kind of block (agent kind SYSTEM) that can interact with the environment of open SDL systems.

Figure 2.1 shows SDL agent PingPong, with a substructure consisting of agent type references Ping and Pong, two SDL agent sets containing SDL agents of type Ping

[1] Several tool developers offer tool support for SDL and related languages, for example Telelogic, Cinderella, PragmaDev and Solinet.

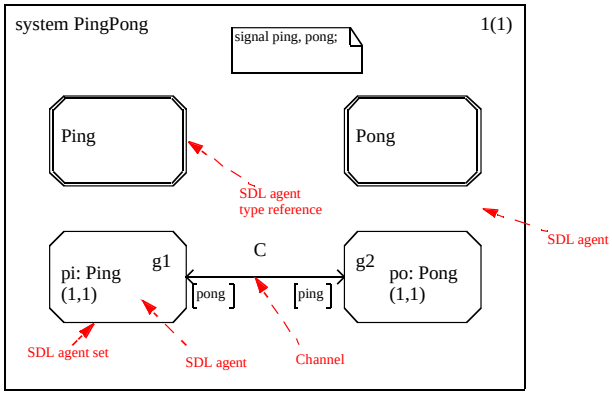


Figure 2.1: SDL agent PingPong with substructure

and Pong, respectively, and a channel C connecting the agents sets. Initially, one SDL agent of type Ping and Pong is created.

2.1.2 Communication

In SDL, agents communicate by exchanging signals over a communication structure. Signals are generated and consumed by state machines. A signal has a type, and can contain data and addressing information. A signal can be addressed explicitly by sending it to a process identifier, or implicitly by sending it either to an agent identifier or with no target information. The route a signal takes can be further constrained to certain channels and gates.

Channels and gates form the communication structure of an SDL system. Gates define the interface of an agent, as the list of signals that can be sent to and from the agent. Gates have associated signal queues containing the signals waiting at this gate for transmission over a channel, or for delivery to an agent. A special gate is the *inport* associated with each agent instance, with signals waiting for consumption by the state machine of the agent instance. Channels connect agents with each other and the environment via their gates. A channel can be uni- or bi-directional, delaying or not delaying, and have a list of signal types that can be sent over this channel. Signals can not be lost, duplicated or reordered by a channel.

Signals are generated by output actions of state machines, and are placed in a suitable outgoing gate of the agent set, according to the addressing information of the signal. Signals are forwarded over channels and gates to a valid destination, where they are placed in the inport of the agent.

In Figure 2.1, a bi-directional channel C connects the gates g1 of agent set pi and g2 of agent set po. Signals of type ping can be sent from pi to po, and signals of type pong from po to pi. The interfaces defined by gates g1 and g2 must match the signal lists defined for channel C: g1 (g2) must offer ping (pong) and accept pong (ping).

2.1.3 State Machines

State machines, together with the nested agents in the substructure, define the behaviour of an SDL agent. State machines define states, transitions between states, signal consumption and actions, for example signal output, setting timers and variables. SDL-2000 introduces hierarchical state machines, structured by *composite states*. A composite state has a substructure consisting of state-nodes (which can again be composite states) and transitions.

The topmost state of an SDL statemachine is a composite state, representing the state machine. A composite state is refined either into a *composite state graph*, consisting of state-nodes and transitions, or into a *state aggregation*, which is a parallel composition of composite states with disjoint interfaces, called *state partitions*. This assures that for a given signal, only one of the state partitions can fire a transition.

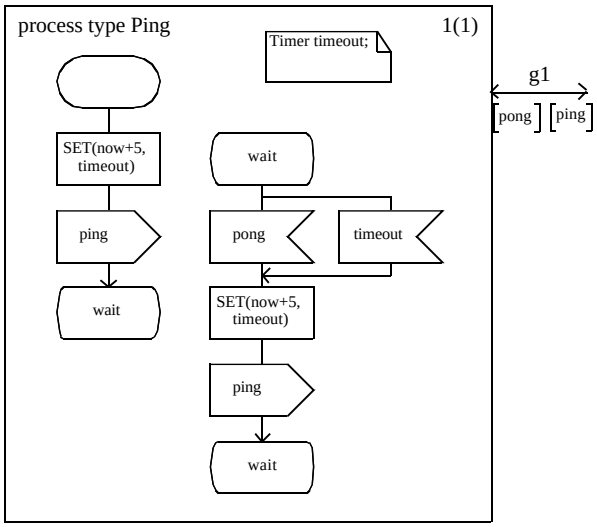


Figure 2.2: Agent type Ping

State transitions are triggered by signals in the inport of the agent, or spontaneously. SDL defines several kinds of transition triggers:

- **Priority input:** If a signal of the specified type[2] is in the inport, it is consumed and the transition is fired. Priority input transitions have precedence over other transition kinds, except for spontaneous transitions.
- **Input:** The first signal in inport is consumed and the corresponding transition is fired. If no input node exists for the first signal of the inport, it is either discarded or saved (remaining at the front of the inport), and the next signal in inport is checked.

[2] Priority is associated with the trigger of a transition, not with the signal type.

- **Continuous signal:** If inport is empty and the guard is true, the transition is fired.
- **Spontaneous:** A spontaneous transition is fired indeterministically.

Figure 2.2 shows the state machine definition of agent type Ping. The start transition (left) sets timer timeout, outputs signal ping[3], and enters state wait. In state wait, Ping either receives signal pong, or the timeout signal. In both cases, the timer is reset, and a new ping signal is sent.

2.2 Unified Modeling Language

2.2.1 Overview

The Unified Modeling Language (UML, [50, 52]) is a graphical specification language for object modelling, standardised by the Object Management Group (OMG). UML aims at being a universal modelling language for modelling software, hardware, and processes for a large variety of application domains. It is best suited for modelling software systems in an object-oriented fashion. A UML specification defines an abstract model of a system, using model elements and relationships between them.

2.2.2 Meta-model Architecture

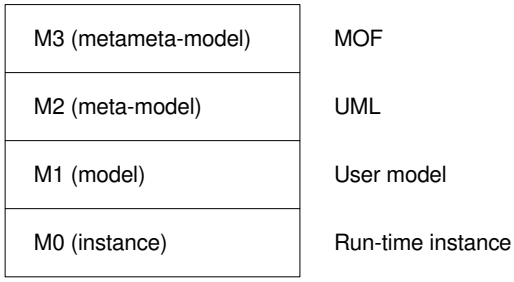


Figure 2.3: Meta-model hierarchy of UML

UML defines a four-layered meta-model hierarchy (see Figure 2.3), based on the MetaObject Facility (MOF, [51, 53]). In this hierarchy, each layer is an instance of the directly superordinated layer.

- The topmost layer (M3) is a metameta-model that defines a language for specifying a meta-model. The metameta-model is reflexive, meaning it is an instance of itself.

[3]Due to the structure of system PingPong, signal ping can only take one communication path. Therefore, no addressing information is needed.

- Layer M2 is the set of meta-models defining languages for specifying user models. The abstract syntax of UML is defined as a meta-model.
- Layer M1 is the set of UML models - instances of the UML meta-model - as specified by the user. The models define a languages for describing application domains.
- Layer M0 is the set of run-time instances of UML models.

2.2.3 UML Diagrams

UML introduces several types of diagrams as graphical representation of parts of the model. Each diagram type describes a certain aspect of a model, for example static relationships between classifiers for *class diagrams*. There are two groups of diagram types: structure diagrams describe the static structure of objects in a system, behaviour diagrams show their dynamic behaviour. A set of model elements is associated with each diagram type. However, UML does not define strict boundaries between the different types, and allows mixing several diagram types in a single diagram.

Structure Diagrams

- **Class Diagrams** describe classes and interfaces, and their relationship by association, aggregation and composition. Class diagrams are one of the most common types of UML diagrams.
- **Package Diagrams** describe packages and their relationship by package extension, import and merge.
- **Object Diagrams** describe objects and links between them. Object diagrams are the instance-level (M0) counterpart of class diagrams.
- **Component Diagrams** describe the physical components that make up a system, and their dependencies.
- **Composite Structure Diagrams** describe the internal structure of a class, consisting of classes, parts and connectors. Composite structure diagrams describe nested system structure similar to agents in SDL.
- **Deployment Diagrams** describe the execution architecture of a system by assigning software artifacts to computational resources.

Behaviour Diagrams

- **Activity Diagrams** describe activities, and their control and object flow.
- ***Interaction Diagrams*** describe the interactions between entities of a system. There are four kinds of interaction diagrams, which provide different *views* on the same part of the model:

  - **Sequence Diagrams** describe interactions between entities of a system, with the focus on the causal order of send and receive events. Sequence diagrams correspond to MSCs [34] in SDL [30].
  - **Communication Diagrams** describe interaction between entities of a system, with the focus on the architecture and relationships between entities.
  - **Interaction Overview Diagrams** describe interactions between entities of a system on a higher level of abstraction, using a combination of an activity-like notation and sequence diagrams. Interaction overview diagrams correspond to HMSCs [34] in SDL.
  - **Timing Diagrams**, introduced in UML 2.0 [52], describe the state changes of entities over time.

- **Statemachine Diagrams** describe discrete behaviour with finite state transition systems.
- **Use Case Diagrams** provide a graphical overview of the functional requirements of a system.

2.3 Abstract State Machines

Abstract State Machines (ASMs) [27, 28, 8] are a general model of computation introduced by Yuri Gurevich. They combine declarative concepts of first-order logic with the abstract operational view of distributed transition systems. ASMs are based on sorted first-order structures, called *states*. A state consists of a signature (or vocabulary) containing domain names, function names, and relation names, together with an interpretation of these names over a superuniverse X. A state can be viewed as a memory snapshot of the ASM, where locations - identified by functions and parameter values - are mapped to result values.

The computation model of Distributed ASMs is based on a set of autonomously operating ASM agents. Starting from an initial state, the agents perform concurrent computations and interact through shared locations (see Section 2.3.2) of the state. The behaviour of ASM agents is determined by ASM programs, consisting of ASM rules. Complex ASM rules are defined as compositions of guarded update instructions using a small set of rule constructors. From these rules, update sets, i.e. sets of memory locations and new values, are computed. These update sets define state transitions that result from applying all updates simultaneously.

2.3.1 States

As abstract representation of states of arbitrary algorithms, ASMs rely on structures from mathematical logic. Structures were first introduced by Tarski in 1936 and are an accepted standard in mathematics. Structures are based on set theory - they consist of a base set X, called *superuniverse*, together with an interpretation of the names from

a vocabulary V in X. Structures, slightly modified for dynamic purposes, describe the states of an ASM.

Vocabularies

A *vocabulary* V is a finite collection of *domain*, *function*,*relation*, and *variable names* with fixed arity. Names in the vocabulary can be marked as *static*, meaning they have the same interpretation in all states of the ASM, or *dynamic*. Function and relation names can be specified as either *monitored*, *controlled*, or *shared*. Controlled functions can only be modified by the machine, while monitored functions can only be modified by the environment. Shared functions can be modified both by machine and environment. By default, functions are controlled and dynamic. Every vocabulary of an ASM contains the following static names: the equality sign, the 0-ary functions `true`, `false`, and `undefined`, and the common boolean operators. The vocabulary can be extended by a set of variables names.

Below is the vocabulary V_P for a specification of the dining philosophers, consisting of the domain names PHILOSOPHER, FORK, and MODE, the function names *numphil*, *left*, *right*, *owner*, and *mode*, and the relation name *hungry*. Since the setup of philosophers and forks doesn't change, *numphil*, *left*, and *right* are marked as static.

```
1  static domain Philosopher
2  static domain Fork
3  static domain Mode
4
5  static numphil: → ℕ
6  static left: Philosopher → Fork
7  static right: Philosopher → Fork
8
9  owner: Fork → Philosopher
10 mode: Philosopher → Mode
11
12 monitored hungry: Philosopher → Boolean
```

Terms can be constructed syntactically from variable names and names from the vocabulary. Every variable is a term, and if f is a function (relation) name with arity r and $t_1, \ldots t_r$ are terms, $f(t_1, \ldots, t_r)$ is a term. If v is a variable and g and t are boolean terms, $\forall v : g.\, t$ and $\exists v : g.\, t$ are terms.

For example, *owner(left(p))* is a term with variable name p and function names *owner* and *left*. $\exists p : hungry(p).(owner(left(p)) = p)$ is a first-order term over V_P.

States

A *state* A of an ASM consists of a non-empty superuniverse X and an interpretation of the names of vocabulary V in X. A domain name in V is interpreted as a *universe*, a subset of X. Universes can be defined as unary predicates that hold for elements in the domain. A function name f in V with arity r is interpreted as a function f_A from X^r to X.[4] A relation name in V with arity r is interpreted as a function

[4] We use function and relation names in the vocabulary that are typed by domain names. We interpret these types as constraints on the ASM.

from X^r to {`true`, `false`}. Variable names are interpreted as 0-ary functions, denoting elements of the superuniverse. The 0-ary constants `true`, `false` and `undefined` denote distinct elements of the superuniverse X. Superuniverse and vocabulary are fixed for the states of an ASM, the interpretation for non-static names of the vocabulary can change between states.

In order to support algorithms that allocate additional space dynamically, every state of an ASM contains an infinite store of "fresh" elements, called the *reserve*. Elements of the reserve have the following properties:

- Every basic relation with the element as an argument, with the exception of equality, evaluates to `false`.
- Every function with the element as an argument evaluates to `undefined`.
- The element is not included in the image of any function, or in a domain.

The reserve is defined as the set of elements that satisfy these properties.

An ASM has a set of *initial states* as possible start states. This set can be characterised by constraints on the initial state. In the philosophers example, the initial state has at least two philosophers and an equal number of forks. Each philosopher has a fork to his left and right, and shares each fork with another philosopher. All philosophers are thinking, and none of the forks are owned by any of the philosophers.

```
1  initially numPhil > 1
2
3  initially Philosopher = {p_0, ..., p_numPhil-1}
4  initially Fork = {f_0, ..., f_numPhil-1}
5  initially Mode = {thinking, waiting, eating}
6
7  initially ∀p_i: Philosopher. mode(p_i) = thinking
8  initially ∀p_i: Philosopher. left(p_i) = f_(i-1) mod numPhil
9  initially ∀p_i: Philosopher. right(p_i) = f_i
10
11 initially ∀f: Fork. owner(f) = undef
```

2.3.2 Actions

State transitions in Abstract State Machines are performed by reinterpreting the state in a bounded number of locations. A transition is described by an update set - a set of locations together with the new values of the locations.

Locations

A *location* l of a state A over vocabulary V is a pair $(f, \overrightarrow{a})$. f is a function or relation name of V, $\overrightarrow{a}$ is a tuple of elements from the superuniverse X of A with a size corresponding to the arity of f. In case f is a relation name of V, location l is called *relational*. The *content* of a location is the output of the function $f_A(\overrightarrow{a})$.

Updates

An *update* is a pair (l, v) of a location l and an element v of X. Firing an update at a state S changes the interpretation of S at l to the new value v. If l is relational, v must be boolean-valued.

An *update set* α is a set of updates. An update set is *consistent*, if there are no two updates in the set that write different values to the same location: $\forall(l_1, v_1), (l_2, v_2) \in \alpha.(l_1 = l_2 \rightarrow v_1 = v_2)$. Firing a consistent update set at a state S fires every update in the set simultaneously. Firing an *inconsistent* update set has no effect on the state.

2.3.3 Rules

Rules describe state transitions of the Abstract State Machine. Given a state of the ASM, an update set can be calculated from a rule. Firing the update set of the rule at the state results in the subsequent state of the ASM. In sequential abstract state machines, a rule fires in every state of the ASM. Further execution semantics are described in Section 2.3.4.

Basic Update Rule The basic update rule has the form $f(\overrightarrow{a}) := t$ with a function or relation f, a tuple of terms $\overrightarrow{a}$ with a number of elements according to the arity of f, and a term t. A basic update writes the value of t in state S at the location described by $f(\overrightarrow{a})$.

$$\mathrm{Update}(f(\overrightarrow{a}) := t, S) = \{((f, \langle \mathrm{val}_S(\overrightarrow{a})\rangle), \mathrm{val}_S(t))\}$$

Guarded Updates Rules can be guarded with a first-order term, firing the update set associated with the rule only in states in which the guard holds. A second rule can be specified that is fired instead in states in which the guard doesn't hold.

$R \equiv$ **if** g **then** R_1 **else** R_2 **endif**

$$\mathrm{Update}(R, S) = \begin{cases} \mathrm{Update}(R_1, S) & \mathrm{val}_S(g) \\ \mathrm{Update}(R_2, S) & \text{else} \end{cases}$$

Rule Blocks Rules can be collected in rule blocks that are executed in parallel. The resulting update set is the union of the update sets of the individual rules. If the resulting update set is inconsistent, the state is not modified[5]. Parallel execution of rules is the default, so the **do in-parallel** keyword can be omitted.

$R \equiv$ **do in−parallel** $R_1 \ldots R_n$ **enddo**

$$\mathrm{Update}(R, S) = \mathrm{Update}(R_1, S) \cup \ldots \cup \mathrm{Update}(R_n, S)$$

[5] In particular, if any of the rules in the rule block produces an inconsistent update set, the update set of the rule block is inconsistent.

Importing Elements ASMs can allocate additional resources by importing fresh elements from the reserve, using **extend**. The **extend**-rule binds reserve elements to variables $v_1, \ldots, v_n$, and fires rule R with these bindings. If the resulting update set is consistent, the elements are removed from the reserve and inserted into domain D in the subsequent state. Otherwise, they remain in the reserve.

extend D **with** $v_1, \ldots, v_n$
  R
endextend

Bounded Parallelism For a finite number of non-reserve elements for which the guard g holds, rule R can be fired in parallel with variable v bound to a different element from the range of the guard. BaseSet(S) is the base set X of state S.

do forall v: $g(v)$
  $R(v)$
endextend

$$\text{Range}_S(v : g(v)) = \{a \in \text{BaseSet}(S) : a \notin \text{Reserve} \wedge S \models g_S(a)\}$$

$$\text{Update}(R, S) = \bigcup \{\text{Update}(R, S(v \rightarrow a)) : a \in \text{Range}_S(v : g_S(v))\}$$

Non-deterministic Choice An element is selected non-deterministically from the range of the guard g. Rule R is fired with variable v bound to this element. Nondeterministic choice must only range over a finite number of elements. Choosing from an empty set results in an empty (not inconsistent) update set.

choose v: $g(v)$
  $R(v)$
endextend

For rules with non-deterministic choice, Update produces a set of update sets, representing the possible update sets produced by the choice rule. Update for non-deterministic ASMs is defined in [28], Chapter 6.

Another way to model non-deterministic choice is to introduce a monitored function, constrained by g.

Shortcuts As a shortcut for long terms, the result of evaluating a term t in state S can be bound to a variable x for a rule $R(x)$. This is equivalent to the rule $R(x)$ with every occurrence of x replaced by term t.

let $x = t$ **in**
  $R(x)$
endlet

2.3.4 Programs

Sequential Abstract State Machines

A *run* of a Sequential Abstract State Machine is a sequence of states $A_0 \dots A_n$ over vocabulary V, where A_0 is an initial state, and state A_{n+1} results from firing the rules of the ASM on state A_n. Sequential ASMs can be seen as a special case of Distributed ASMs with a single agent.

$$A_0 \xrightarrow{\delta} A_1 \xrightarrow{\delta} A_2 \xrightarrow{\delta} \dots \xrightarrow{\delta} A_n \tag{2.1}$$

Distributed Abstract State Machines

In *Distributed Abstract State Machines* (DASMs), several ASMs act as *agents* performing *moves* on a shared state. Each agent is represented by an element of the superuniverse X. The vocabulary of a DASM contains the following additional names: domain names AGENT and PROGRAM, the sets of elements representing ASM agents and programs; the function names *program*, assigning programs to agents, and *Self*, identifying the element corresponding to the agent executing the program.

```
1 domain AGENT
2 domain PROGRAM
3
4 program: AGENT → PROGRAM
5 monitored Self: → AGENT
```

Programs are rules without free variables. An agent performs a move by firing its program at the current state. Agents can make moves on the shared state concurrently and in parallel, restricted by the notion of *partially-ordered runs*.

Partially-ordered runs [27] define a *coherence condition* that imposes a minimal restriction on the parallel execution of ASM agents[6]. Formally, partially-ordered runs are defined as follows: A partially ordered run of a Distributed ASM is a triple (M, A, σ), with a partially-ordered set M of moves performed by the agents of the ASM, a function A mapping moves to agents, and a function σ mapping sets of moves to states. The following properties hold for partially-ordered runs:

- For every move m of M, the set of predecessors $\{x : x < m\}$ is finite.
- The set of moves by a single agent a, $M_a = \{m \in M : A(m) = a\}$, is a total order with regard to the ordering relation of M, restricted to elements of M_a.
- An *initial segment* is a substructure I of M so that for every $m \in I$ and $x < m$ in M follows $x \in I$. σ maps an initial segment I to a state that is the result of performing all moves in I. $\sigma(\emptyset)$ is an initial state.

[6] The parallel execution of ASM agents should not be confused with parallel updates within an ASM rule

- *coherence condition*: If m is a maximal element in a finite initial segment[7] X of M and $Y = X - \{m\}$, then $a = A(m)$ is an agent in $\sigma(Y)$, m is a move of a and $\sigma(X)$ is obtained from $\sigma(Y)$ by performing m at $\sigma(Y)$.

From the coherence condition it follows that for a finite initial segment, all linearisations produce the same final state.

Listing 2.1 shows the program of the Distributed ASM for the dining philosopher example. The program consists of three **if**-rules executed in parallel. A thinking philosopher changes into mode waiting when hungry. The philosopher waits until the forks to the left and right are free, then acquires them and goes into eating mode. Since the forks are acquired in a parallel, atomic step, no deadlock can occur. An eating philosopher releases the forks and goes into thinking mode once he is not hungry.

```
1  PhilosopherProgram ≡
2    if hungry(Self) ∧ mode(Self) = thinking then
3      mode(Self) := waiting
4    endif
5    if mode(Self) = waiting then
6      if owner(left(Self)) = undef ∧owner(right(Self)) = undef then
7        owner(left(Self)) := Self
8        owner(right(Self)) := Self
9        mode(Self) := eating
10     endif
11   endif
12   if mode(Self) = eating ∧ ¬ hungry(Self) then
13     owner(left(Self)) := undef
14     owner(right(Self)) := undef
15     mode(Self) := thinking
16   endif
17
18 initially ∀p_i: Philosopher. p_i ∈ Agent
19 initially ∀p_i: Philosopher. program(p_i) = PhilosopherProgram
```

Listing 2.1: The program for the philosopher example

The coherence condition ensures that two philosophers can not acquire the same fork at the same time. Take two philosophers p_0, p_1 in waiting mode, and fork f_0 with an undefined owner. Philosopher p_0 (p_1) can make a move m_0 (m_1) acquiring fork f_0. Given the partially ordered set $\{m_0, m_1\}$ with both moves unordered, all linearisations must produce the same final state. Executing m_0 before m_1 results in p_0 acquiring f_0 while p_1 remains in waiting mode. Executing m_1 before m_0 results in p_1 acquiring f_0 while p_0 remains in waiting mode. This contradicts the coherence condition. Therefore, in a valid partially ordered run, either $m_0 < m_1$ or $m_1 < m_0$ must hold.

The Environment

In the case of Sequential ASMs, the environment can be seen as an entity changing the interpretation of monitored and shared functions in between two steps of the ASM. In

[7] This requires the set of agents to be finite.

the case of Distributed ASMs, the environment can be seen as a set of invisible agents running concurrently or in parallel to the agents of the DASM, while subject to the coherence condition.

Generally, the environment should not change the monitored and shared functions in an arbitrary fashion. Therefore, the behaviour of the environment can be restricted, for example by formulas of temporal logic, to exclude unexpected behaviour. For the dining philosophers example, we expect hungry philosophers to stay hungry until they start eating, and to eventually stop being hungry after starting to eat. This is expressed in the following constraints.

constraint $\forall p_i$: PHILOSOPHER. $\Box\ (hungry(p_i) \rightarrow (hungry(p_i)\ U\ mode(p_i) = eating))$
constraint $\forall p_i$: PHILOSOPHER. $\Box\ (mode(p_i) = eating \rightarrow \Diamond\ \neg hungry(p_i))$

Real-time Abstract State Machines

Real-time Abstract State Machines are Distributed Abstract State Machines that introduce a notion of *real time*. Agents perform instantaneous actions in continuous time, defined by the monitored, real-valued function *currentTime*. Function *currentTime* must satisfy constraints that ensure behaviour according to the nature of physical time. For example, *currentTime* must increase monotonically over an ASM run.

monitored *currentTime*: $\rightarrow$ REAL

2.4 Formal Semantis of SDL

In November 2000, the formal semantics of SDL-2000, the current version of SDL, was officially approved to become part of the SDL language definition (for a detailed survey, see [16] and [19]). It covers all static and dynamic language aspects, and consists of two major parts:

- The *static semantics* of SDL defines well-formedness conditions on the concrete syntax of SDL. Furthermore, transformations map extended features of SDL to core features of the language, reducing the complexity of the dynamic semantics. The static semantics contains over 5600 lines of specification.
- The *dynamic semantics* of SDL defines the dynamic behaviour of well-formed SDL specifications, based on ASMs. At the core of the dynamic semantics is the *SDL Virtual Machine* (SVM), which includes an abstract machine for the execution of SDL specifications, plus operating system functionality. A *compilation function* maps actions from transitions in an SDL specification to instructions of the SVM. The dynamic semantics contains over 2800 lines of ASM specification.

2.4.1 Static Semantics

The static semantics of SDL [41] defines which syntactically correct SDL specifications are valid, how extended features of the language are transformed to basic features of

the language, and how the mapping between the different kinds of syntax (CS, AS0, AS1) is performed. SDL has two kinds of abstract syntax. The abstract syntax AS0 corresponds to the concrete syntax (CS) of SDL, without delimiters and keywords. The abstract syntax AS1 is the basis for the specification of the dynamic semantics of SDL. Its structure is simplified compared to CS and AS0, and only elements covered by the dynamic semantics are included. Figure 2.4 shows the steps defined in the static semantics of SDL from the concrete syntax to the abstract syntax AS1.

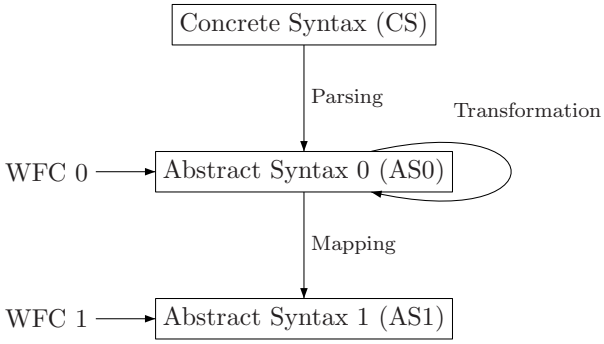


Figure 2.4: Static semantics of SDL

1. The *parsing* step defines the relationship between the concrete syntax CS and abstract syntax AS0. This step is not formally defined, but is implicitly given by the direct correspondence between the two syntax definitions. Delimiters and keywords are omitted, and the textual specification is tranformed into an abstract syntax tree.

2. On the abstract syntax AS0, several *transformation* steps are performed. During the transformation, names are resolved and replaced with identifiers (qualified names), and SDL features not covered by the dynamic semantics are transformed to equivalent SDL features. For example, remote procedure calls are transformed to local procedure calls and signal exchange.

   Because of dependencies between transformations, some transformations must be completed for the entire specification before other transformations can be applied. Therefore, transformations are divided into 17 groups. Transformations in a group must be completed for the entire specification before transformations of the subsequent group are applied.

   On the abstract syntax AS0, well-formedness conditions (WFC 0) that must be satisfied by a valid SDL specification are defined. These well-formedness conditions are checked after parsing, and after the completion of all transformations in a group.

3. The *mapping* step defines the relationship between abstract syntax AS0 and AS1. After the transformations on AS0 are completed, the mapping to AS1 is

straightforward. On AS1, another set of well-formedness conditions (WFC 1) that a valid SDL specification must satisfy is defined.

2.4.2 Dynamic Semantics

The dynamic semantics of SDL [42] defines the behaviour of syntactically correct, well-formed SDL specifications, given by abstract syntax AS1. The possible computations of an SDL specification are described in the form of a partially-ordered run of a DASM. Due to the complex, distributed nature of SDL, a virtual machine approach is used for the formal semantics of SDL.

Based on *distributed real-time ASMs* (DASMs, see Section 2.3.4), which support concurrency, asynchronous computation, and time, an *SDL virtual machine* (SVM) is defined. The SVM consists of an *SDL abstract machine* (SAM), the logical hardware for the execution of SDL specifications, plus operating system functionality.

SDL Abstract Machine (SAM)

The SAM defines the logical hardware for the execution of SDL specifications, and consists of the following parts:

- *Behaviour primitives* form the *instruction set* of the SDL abstract machine. Instructions include setting timers, procedure calls, and entering and leaving state nodes.

```
1 Set =def TimeLabel ×Timer ×ValueLabel* ×ContinueLabel
2
3 EvalSet(a : Set) ≡
4   SetTimer(a.s-Timer, values(a.s-ValueLabel-seq, Self), semvalueReal(value(a.s-
        TimeLabel, Self)))
5   Self.currentLabel := a.s-ContinueLabel
```

Listing 2.2: Behaviour primitive for setting timers

Listing 2.2 shows the behaviour primitive for setting timers, consisting of a domain Set (line 1) and a rule macro EvalSet (line 3). Set is a tuple consisting of the type of the timer, and several labels referring to the time the timer is set to, timer parameters, and the next instruction to be evaluated. EvalSet evaluates instructions of type Set, by calling rule macro SetTimer, and setting the abstract program counter *currentLabel* to the next instruction.

- *SAM agents* define several agent types: SDLAgent and SDLAgentSet for SDL agents and their containers, and Link for SDL channels.
- The *signal flow model* provides communication resources (links, gates and signal queues) for signal exchange between agents.

SDL Virtual Machine (SVM)

The SVM provides operating system functionality on top of the SAM, defining programs for SDL agents, SDL agent sets, and links. The SVM defines transition selection, firing of actions of a transition, a communication system, and a runtime system for the execution of SAM agents. Transition selection freezes the state of inport of an agent, and iterates for each transition type (priority input, input, ...) through the signals in the queue and the current states of the agent, until a fireable transition is found. Transition firing executes the actions of the transition by calling the behaviour primitives of the SAM. The runtime system manages execution rights, and enables either parallel or interleaving execution of agents, depending on the agent type.

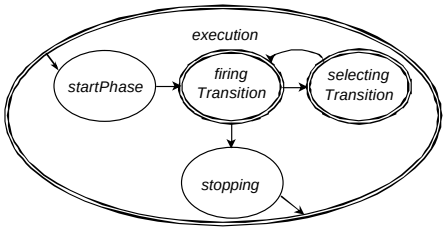


Figure 2.5: Agent mode *execution* (level 2) and submodes [42]

For transition selection, firing, and agent execution, the SVM defines an *agent control block*, storing context information for each SDL agent, such as the state of inport at the start of transition selection, the state and transition currently being checked, and the mode of the agent.

Each SDL agent and SDL agent set has a hierarchy of agent modes. An agent mode models an activity phase of an agent. On the top level (level 1), an agent is in agent mode *initialisation* or *execution*. These high level activities are split up in subactivities, represented by agent modes on a lower level (levels 2 - 5). Figure 2.5 shows the submodes of agent mode *execution*, and the order in which these activities are executed. An agent entering mode *execution* first enters submode *startPhase*, then alternates between modes *firingTransition* and *selectingTransition*. These two modes contain further subactivities. From mode *firingTransition*, the agent can enter agent mode *stopping*, after which agent mode *execution* is left. The agent mode hierarchy consists of up to five levels, stored in functions *agentMode1* to *agentMode5*.

Compilation Function

The compilation function defines the compilation of SDL transitions from the abstract syntax AS1 to an ASM representation - a set of behaviour primitives. Each transition is compiled into a set of actions and an action flow. The compilation generates a label for each action - via function *uniqueLabel* - as an abstract program counter, and as a location to store the results of expressions and procedure calls.

Listing 2.3 shows the compilation function for the *Assignment* node of the AS1, assigning an expression *expr* to a variable *id*. Function *compileExpr* compiles *expr* to

an ASM representation. Label *uniqueLabel*(*a*,1) is passed as the next label to process after the expression. The compilation function also generates a behaviour primitive of type TASK, with label *uniqueLabel*(*a*,1), variable *id*, the label storing the result of the expression *uniqueLabel*(*expr*,1), and the next label to be processed, a parameter of the compilation function.

```
| a=Assignment(id, expr) =>
    compileExpr(expr, uniqueLabel(a,1)) ∪
    {mk-PRIMITIVE(uniqueLabel(a,1), mk-TASK(id, uniqueLabel(expr,1), next) )}
```

Listing 2.3: Compilation of *Assignment* node to behaviour primitive TASK

2.5 Formal Semantics of UML

UML is a language without a standardised formal semantics definition. While the abstract syntax of UML is defined in a semi-formal fashion (using meta-models), the semantics is defined for fragments of the language in an informal fashion. Steps towards a more complete language semantics were taken with UML 2.0, the precise meaning of UML subsets is only given by tool implementations.

In the research community, a substantial body of work is concerned with the formalisation of subsets of UML. Several works exist for the formalisation of specific UML diagrams, in particular Behaviour Diagrams. For example, Börger et al. have provided a formal semantics for State Machine [5] and Activity Diagrams [4] using Abstract State Machines. These works provide a formal semantics for the view of an UML model provided by the specific diagram, but do not address the semantics of the complete UML model - for example, the interdependencies of state machines and class diagrams concerning object creation and inter-object communication.

Other approaches define formal semantics for subsets of the UML model. In [49], static semantics derived from the UML meta-model is combined with dynamic semantics using Abstract State Machines. The dynamic semantics formalises the UML action language, concurrency and communication (partially based on the signal flow model of SDL). In [11], a subset of UML - called *krtUML* - covering behavioural modelling entities of UML used for real-time applications is formalised using symbolic transition systems. The ITU defines the UML Profile for SDL [39] - scheduled for standardisation in 2007 -, giving a precise semantics to a large subset of UML by mapping modelling entities to SDL entities with compatible semantics. While SDL is a formal language, the mappings defined in the UML Profile for SDL are defined informally. We introduce an approach for the formalisation of the profile in [22] (see Section 3.6.4 in this thesis).

Work of the pUML Group [1] focuses on the precise definition of languages, UML in particular, using an object-oriented meta-modelling language (MML) [9, 10]. This language incorporates a subset of the UML, defining UML in itself using a precise meta-modelling approach.

3 Harmonisation of Modelling Languages

This chapter is concerned with the harmonisation of modelling languages SDL and UML, that is, the integration of these languages taking into account their underlying similarities and differences. As a first step, we provide mappings between the abstract syntax representations of SDL (abstract grammars) and UML (meta-models) (Section 3.3). Then, using abstract grammars as abstract syntax representation, we examine how common constructs of SDL and UML are reflected in common parts of the abstract syntax of both languages (Section 3.4). We analyse semantic similarities and differences of state machines in SDL and UML, which have a comprehensive semantics in both languages (Section 3.5). Section 3.6 describes the approach of the ITU to integrate SDL as a profile of UML. We contribute an approach to the formalisation of this profile, giving a formal semantics to a subset of UML, and enabling the automatic generation of tool support.

3.1 Motivation

With SDL-2000, several important steps towards its future harmonisation with UML have been made. For instance, classes and associations including aggregation, composition, and specialisation were added to the language. Furthermore, composite states that are similar to submachines in UML statecharts were incorporated. In turn, UML 2.0 introduced structured classes, which extend classes by an internal structure consisting of nested structured classes, ports and connectors. This makes it possible to model architectural aspects of systems in a fashion similar to SDL.

First attempts to harmonise UML and SDL have already been made for previous language versions. The old Z.109 standard [33] defines a subset of UML 1.3 [50] that has a mapping to SDL-2000. The UML subset is used in combination with SDL, with the semantics based on SDL-2000. In [62], Selic and Rumbaugh define a transformation from SDL-92 to UML 1.3 extended with the Rational Rose real-time profile.

Ultimately, these efforts are directed towards an integration of both languages and the corresponding notations. However, at the time being, UML and SDL still deviate in many ways, making it hard to see whether and when integration might actually be achieved. Differences range from pure syntactic aspects to semantic concepts, resulting from the origin of the languages. Also, it is not clear whether different views of a system even if expressed in notations belonging to the same family are consistent. Furthermore, while SDL is a complete language with formal semantics, UML is a language framework with a formal syntax, but many semantic variation points.

These similarities and differences make UML and SDL suitable objects for studying the harmonisation of modelling languages.

In order to derive a common syntactic and semantic basis, the existing language definitions of UML and SDL should be taken as a starting point. In this chapter, we present the results of analysing several corresponding excerpts of UML and SDL, compare them, and derive a common subset. On the syntactical level, this is done by defining conceptually sound and well-founded mappings from meta-models (used to define the abstract syntax of UML) to abstract grammars (used by SDL) and vice versa, and by extracting common production rules. On the semantic level, we compare statemachines of UML with process graphs of SDL, and outline their similarities and differences. Finally, we describe the approach of the ITU to integrate UML and SDL via the UML profile mechanism as documented in the Z.109 standard [39], and propose a formalisation for this approach.

3.2 Abstract Syntax Representations

3.2.1 The Abstract Grammar Approach

The definition of a language consists of its syntax and semantics. The concrete syntax of a language includes separators and other constructs needed for parsing the language. The abstract syntax omits these details and contains only the elements relevant for the definition of the semantics. Both the concrete and the abstract syntax of a language can be defined in terms of a grammar, consisting of a set of production rules that define the syntactically correct sentences.

For SDL, a concrete (textual and graphical) syntax and two abstract syntaxes, AS0 and AS1, are defined. The AS0 is obtained from the concrete syntax by omitting details such as separators and lexical rules. Otherwise, it is very similar to the concrete syntax of SDL. The abstract syntax AS1 is obtained from the abstract syntax AS0 through a step of transformations followed by a mapping. During the transformation, several concepts are translated into core concepts of SDL as described in the standard.

The abstract syntax of SDL is described in terms of a context-free abstract grammar (see [35], Section 5.4.1), based on the meta-language of the Vienna Development Method (VDM) [2, 44]. A production rule of the abstract grammar has the form:

`Name ::=(::) Rule` or `Name ::=(=) Rule`

We call production rules of the form `Name ::=(::) Rule` *concatenations*, and production rules of the form `Name ::=(=) Rule` *synonyms*. We refer to the name as the left hand side, and to the rule as the right hand side of the production rule. A rule consists of one or more *terms*, separated by vertical bars ("|"). A term is a sequence of *items*. Items are names, modified names (see below), and bracketed terms (terms with more than one item).

A name as an item refers to the set of objects defined by its rule. For each name N, the modified names N-**set** (the powerset of N), N* (the set of sequences of N), N+ (the set of non-empty sequences of N) and [N] (the union of N and "undefined") are

defined. A bracketed term is a composite, tree-like object. For each bracketed term, a **mk-**α operator with a unique name α is introduced. If the bracketed term is the only term on the right hand side of a concatenation, α is the name on the left hand side of the concatenation. Table 3.1 (corresponding to Figure 4.1 in [44]) shows the rule forms used for the abstract grammar of SDL, and the sets of objects they describe.

Rule	**Sets of Objects**
A ::=(=) B	B
A ::=(=) B | C	B ∪ C
A ::=(=) $\mathbf{t_1}$ | $\mathbf{t_2}$ | ...	{ $\mathbf{t_1}$, $\mathbf{t_2}$, ... } for terminals $\mathbf{t_1}$, $\mathbf{t_2}$, ...
A ::=(::) B	{ **mk-**A(b) | b ∈ B }
A ::=(::) B C	{ **mk-**A(b,c) | b ∈ B, c ∈ C }
A ::=(::) B (C | D)	same as A ::=(::) B X; X ::=(=) C | D

Table 3.1: Rules and their meanings

For the mapping described in Section 3.3, we assume a normal form of the abstract grammar, where concatenations have no alternatives ("|") on the right hand side. The SDL abstract grammar can be easily transformed into this normal form by introducing new synonyms for these alternatives (see Table 3.1).

Below, an excerpt of the AS1, the concatenation State-node and the synonym Nextstate-node, is shown. State nodes are composite objects consisting of a state name, a save signalset, and sets of input nodes, spontaneous transitions, continuous signals, and connect nodes. Optionally, a state node can have a composite state type identifier (in that case, the state represents a composite state of the respective type) and an associated exception handler.

State-node ::=(::) State-name
[On-exception]
Save-signalset
Input-node-**set**
Spontaneous-transition-**set**
Continuous-signal-**set**
Connect-node-**set**
[Composite-state-type-identifier]

A Nextstate-node is a synonym for either a Named-nextstate or a Dash-nextstate, a transition terminator containing either a state name or a dash symbol.

Nextstate-node ::=(=) Named-nextstate | Dash-nextstate

3.2.2 The Metamodel Approach

A meta-model [51, 53, 52] is a model used to define a language for the specification of models. In UML, this meta-model approach is used to define the language syntax. In

particular, the abstract syntax of the language is defined using UML class diagrams. This approach is reflective, since class diagrams are UML models, and therefore described in terms of themselves. On top of the model and the meta-model, more layers can exist (meta-meta-models, etc.). UML uses a four layer meta-model structure [52]: user objects (M0), model (M1), meta-model (M2), and meta-meta-model (M3). Every element in a layer is an instance of an element of the direct superordinate layer.

UML class diagrams used for the description of the abstract syntax comprise packages, classes, attributes, associations, and specialisation. Classes in the UML meta-model describe *language elements*. An occurrence of the language element in the model (M1) is an instance of the meta-model class. Classes in the meta-model can be parameterised by attributes. Attributes describe properties of the language element described by a class. Composition between meta-model classes describes that a language element contains another. General associations relate language elements, for example, a transition with a trigger. The meta-model uses packages, abstract classes, and specialisation to structure the abstract syntax.

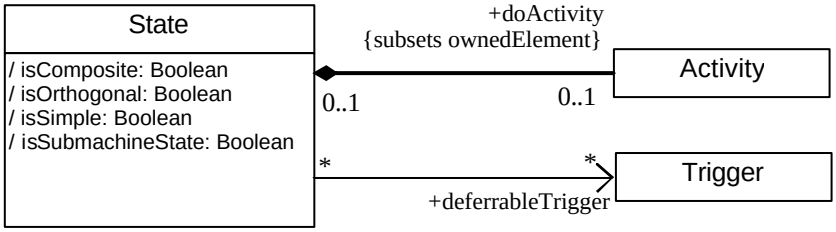


Figure 3.1: Excerpt of the abstract syntax of states

Fig. 3.1 shows an excerpt from the abstract syntax of statemachine states. States are described by a class with four attributes, describing the type of the state. These attributes are derived ('/'), meaning that their value is derived from other information in the meta-model. A State contains up to one Activity in the role of a 'doActivity'. It is also associated with an arbitrary number of triggers in the role of deferrable triggers. The composition between State and Activity is a subset of the association 'ownedElement'.

3.3 Mapping Between Meta-models and Abstract Grammars

As a contribution, we define precise, sound mappings from meta-models to abstract grammars, and vice versa (this work was submitted and accepted to the Forte'04 conference [24]). As it turns out, not every element of UML meta-models can be mapped. Also, several meta-model elements may have the same representation in the abstract grammar. Therefore, the mapping is not completely reversible. However, it is possible to map every element of an abstract grammar to a meta-model representation.

In Section 3.4, these mappings will be applied to UML and SDL to extract a common syntactical basis.

In the following sections, we define mappings for elements of the meta-object facility (MOF) [51, 53] relevant for the definition of the UML meta-model. These elements are classes and enumerations, attributes, associations, multiplicity, and specialisation.

3.3.1 Classes and Enumerations

map(MM): A *concrete class* of the meta-model represents a language element. For example, the meta-model class State represents all state descriptions in a UML statemachine. In an abstract grammar, a language element (non-terminal) is represented by a specific production rule, namely a concatenation. Therefore, a concrete class in the meta-model is mapped to a concatenation of the abstract grammar. The name of the non-terminal is derived from the class name and the package structure of the meta-model (see below). The right hand side of the concatenation is derived from the class definition (attributes) and context (associations) as defined below (see 3.3.2, 3.3.3).

An *abstract class* of the meta-model describes properties that are common to its subclasses. For example, the meta-model class *Vertex* describes properties that are common to states and pseudo-states (initial states, ...). Since an abstract class can not be instantiated, it does not represent a language element. Therefore, no concatenation is used in the mapping. Instead, we map an abstract class of the meta-model to another kind of production rule, namely a synonym, of the abstract grammar. In an abstract grammar, a synonym replaces the element on its left hand side with an element of the right hand side. This is similar to abstract classes in the meta-model, which must be replaced by one of their concrete subclasses. The name of the non-terminal is selected as in the case of a concrete class. The right hand side is derived from the context (specialisation) as described below (see 3.3.5).

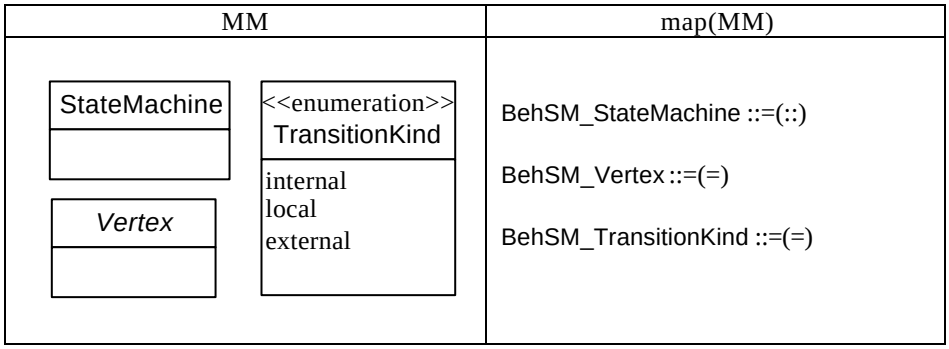
An *enumeration* in the meta-model is a set of values used to parameterise meta-model classes. For example, the meta-model class Pseudostate describes different language elements (entry point, exit point, ...) of the model depending on the value of the attribute 'kind' of the enumeration type PseudostateKind. Enumerations do not directly describe language elements. Therefore, as in the case of abstract classes, no concatenation is used in the mapping. Instead, enumerations are also mapped to synonyms of the abstract grammar. This production rule replaces the enumeration by one of its values.

The name of a non-terminal introduced by one of the mappings described above is the qualified name of the class or enumeration. The qualified name is a sequence of the packages the class or enumeration is contained in (from outermost to innermost) and the name of the class or enumeration, each separated by underscores. For example, Kernel_Element is the name of the non-terminal introduced by the class Element in the package Kernel. The qualified name is used in order to avoid name clashes between equally named classes in different packages.

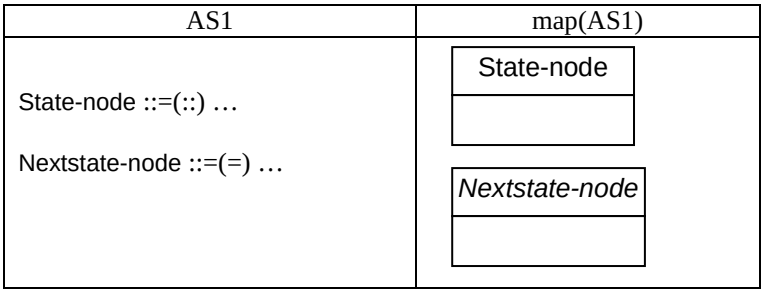
Example: The following example comes from the meta-model of UML state machines. It describes two classes, an abstract class *Vertex* and a concrete class StateMachine. Furthermore, there is an enumeration TransitionKind. All of these elements are

contained in the package BehaviorStatemachines (not shown) that we will shortly refer to as BehSM.

StateMachine is a concrete class, and is therefore mapped to a concatenation. The name BehSM_StateMachine comes from the package structure and the name of the class. The abstract class *Vertex* and the enumeration TransitionKind are mapped to synonyms.



map(AG): As mentioned when defining the mapping from meta-models to abstract grammars, concrete classes and *concatenations* both represent language elements of the model. Therefore, concatenations of the abstract grammar are mapped to concrete classes in the meta-model. The name of the concrete class is derived from the production rule (see below).



A *synonym* of the abstract grammar represents a language element that does not appear in the model, but stands for other language elements. For example, a Data-type-definition in the SDL abstract grammar is a synonym for a Value-data-type-definition, an Object-data-type-definition or an Interface-definition. This is a similar concept to abstract classes in the meta-model, which we have mapped to synonyms in the abstract grammar. However, it is also similar to an enumeration, where the enumeration stands for one of its values. Therefore, we map a synonym in the abstract grammar either to an abstract class or an enumeration. The exact mapping depends on the right hand

side of the synonym (see 3.3.2, 3.3.3). The name of the class or enumeration is the name of the non-terminal on the left hand side of the production rule.

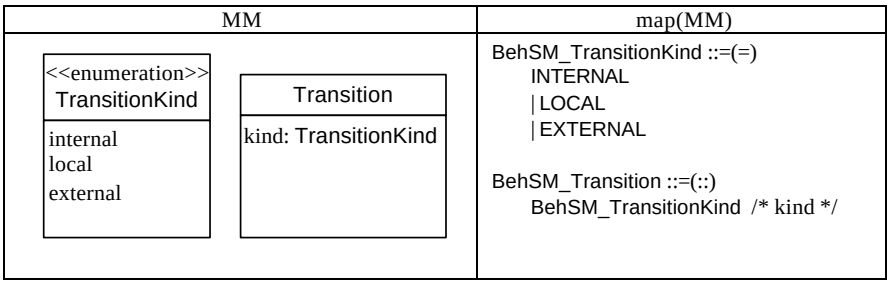
Example: In the abstract grammar of SDL, the production rule for the non-terminal State-node is a concatenation. It is therefore mapped to a concrete class. The production rule for Nextstate-node is a synonym. It would therefore either map to an abstract class or an enumeration. Because of the right hand side of the production rule, which we do not treat at this point, it is mapped to an abstract class.

3.3.2 Attributes

map(MM): In the meta-model, *attributes* of a class represent properties of a language element. For example, the attribute 'kind' of the meta-model class Transition describes if the transition is internal, local or external. In an abstract syntax tree, an attribute is represented as a sub-node of the non-terminal and corresponds to a class. We have mapped concrete classes to concatenations of the abstract grammar. Therefore, an attribute of a concrete class is mapped to either a terminal or a non-terminal on the right hand side of the concatenation. We map attributes to terminals if they do not need to be refined any further, and to non-terminals otherwise. If the type of the attribute is an enumeration type, we always map the attribute to a non-terminal, since we have mapped enumerations to synonyms and non-terminals. The name of the terminal is the name of the type (for example, Boolean). The name of the non-terminal is derived from the name of the enumeration and the package structure, as defined in Section 3.3.1.

Attributes that are marked as *derived* carry no additional information and can be omitted. For example, the attribute 'isComposite' of State can be derived from the number of associated regions. If they are not omitted, additional constraints are needed to define the dependencies between the original and the derived attributes. Default values of attributes can not be mapped to the abstract grammar.

Elements of an enumeration represent values of the enumeration type. In an abstract grammar, a value is represented by a terminal. Therefore, enumeration elements are mapped to terminals of the abstract grammar. The name of the terminal is the name of the enumeration element. An enumeration is mapped to a synonym of the abstract grammar. Therefore, we map the terminals to the right hand side of the synonym corresponding to the enumeration.

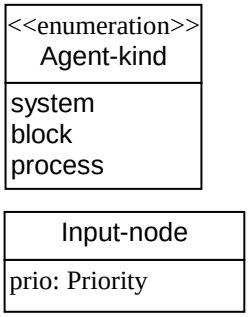


Example: The example above (again from BehaviorStatemachines) contains a concrete class Transition and the enumeration TransitionKind. The classes are mapped as described in the previous section. The attribute 'kind' of Transition is an element on the right hand side of BehSM_Transition. In this special case ('kind' is an enumeration), it is a non-terminal that refers to the mapping of the enumeration TransitionKind. The name of the attribute is appended as a comment. The enumeration literals of TransitionKind appear as alternatives of terminals on the right hand side of the production rule, written in all caps for better distinction.

map(AG): A *terminal* on the right hand side of a concatenation represents a property of the language element. In the meta-model, an attribute represents a property of a language element. The terminal is therefore mapped to an attribute of the concrete class corresponding to the concatenation. The type of the attribute is the name of the terminal. The name of the terminal can be chosen arbitrarily as long as it does not conflict with other attribute names of the class.

A synonym with only terminals on the right hand side represents an enumeration of values. For example, the synonym Agent-kind of the SDL abstract grammar is an enumeration of the values SYSTEM, BLOCK and PROCESS. Therefore, the terminals are mapped to enumeration values of the enumeration corresponding to the synonym.

Example: Agent-kind is a synonym and has only terminals on the right hand side. Therefore, it is mapped to an enumeration. The alternatives on the right hand side (SYSTEM, BLOCK and PROCESS) are mapped to enumeration literals of the corresponding enumeration. The terminal PRIORITY on the right hand side of Input-node is mapped to an attribute of the corresponding concrete class.

AS1	map(AS1)
Agent-kind ::=(=) SYSTEM | BLOCK | PROCESS Input-node ::=(::) PRIORITY ...	

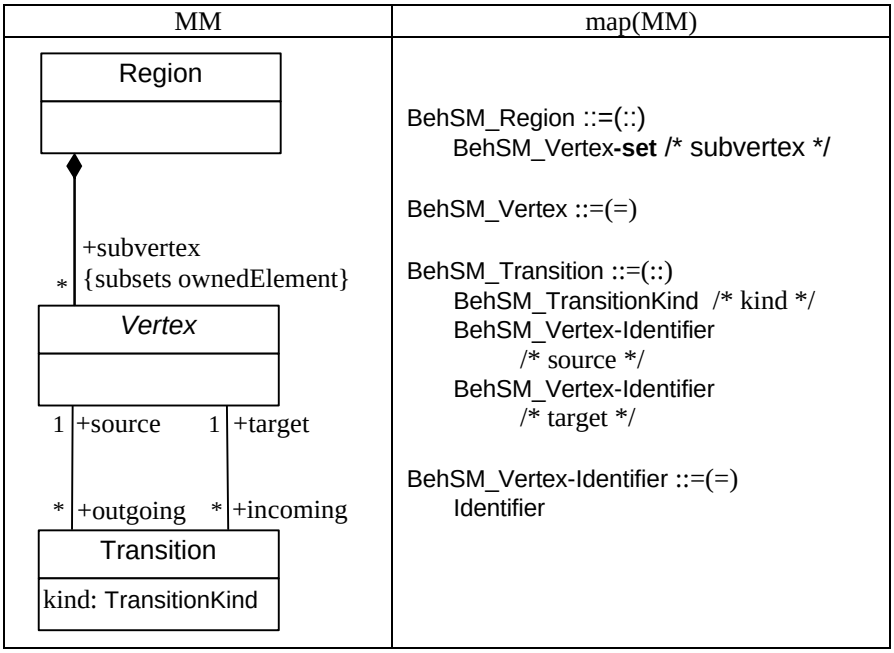
3.3.3 Associations

map(MM): An *aggregation* or *composition* between two classes means that one language element contains or is made up of other language elements. For example, a Region in a statechart contains vertices and transitions. In the same way, a node in an abstract syntax tree can have sub-nodes. For example, a State-transition-graph of

the SDL abstract grammar has a set of State-nodes as sub-nodes. Therefore, we map aggregation and composition to the abstract grammar so that the definition of the aggregated class is a sub-node of the aggregating class. This is achieved by adding the non-terminal corresponding to the aggregated class on the right hand side of the concatenation corresponding to the aggregating concrete class.

A *general association* between two classes is an association between language elements, in which the elements play a certain role. For example, a State is associated with a number of triggers, the triggers playing the role of deferrable triggers. In the SDL abstract grammar, two language elements are associated by identifiers. For example, an Input-node is associated with a Signal by a Signal-identifier on the right hand side of the concatenation corresponding to the Input-node. Therefore, a directed general association is mapped to an identifier on the right hand side of the concatenation corresponding to the concrete class the association originates from. An undirected general association is split into two directed general associations.

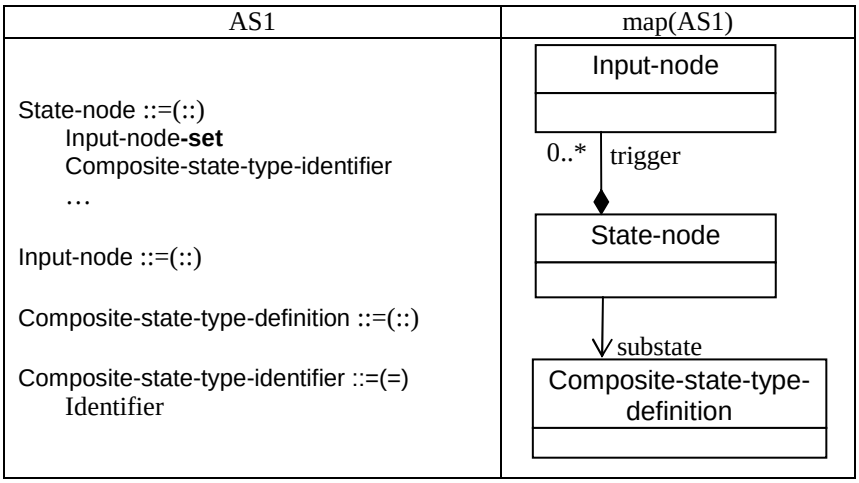
An association with the *union* property is the union of the associations that subset it. This is expressed by the property *subsets*. As in the case of derived attributes, associations with the union property are not mapped to the abstract grammar.



Example: The example above shows the abstract class *Vertex* and the concrete classes Transition and Region. Region is composed of a set of *Vertex*es called subvertex. This composition is a subset of the association 'ownedElement' between two elements. In the abstract syntax tree, BehSM_Region thus has BehSM_Vertex-**set** on the right hand side, with the name appended as a comment. Between *Vertex* and Transition

there are two bidirectional associations, which are split into two unidirectional associations each. Attributes and associations of an abstract class are not mapped to the corresponding synonym in the abstract syntax tree, since an abstract class is not a synonym for one of its attributes or associations. Instead, they are copied into the respective subclasses, as described in Section 3.3.5. In this example, *Vertex* has no subclasses. Therefore, we only have to map the two general associations 'source' and 'target'. To distinguish between general association and composition, an association is mapped to an identifier (in this case, BehSM_Vertex-Identifier) on the right hand side of the corresponding production rule. How the identifier looks like is not further specified. It could be a qualified name like in the case of SDL.

map(AG): *Non-terminals* on the right hand side of a *concatenation* can stand for an enumeration or a class in the meta-model. In case they represent an enumeration, they represent an attribute of the class (see Section 3.3.2). In case they represent a class, this class is a sub-node of the class corresponding to the concatenation. This is similar to a class in the meta-model that is composed of other classes. Therefore, in this case we map a non-terminal on the right hand side of a concatenation to a composition in the meta-model. The composing class is the class corresponding to the concatenation; the composed class is the class corresponding to the non-terminal on the right hand side. The role of the classes can be chosen arbitrarily.



An *identifier* on the right hand side of a concatenation identifies a language element that is associated with the language element described by the concatenation. For example, in the SDL abstract grammar, an Input-node is associated with a Signal by a Signal-identifier. Therefore, we map an identifier on the right hand side of a concatenation to a directed general association in the meta-model. The source of the association is the concrete class corresponding to the concatenation, according to the mapping in Section 3.3.1. The target is the concrete class corresponding to the language element referenced by the identifier. The roles of the classes can be chosen

arbitrarily.

Example: The non-terminal Input-node on the right hand side of State-node is mapped to a composition of Input-node in State-node in the meta-model. The set-suffix is mapped to the multiplicity '0..*', as defined in Section 3.3.4. Composite-state-type-identifier is an identifier referring to a Composite-state-type-definition (in the abstract grammar of SDL, identifier/definition pairs usually have the same name with a -identifier/-definition suffix, e.g. Signal-identifier and Signal-definition). This is mapped to a general association between the two corresponding classes in the meta-model.

3.3.4 Multiplicity

In UML, multiplicities consist of a lower bound and an optional upper bound, which can be infinite. The property *ordered* expresses that there is a linear order for the elements. The property *unique* expresses that no element appears more than once. In the abstract grammar, an optional element is enclosed by square brackets. A possibly empty list of elements is marked by a '*' behind the element, a non empty list by a '+'. A set of distinct elements is marked by the suffix '**-set**'.

MM	**AG**
0..1	*[Name]*
0..n, 1 < n	(as 0..*)
*0..**	*Name **
0.. {unique}*	*Name**-set***
1	*Name*
1..n, 1 < n	(as 1..*)
*1..**	*Name +*
1..* {unique}	(as 0..* {unique})
n	(as 0..*)
n..m, 1 < n < m	(as 1..*)
n..*, 1 < n	(as 1..*)

Table 3.2: Mapping of Multiplicities

Table 3.2 shows the mapping of multiplicities between meta-model and abstract grammar. If we use lists in the abstract grammar, the elements are ordered and not necessarily unique. If we use sets, they are not ordered and unique. Therefore, we can only map one of the properties to the abstract grammar. In this case, the property *ordered* is omitted from the mapping.

3.3.5 Specialisation

In the UML meta-model, abstract classes and specialisation are used frequently to capture common aspects of different classes, and as part of a meta-language core reused in several standards (see UML: Infrastructure [52]). For the abstract syntax, abstract

classes are not directly interesting, since they can not be instantiated and therefore do not appear in a model, except through their subclasses. Nonetheless we map them to the abstract grammar, to preserve as much of the structure of the meta-model as possible.

map(MM): We have to map specialisation to the abstract grammar, and the fact that a specializing class inherits properties of the specialised class. The easiest way to do this is to copy these properties into the specializing classes before the mapping. This has the advantage that redefinition of properties is easy to realise. They are not copied to subclasses that overwrite them.

This is done as follows:

Step 1) For every class that has subclasses, copy all attributes of the class and all associations that originate from this class to each of its direct subclasses.

a) An attribute is only copied to a subclass if no attribute of the same name already exists, i.e., if the attribute is not redefined.

b) An association is only copied to a subclass if it is not redefined in the subclass.

Step 2) Repeat Step 1 for all subclasses that have new attributes and associations after the last execution of Step 1.

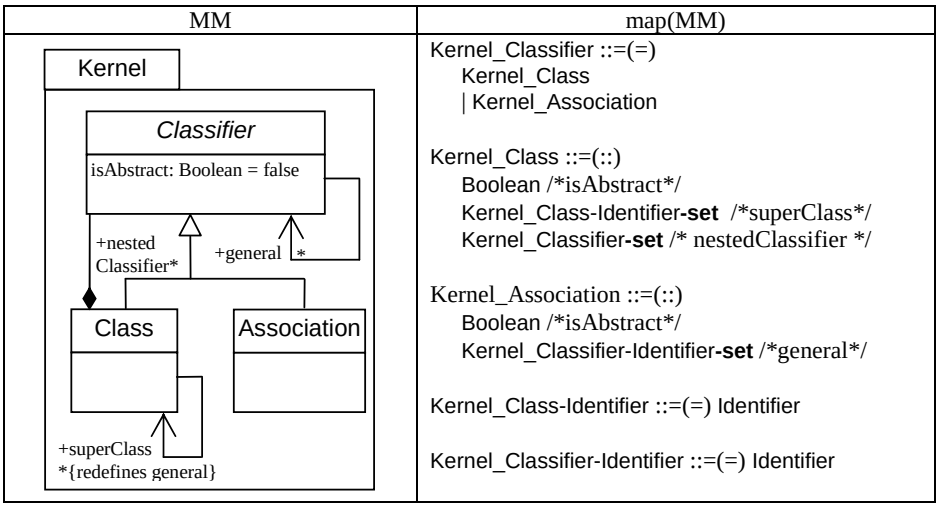
In the meta-model, an abstract class can take part in an association. In the model, an instance of a concrete class that specialises the abstract class takes part in the association instead. For example, a *Vertex* is associated with Transition as the source of transitions. In the model, the source of transitions is either a State or a Pseudostate. In the abstract grammar, we can express this using a synonym. We have already mapped an abstract class to a non-terminal and a synonym (see Section 3.3.1). To map the specialisation to the abstract grammar, we add the non-terminals corresponding to the direct sub-classes of the abstract class to the right hand side of the synonym. This means that every occurrence of the non-terminal (the abstract class) is replaced by a non-terminal (one of the subclasses) in the abstract syntax tree.

To map specialisation to the abstract grammar, we need synonyms. On the other hand, a concrete class can have subclasses, but is mapped to a concatenation (see Section 3.3.1). In this case, we transform the meta-model before we perform the mapping. The concrete class with subclasses is replaced by an abstract class of the same name. The concrete class is renamed, e.g. by adding a special prefix, and added as a subclass of the new abstract class. The subclasses of the concrete class are now subclasses of the new abstract class and the mapping can be performed. However, we still have to copy the attributes of the concrete class to its former subclasses, as described above.

Example: The following example is taken from the package Kernel and covers classifiers, classes and associations. *Classifier* is an abstract class with the attribute 'isAbstract'. A classifier can be generalised by another classifier, described by the association named 'general'. Concrete subclasses of *Classifier* are Class and Association. The association 'superClass' between two classes redefines the association 'general'.

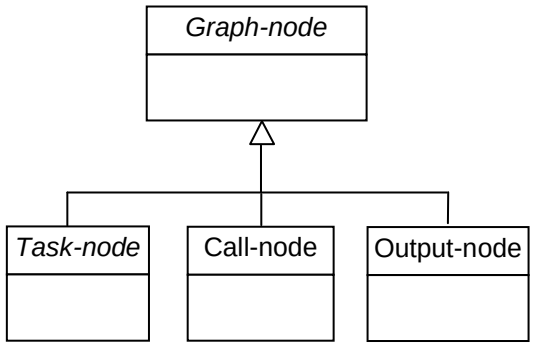
Before mapping to the abstract grammar, we have to copy the attributes and associations of the abstract class *Classifier* to its subclasses. The attribute 'isAbstract' is copied to the classes Class and Association, since no attribute of the same name exists. A new association 'general' from Association to *Classifier* is added. The association 'superClass' redefines 'general', therefore no new association is added to Class.

The abstract class *Classifier* is mapped to a synonym. Class and Association are direct subclasses of *Classifier*; therefore, we add the non-terminals corresponding to these classes on the right hand side of the synonym.



map(AG): *Non-terminals* of the abstract grammar represent language elements. A synonym of the abstract grammar with non-terminals on the right hand side replaces a language element by another. For example, a Return-node in the abstract grammar of SDL is replaced by an Action-return-node, a Value-return-node or a Named-return-node. We map synonyms to abstract classes in the meta-model. Abstract classes can not be instantiated, but can have instances through their subclasses. Therefore, we map a synonym with non-terminals on the right hand side in the abstract grammar to a specialisation relationship. The specialised class is the class corresponding to the non-terminal on the left hand side. The specialising classes are the classes corresponding to the non-terminals on the right hand side.

Example: The synonyms for Graph-node and Task-node are mapped to abstract classes. Output-node and Call-node are concatenations and therefore concrete classes. For the non-terminals Task-node, Output-node and Call-node on the right hand side of the synonym, a specialisation relationship is added between *Graph-node* and Task-node, Call-node, and Output-node.

AS1	map(AS1)
Graph-node ::=(=) Task-node | Output-node | Call-node | … Task-node ::=(=) Output-node ::=(::) Call-node ::=(::)	

3.3.6 Meta-Model Approach vs. Abstract Grammar Approach

From the discussion so far, it seems that the meta-model approach to defining an abstract syntax is more expressive than the (context free) grammar approach. As a consequence, the mapping from the SDL abstract grammar to a meta-model is completely reversible. However, this is not the case for the mapping from the UML meta-model to an abstract grammar. Several elements of the UML meta-model can not be expressed in the abstract grammar, using our mapping, including the following:

- Visibility information, default values and derived attributes are not expressible in a context free grammar.
- Associations with the property union, as well as the property subsets are not expressible.
- When an attribute or an association of an abstract class is redefined in a subclass, it cannot be properly reproduced.
- Several multiplicities have identical mappings, and cannot be properly reproduced with the mapping described in the previous section. The property ordered (alternatively, the property unique, see Section 3.4) can not be mapped to a context free grammar.

In consequence, the meta-model approach seems to be preferable as a basis for the harmonisation of UML and SDL. It covers and extends the expressiveness of abstract grammars, and thus seems to be the right choice. However, the same expressiveness can be achieved with the given mapping by adding static conditions as in the static semantics of SDL [41], leading to a context-sensitive grammar. Therefore, the main advantage of the meta-model approach is the ability to structure the abstract syntax, using packages and specialisation. On the other hand, the flat meta-model structure

obtained by the meta-model transformation described in Section 3.3.5 increases readability, by putting the definition of all attributes and associations of a class in one place. When it comes to implementing a language by providing tool support, abstract grammars have an advantage due to their precise semantics, and the existence of sophisticated compiler generation tools. With the mapping defined above, such an abstract grammar can be systematically derived.

3.4 Syntactic Harmonisation of SDL and UML

Translating the meta-model of UML 2.0 into an abstract grammar supports the comparison of the abstract syntax of UML 2.0 and SDL-2000. In particular, it enables us to examine how the common constructs of SDL and UML are reflected in common parts of the abstract syntax of both languages, and to extract the common abstract grammar subset.

As it has turned out, some information of the meta-model is lost when it is mapped to an abstract grammar (see Section 3.3.6). However, the information lost is not important for the extraction, because it is not present in the abstract syntax of SDL.

Instead of mapping the UML meta-model to an abstract grammar, we could apply the mapping from the SDL abstract grammar to a meta-model. This way, no information would be lost, as the meta-model is more expressive. However, the extraction process would not benefit from this choice. Even worse, the extraction would be harder, since the UML meta-model defines a large number of abstract classes with attributes and associations, which would not show up in the SDL meta-model. It would be necessary to either copy the attributes of abstract classes to their subclasses in the UML meta-model (as described in Section 3.3.5), or to identify common attributes and associations, and shift them to super-classes in the SDL meta-model.

To relate language elements of SDL-2000 and UML 2.0 on a syntactical level, substantial knowledge of both languages is required. In particular, it is necessary to take the semantics of language elements into account. E.g., we need knowledge of the semantics of the language elements to relate the Package-name of a Package-definition in the abstract syntax of SDL with the String of a structured class in the abstract syntax of UML. Also, it can be expected that for some of the common constructs, the abstract syntax will be different, although the semantics is the same. In some cases, there might even be a common abstract syntax, although the semantics is different.

To extract the common abstract syntax of the two languages, we take the production rules for language elements that are similar in UML and SDL, e.g. packages, as a starting point, and compare their right hand sides. For corresponding elements in both sets of production rules that represent similar concepts, the production rules for these elements are compared. If they overlap, we can relate the two elements with each other and include them in the common abstract syntax. We exemplify the extraction with high level language elements, namely packages and agent-types/classes, as well as language elements with a finer granularity.

Packages

Both SDL and UML have a concept of packages for grouping and reuse of elements of the specification (see Table 3.3). Both support the nesting of packages (2). The abstract syntax of UML describes the contents of a package as a set of PackageableElements, a synonym for all elements that can be contained in a package. SDL describes sets of the elements that can be contained in a package, e.g. Signal-definition-**set**. Common packageable elements in SDL and UML are agents/classes (4), signals (3) and composite states/statemachines (5).

SDL-2000 (AS1)	UML 2.0 (derived AS)
Package-definition ::=(::)	Kernel_Package ::=(::)
1 Package-name	1 [String]
2 Package-definition-**set**	2 Kernel_Package-**set**
Data-type-definition-**set**	Kernel_PackageableElement-**set**
Syntype-definition-**set**	Kernel_PackageMerge-**set**
3 Signal-definition-**set**	Kernel_ElementImport-**set**
Exception-definition-**set**	Kernel_PackageImport-**set**
4 Agent-type-definition-**set**	Kernel_PackageableElement ::=(=)
5 Composite-state-type-	4 StructuredClasses_Class
definition-**set**	5 BehStateMachines_StateMachine
Procedure-definition-**set**	3 Communications_Signal

Table 3.3: Common syntax of packages

Agent-type/Class

UML 2.0 introduces structured classes, which are classes extended with internal structure and ports (see Table 3.4). Structured classes are semantically and syntactically similar to Agent-types in SDL. Both have an internal structure of properties (respectively agents, 9), connectors (channels, 7) and gates (ports, 6). Both agent-types and structured classes can specialise other agent-types and structured classes (2), however SDL only supports single inheritance while UML supports multiple inheritance.

Behaviour is associated with an Agent-type as a State-machine-definition, which consists of a name and a Composite-state-type-identifier. Behaviour is associated with structured classes by a Behavior-Identifier (8). Behaviour in the abstract syntax of UML is a synonym for statemachines and other behaviour models. Statemachines are syntactically similar to composite-state-types in SDL. The abstract syntax of the two languages differs slightly, since UML does not have a State-machine-definition. In the common abstract grammar, we include the State-machine-definition but discard the name associated with it, since it does not exist in UML.

SDL-2000 (AS1)	UML 2.0 (derived AS)
Agent-type-definition ::=(::)	StructuredClasses_Class ::=(::)
1 Agent-type-name	1 [String]

	Agent-kind
2	[Agent-type-identifier]
	Agent-formal-parameter *
	Data-type-definition-**set**
	Syntype-definition-**set**
3	Signal-definition-**set**
	Timer-definition-**set**
	Exception-definition-**set**
	Variable-definition-**set**
4	Agent-type-definition-**set**
5	Composite-state-type-definition-**set**
	Procedure-definition-**set**
9	Agent-definition-**set**
6	Gate-definition-**set**
7	Channel-definition-**set**
(8)	[State-machine-definition]
State-machine-definition ::=(::)	
	State-name
8	Composite-state-type-identifier

	...
	Kernel_Classifier-Identifier-**set**
2	StructClasses_Class-Identifier-**set**
	[Kernel_Type]
	Kernel_ElementImport-**set**
	Kernel_PackageImport-**set**
	Kernel_Constraint-**set**
	Kernel_Behavior-**set**
8	[Kernel_Behavior-Identifier]
	Boolean /*isActive*/
	Communications_Reception-**set**
6	Ports_Port-**set**
7	CompStruct_Connector-**set**
9	IntStruct_Property-**set**
	Kernel_Property *
	Kernel_Classifier-**set**
	Kernel_Operation *
Kernel_Classifier ::=(=)	
4	StructuredClasses_Class
5	BehStateMachines_StateMachine
3	Communications_Signal
	...

Table 3.4: Common syntax of agent types and classes

Signals

Signal types exist in SDL and UML to describe communication between agents/objects (see Table 3.5). Signals have a name (1) and parameters, which are represented by sorts in SDL and properties in UML. While representing similar concepts, the abstract syntax of sorts and properties are different, therefore signals in the common abstract grammar have no parameters.

SDL-2000 (AS1)	
Signal-definition ::=(::)	
1	Signal-name
	Sort-reference-identifier *

UML 2.0 (derived AS)	
Communications_Signal ::=(::)	
	Kernel_Property *
1	[String]
	...

Table 3.5: Common syntax of signals

Channel/Connector

Channels/connectors connect gates/ports (see Table 3.6). In SDL, a channel has one or two channel-paths. In case of two channel-paths, the channel is bi-directional and the originating gate of the first path is the destination gate of the second path and vice versa. In UML, the connector connects two or more ports. In the common AS, a channel is a set of channel-ends (2), which is a pair of ports (3). No direction is specified.

SDL-2000 (AS1)	UML 2.0 (derived AS)
Channel-definition ::=(::)	IntStruct_Connector ::=(::)
1 Channel-name	IntStruct_Connector-Identifier
[NODELAY]	2 Ports_ConnectorEnd * /* 2..* */
2 Channel-path-**set**	[Kernel_Association-Identifier]
	...
Channel-path ::=(::)	1 [String]
3 Originating-gate	[Kernel_Type]
3 Destination-gate	Ports_ConnectorEnd ::=(::)
Signal-identifier-**set**	3 [IntStruct_ConnectableElement-Identifier]

Table 3.6: Common syntax of channels and connectors

Gate/Port

Gates/ports are endpoints for channels/connectors (see Table 3.7). Gates specify valid signals for both directions, while ports have required and provided interfaces (2, 3).

SDL-2000 (AS1)	UML 2.0 (derived AS)
Gate-definition ::=(::)	Ports_Port ::=(::)
1 Gate-name	3 Interfaces_Interface-Identifier-**set**/*required*/
2 In-signal-identifier-**set**	2 Interfaces_Interface-Identifier-**set**/*provided*/
3 Out-signal-identifier-**set**	Ports_Port-Identifier-**set**
	IntStruct_ConnectorEnd-**set**
	1 [String]
	...

Table 3.7: Common syntax of gates and ports

Agent/Property

Agents and properties (see Table 3.8) are both instances of a type (2) (agent-type in SDL, structured class in UML). Both specify upper and lower bounds for the number of instances (3). While the lower bound in UML is optional, it is required in SDL, and therefore in the common abstract syntax.

SDL-2000 (AS1)	UML 2.0 (derived AS)
Agent-definition ::=(::)	IntStruct_Property ::=(::)
1 Agent-name	Kernel_AggregationKind
Number-of-instances	Kernel_Property* /*subset*/
2 Agent-type-identifier	Kernel_Property* /*refined*/
	[Kernel_ValueSpecification]
Number-of-instances ::=(::)	[Kernel_Association-Identifier]
3 Initial-number	Ports_ConnectorEnd-Identifier-**set**
3 [Maximum-number]	1 [String]
	2 [Kernel_Type-Identifier]
Initial-number ::=(=) Nat	3 [Kernel_ValueSpecification]
Maximum-number ::=(=) Nat	3 [Kernel_ValueSpecification]
	...

Table 3.8: Common syntax of agents and properties

Parameter

Like signals, parameters in the common AS (see Table 3.9) have a name (1) but no sort or type, since the abstract syntax of sorts and types is different.

SDL-2000 (AS1)	UML 2.0 (derived AS)
Parameter ::=(::)	Kernel_Parameter ::=(::)
1 Variable-name	1 [String]
Sort-reference-identifier	[Kernel_Type-Identifier]

Table 3.9: Common syntax of parameters

Composite-state-type/Statemachine

Composite-state-types as well as statemachines (see Table 3.10) have a name (1), a sequence of parameters (3) and an identifier of the composite-state-type/statemachine that they specialise (2), if any. In UML, a statemachine has one or more regions that contain states and transitions. The equivalent in SDL is a Composite-state-graph (one region) or a State-aggregation-graph (two or more regions). A Composite-state-graph contains a State-transition-graph which contains the states of the Composite-state-type. A Region in UML maps to a State-transition-graph in SDL. Both contain the states (5) and transitions of the composite-state-type/statemachine. Multiple regions are not included in the common AS, because of the different syntax and semantics in SDL and UML.

SDL-2000 (AS1)	UML 2.0 (derived AS)
Composite-state-type-definition ::=(::)	BehSM_StateMachine ::=(::)

1 State-type-name	BehSM_Pseudostate-**set**
2 [Composite-state-type-identifier]	4 BehSM_Region +
3 Composite-state-formal-parameter*	2 [BehSM_StateMachine-Identifier]
State-entry-point-definition-**set**	3 Kernel_Parameter *
State-exit-point-definition-**set**	BasBeh_Behavior-Identifier-**set**
Gate-definition-**set**	...
...	Kernel_Constraint-**set**
Composite-state-type-definition-**set**	Kernel_Constraint-**set**
Variable-definition-**set**	Kernel_Property-**set**
Procedure-definition-**set**	Kernel_Class-Identifier-**set**
[Composite-state-graph |	Kernel_Classifier-**set**
State-aggregation-node]	Kernel_Operation-**set**
Composite-state-graph ::=(::)	...
4 State-transition-graph	Kernel_VisibilityKind
[Entry-procedure-definition]	1 [String]
[Exit-procedure-definition]	Kernel_Classifier-Identifier-**set**
Named-start-node-**set**	Kernel_Generalization-**set**
State-transition-graph ::=(::)	BehSM_Region ::=(::)
[On-exception]	5 BehSM_Vertex-**set**
[State-start-node]	BehSM_Transition-**set**
5 State-node-**set**	[BehSM_Region]
Free-action-**set**	[Kernel_VisibilityKind]
Exception-handler-node-**set**	[String]

Table 3.10: Common syntax of composite states and state machines

State-node/State

SDL-2000 (AS1)	UML 2.0 (derived AS)
State-node ::=(::)	BehSM_State_Concrete ::=(::)
1 State-name	[Beh_ConnectionPointReference]
[On-exception]	2 [BehSM_StateMachine-Identifier]
Save-signalset	...
Input-node-**set**	Com_Trigger-Identifier-**set**
Spontaneous-transition-**set**	BehSM_Region-**set**
Continuous-signal-**set**	[BehSM_State-Identifier]
Connect-node-**set**	BehSM_Transition-Identifier-**set**
2 [Composite-state-type-identifier]	BehSM_Transition-Identifier-**set**
	1 [String]

Table 3.11: Common syntax of states

State-nodes in SDL are similar to states in UML, however, the syntax is different (see Table 3.11). Both have a name (1) and an identifier of the composite-state-type/statemachine that is the submachine of this state (2), if any. States are the source of transitions, but in SDL these transitions are associated with the trigger of the transition (Input-node) and in UML with the state itself.

3.5 Semantic Comparison of SDL and UML

In the previous section, we have argued for the necessity to take the semantics of language elements into account. Particularly on the behavioural level, UML and SDL differ syntactically, but have several concepts in common semantically. Following the conclusions reached so far, we now compare corresponding language elements of UML and SDL on a semantic level.

While the semantics of SDL is defined completely, UML only provides semantics for fragments of the language. Therefore, we choose UML statecharts and SDL process graphs for this comparison. UML statecharts have a semantics definition with few omissions and variation points. Several attempts to formally define the behaviour of statecharts exist, for example [5].

The syntactical comparison of UML and SDL revealed that the abstract syntax of statecharts and process graphs is very different. However, there are several language elements in both languages that have a similar notation and represent corresponding concepts, despite major syntactical differences. For example, both languages have the concept of a guarded transition. In the following, we compare the semantics of corresponding constructs of UML statecharts and SDL process graphs. We omit those cases where corresponding constructs are semantically different. Work that provides mappings for these cases, can be found in [62] for SDL-92 to an executable subset of UML 1.3 (using Rational-specific extensions). In this section, we provide a tool-independent comparison of SDL-2000 and UML 2.0. This work was published as technical report 327/03 [23].

States

Both UML and SDL have the concept of states as a condition in which the state machine waits for an event to occur. The state (or set of states) the state machine waits in is called the active state. In UML, it can also model a condition in which the process performs a certain activity (do-activity). There are several kinds of states (see [52], 9.3.11):

- **Simple States:** A simple state is a state without sub-states.
- **Composite States:** A composite state is a state that contains sub-states, which can again be composite states. In UML, the sub-states of a composite state are partitioned into one or more regions. If the state machine is in the composite state, it is also in *exactly one* sub-state of each region (this applies recursively). SDL distinguishes between composite states, which have a set of sub-states of

which exactly one is active when the state is entered (composite state with one region in UML), and state aggregations, which consist of several composite states that are interpreted in an interleaving manner (composite state with multiple regions in UML).

- **Submachine States:** UML introduces submachine states, which are semantically equivalent to UML composite states. Syntactically, they are closer to SDL composite states than the composite states in UML (Figure 3.2). Submachine states make it possible to build the specification in a modular way.

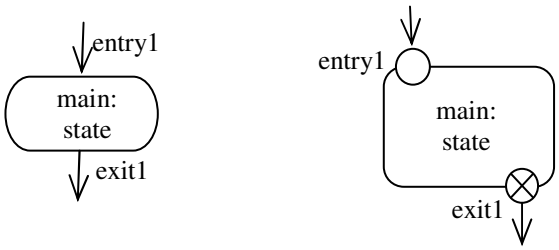


Figure 3.2: Submachine (composite) states and entry/exit-points in SDL and UML

Unlike in SDL, simple states in UML can have entry-activities (exit-activities) that are executed when the state is entered (left).

Composite states in SDL and submachine states in UML (both referred to as composite states below) have entry- and exit-procedures/activities. A transition targeting the composite state leads to an entry of the state at the default entry point (*initial* pseudostate in UML, unnamed state start node in SDL). A transition to an entry connection-point reference of the composite state leads to an entry of the state at the entry point that is referenced by the connection-point reference. If a region of a UML composite state reaches an exit point, or if all regions reach the final state, the composite state is left at the corresponding exit connection point reference (completion transition in case of leaving via final states). In SDL, the composite state is left when all state partitions have reached a return node. The state is left via the corresponding exit point of the composite state. If more than one exit point is valid, one is chosen in an indeterministic way.

Signals and Events

State transitions are triggered by *signals* in SDL and events in UML. An SDL agent has an input port associated with its state machine, in which signals are queued in the order of their arrival time (see [35], Chapter 9). If a signal is *saved* in a state, it is not enabled. The signal that is dispatched is the first enabled signal in the queue; the saved signals in the queue are retained in the order of their arrival (see [35], 11.7).

UML objects have an event pool, with an unspecified ordering of events (see [52], 7.3.5), with the exception of *completion events*, which are dispatched before any other

event in the pool (see [52], 9.3.14). If an event is *deferred* in a state, it is retained in the event pool as long as the state machine is in a state where the event is deferred, or a transition for this event is enabled (see [52], 9.3.11). No priority between events that become enabled again and non-deferred events is specified. If an event was deferred in the previous state and is not deferred in the current state, it has no priority over an event that was not deferred at all.

SDL signals and UML events are both referred to as 'events' in the remainder of this section.

Timers

In SDL, a *timer* can be set to expire at a specified time or after a duration. When the timer expires, an event is generated and put into the input queue. This can trigger a transition when the event is dispatched (see [35], 11.15). UML has TimeEvents that occur at a certain point in time or after a specified duration (though the starting time is not defined in the standard). TimeEvents can trigger a TimeTrigger (see [52], 7.3.27).

Transitions

Transitions in UML and SDL have a *run-to-completion* semantics, meaning that an event is completely processed before the next event can be handled.

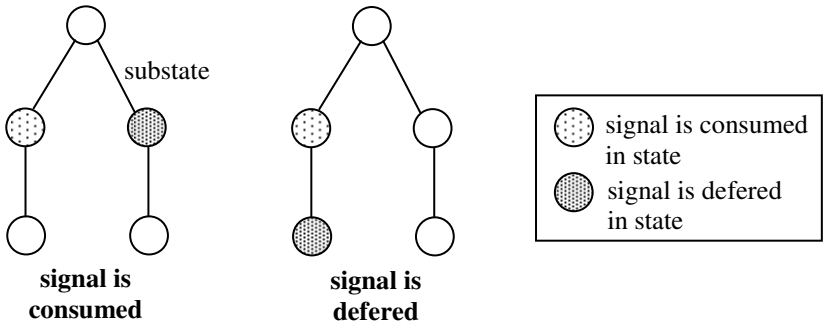


Figure 3.3: Priority of consume and defer

Transitions with a trigger and no conditions have the same semantics in SDL and UML for simple and composite states. The transition in the active state is enabled when the event the transition is labelled with is dispatched. When the transition is fired, the source (composite) state of the transition is left, a sequence of actions is executed, and the target (composite) state is entered. More than one transition can be enabled in a state for an event. When the state machine is in a composite state, and both the composite state and an active sub-state have a transition for the dispatched event, both transitions are enabled. In SDL and UML, the transition of the nested state has a higher priority than the transition of the containing state, and the transition of the sub-state is fired (see [35], 11.11 and [52], 9.3.12). This also

applies to conflicts between deferred (saved) and consumed events (Figure 3.3). In case of conflicts between consumed and deferred events in orthogonal states, the event is consumed (Figure 3.3, left). In case of conflicts between consumed and deferred events in current states where one state is the substate of another, the substate takes precedence (Figure 3.3, right). The former case is only possible in UML, since SDL requires distinct sets of events for orthogonal states.

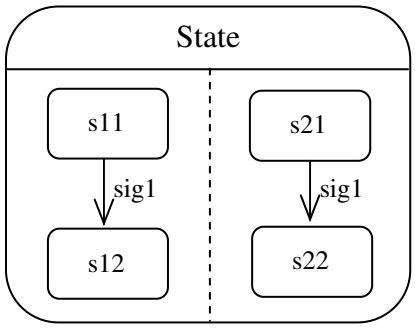


Figure 3.4: State with two enabled transitions (UML)

States in SDL can not have more than one transition for the same event (see [35], 11.2), and orthogonal states must have a disjoint set of input signals (see [35], 11.11.2). However, in UML this is not the case. If there is more than one transition for the same event in the active state, one of them is processed. For every orthogonal region, if there is a transition for the dispatched event in the active state, it is enabled and can be processed. More than one transition, up to the number of orthogonal regions in the state, can be fired in the same run-to-completion step in arbitrary order (see [52], 9.3.12). In Figure 3.4, if the statemachine is in the states s11 and s21 and the event sig1 is dispatched, both transitions are enabled and fired, since they do not conflict with each other.

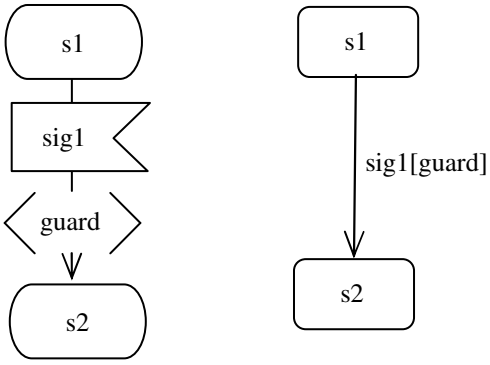


Figure 3.5: Transitions in SDL and UML

Both SDL and UML support transitions that have *guards* (*enabling conditions* in SDL) (Figure 3.5). Transitions are only enabled when their guard evaluates to true. If the guard evaluates to false, in SDL the event is not enabled and the next event is selected from the queue (see [35], 11.6). UML discards the event, unless it is explicitly deferred or there is another transition for this event that is enabled (see [52], 9.3.12 and 9.3.14). Therefore the semantics of guards differ between SDL and UML.

Transitions (with guard) that are not explicitly labelled with an event are called *continuous signals* in SDL and *completion transitions* in UML (see Figure 3.6). Continuous signals are fired when their guard is true and the event queue is empty (see [52], 11.5). Completion transitions are fired when a completion event occurs and their guard is true. Completion events are dispatched before all other events. They occur when a do-activity is finished or when a composite state is left because all regions have reached a final state (see [35], 9.3.14). A completion transition in UML has a similar semantics as a *connect-node* in SDL. A connect-node originates from a composite state and is taken when the composite state is left via the default exit point. A connect-node cannot have a guard.

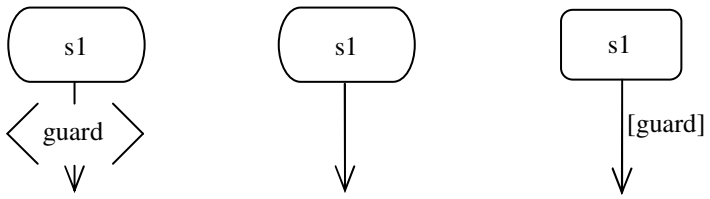


Figure 3.6: Continuous signal, connect node and completion transition

History

UML has two *history pseudostates*, deep history and shallow history (see [52], 9.3.8). Entering a composite state over a history state leads to the restoration of the active states as it was when the composite state was left. For a deep history state this applies recursively for all substates. All entry activities of states that are entered are executed. In SDL, a transition can end in a *history nextstate*. In this case, the next state is the one in which the transition originated. In case of a composite state, the state is re-entered and the entry-procedure is invoked (see [35], 11.12.2.1).

Actions

In UML, actions are used to describe behaviour. Actions take inputs and transform them into outputs, possibly modifying the state of the system. Outputs of actions can be connected to inputs of other actions by an activity flow, as in a statechart transition. UML supports a number of primitive actions that more complex actions can be mapped on. In SDL, transitions perform a sequence of actions. Actions in SDL

manipulate data, output signals and call procedures. SDL supports a small number of specific actions important for communication systems.

- **Task/StructuralFeatureAction, VariableAction:** Tasks in SDL can contain assignments that are interpreted when the task is interpreted. In UML, assignments can be realised by a combination of actions. With ReadStructuralFeatureAction/ReadVariableAction and WriteStructuralFeatureAction/ WriteVariableAction, attributes and variables can be read and written. With ApplyFunctionAction, primitive functions can be applied.

- **Create/CreateObjectAction:** CreateObjectAction creates a new object for a given classifier, without further initialisation. The Create action in SDL creates a new agent in the scope of the creating agent. Its variables are created and its formal parameters initialised.

- **Procedure Call/CallOperationAction:** When a procedure is called in SDL, the interpretation continues at the start node of the procedure graph that was invoked by the call. It resumes after the call node when the interpretation of the procedure is finished. A CallOperationAction in UML leads to the execution of an operation in a local or remote object. If the call is synchronous, the interpretation of the transition resumes after the CallOperationAction when the operation is finished. How parameters and results are transmitted is not specified.

- **Output/SendSignalAction:** An output action in SDL leads to the creation of a signal instance of the specified type and parameters. The signal instance can have an agent set or an agent as target, or it can be transmitted to any agent reachable via a sequence of valid channels, possibly restricted by the via argument. In UML, a SendSignalAction leads to the creation of a signal instance of the specified type and parameters. The signal instance is transmitted to the target object. The path the signal takes, its transmission time and the order in which signals arrive are undefined.

- **Decision/Choice:** A decision node in SDL evaluates a question and selects an outgoing transition that has the answer to the question in its range. If no range of an outgoing transition is matched and an else-branch exists, the else-branch is selected; otherwise a NoMatchingResult exception is thrown. SDL forbids having the same answer in more than one range of an outgoing transition. When a Choice pseudostate is reached in UML, the guards of the outgoing transitions are evaluated, and one of the transitions whose guard evaluates to true is selected in an indeterministic way. If there is an else-branch and none of the guards evaluates to true, the else-branch is selected, otherwise the model is ill-formed.

3.6 The UML Profile Approach

3.6.1 Overview

Since the Unified Modeling Language was introduced, the development of SDL has been influenced by UML, and vice versa. The mutual influence became especially apparent with the most recent language standards, SDL-2000 and UML 2.0. With SDL-2000, the first version of the Z.109 standard [33] was introduced, which described the combined use of SDL and UML 1.3 by providing a mapping from UML to SDL, using *UML profiles*.

With the UML 2.0 standard, the Unified Modeling Language took a big step towards SDL, incorporating many features of the language. For example, structured classes model architecture in a fashion similar to SDL. UML 2.0 also comes with a mature UML profile mechanism, defining it as a specific meta-modelling technique. Profiles have become a part of the UML meta-model, defined in the Profile package, giving UML profiles a formal abstract syntax. The new version of the Z.109 standard [39] takes these changes into account and defines a UML Profile for SDL based on the UML 2.0 and SDL-2000 standards.

SDL is a more mature and complete language than UML, with few semantic variation points (for example, implicit addressing of signals) and a formal semantics. The UML Profile for SDL utilises this by taking SDL as the semantic basis for UML. On the other hand, using UML as front end language utilises its advantages in the early phases of software development, and its integration of different modelling techniques. Mapping UML specifications to SDL provides more flexibility than the common syntactic and semantic basis described in the previous sections.

3.6.2 UML Profiles

The Unified Modeling Language aims at being a universal language for modelling software systems in the early phases of software development, particularly the requirements and design phases. To achieve this goal, UML provides a complete semi-formal abstract syntax definition of the language, using meta-models, but leaves the semantics definition imprecise and incomplete. *Semantic variation points* in the language definition identify parts of the semantics that are explicitly left open for interpretation, or where alternative interpretations are provided. For example, the event pool of a classifier instance is a collection of events that occurred at this instance. Events in the event pool can trigger classifier behaviour. The order in which the events are processed is intentionally left open. This enables a tool provider to implement a strategy that fits the target domain of the tool within the framework given by UML, for example first-in-first-out or priority based strategies.

Semantic variation points make UML a flexible modelling language that can be adapted for a large variety of target domains. Tool providers resolve semantic variation points when implementing a subset of UML, providing a domain-specific solution. However, these solutions are tool-specific, not standardised, and often proprietary. This is a disadvantage for the interoperability of UML tools.

UML provides the UML profile mechanism to extend and adapt existing meta-models using special classes called *stereotypes*. Using UML profiles, it is possible to give precise semantics to parts of the language, and to tailor it for different platforms and domains. UML profiles are tool-independent and can be defined as separate standards, augmenting the UML language definition. Standardised UML profiles include profiles for CORBA, quality-of-service and real-time, and testing.

UML profiles are not a first-class extension mechanism, that is, it is not possible to modify existing semantics of UML. Profiles can add constraints to a meta-model, provide semantics that does not conflict with the semantics of the meta-model, and add different notation for already existing symbols. This ensures that the model with applied stereotypes is still a valid UML model, which can be processed by a UML tool with sufficient compliance to the UML standard. For extensions that modify the semantics of UML, the meta-model itself must be modified. However, this is not recommended.

UML Profile Definition

UML 2.0 defines the UML profile mechanism as a part of the UML meta-model, giving it a formal abstract syntax. Profiles and stereotypes are integrated as specialisations of packages and classes, respectively. The notation to be used for defining UML profiles is generally left unspecified. The Z.119 standard [38] gives a guideline for defining UML profiles for ITU languages in a similar fashion to the UML superstructure document, describing semantics and constraints of a stereotype using informal language and OCL [54].

A profile is a specialised package that is applied to other packages (including profiles) via a *profile application*. A profile uses the same notation as a package, with the keyword «profile» attached to the name of the package.

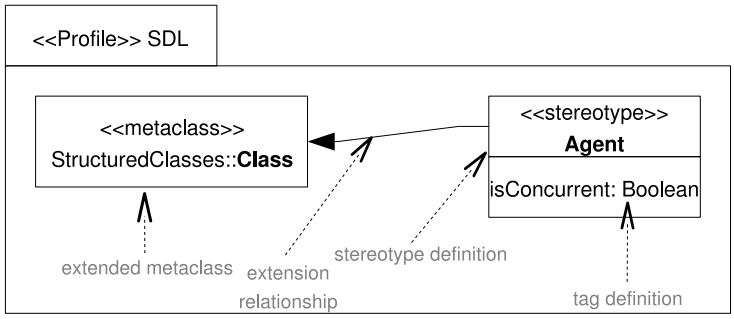


Figure 3.7: Profile package with stereotype

The profile consists of a set of owned *stereotypes*. A stereotype is a kind of meta-class that is linked to a meta-class of the referenced meta-model. Like classes, stereotypes can have properties, called *tag definitions*. Applying a stereotype to a meta-class adds

the constraints, semantics and notations of the stereotype to the meta-class. Figure 3.7 shows the graphical notation of a profile SDL, which contains a stereotype Agent that extends the meta-class Class from package StructuredClasses. Stereotype Agent defines a tag definition isConcurrent of type Boolean. Semantics and constraints can be added to Agent as long as they don't conflict with existing semantics and constraints.

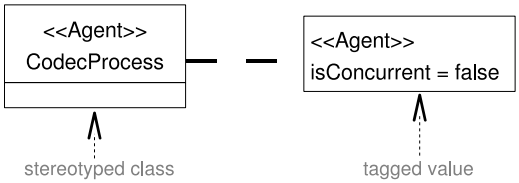


Figure 3.8: Stereotyped class

Figure 3.8 shows a UML model with stereotype Agent applied, using the standard notation for stereotyped UML classes. If the stereotype Agent defines a notation, for example the graphical syntax of process agents in SDL, it can be used instead. Tag value isConcurrent, defined in the stereotype, is set to false.

3.6.3 The UML Profile for SDL

The UML Profile for SDL [43] gives a precise meaning to a subset of UML by mapping UML meta-model elements to elements of the SDL abstract syntax. Several meta-model classes are stereotyped, defining constraints and semantics to tailor the language to SDL. The semantics is defined as a mapping of the stereotyped meta-model classes to the abstract syntax of SDL, thus reusing the formal semantics definition of SDL [42]. The meta-model and abstract syntax elements related by this mapping bear a strong resemblance to the common abstract syntax derived in Section 3.4: for example, packages are mapped to *Package-definitions*, active classes to *Agent-definitions*, and signals to *Signal-definitions*. The mapping defined by the UML Profile for SDL is more flexible than the common syntax and semantics approach. Apart from mapping UML features to directly corresponding SDL features, features that do not have a direct representation in SDL are translated to a combination of related features. For example, while UML guards do not have a direct representation in SDL, they are mapped by the UML Profile for SDL to a decision symbol at the start of the outgoing transition. Furthermore, constraints are added to the UML meta-model, leading to a subset of UML.

For each meta-model class included in the profile, several aspects are defined:

- **Attributes** (tag definitions): Additional attributes defined by the stereotype that can be set in the model. Attributes give additional information that can otherwise not be expressed in the meta-model, and that is important for the mapping of model-elements to the abstract syntax. For example, the stereotype «ActiveClass», which extends Class, defines the attribute isConcurrent of type

Boolean. A class with stereotype «ActiveClass» is mapped to an *Agent-type-definition* in the abstract syntax, and attribute isConcurrent defines the *Agent-kind* of the *Agent-type-definition* - BLOCK if true and PROCESS if false.

- **Constraints**: Constraints define additional checks and conditions that the meta-model must satisfy. The meta-model is constrained to a subset for which a mapping to SDL is provided. For example, a pair of signal-triggered transitions from different orthogonal regions must have distinct triggers.

- **Semantics**: Gives a precise semantics to meta-model elements by describing a mapping to the abstract syntax of SDL, thus reusing its formal semantics definition [42]. Meta-model elements are mapped directly to the AS1 of SDL, bypassing transformations in SDL from AS0 to AS1. The following excerpt from [43] describes a mapping of transitions with a ChangeEvent as trigger.

  > "If the trigger event of a «Transition» Transition is a ChangeEvent, the transition is mapped to a *Continuous-signal*. The changeExpression maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*."

- **Notation**: Describes the notation to be used for the stereotyped model element. Z.109 almost exclusively uses UML standard syntax. For some elements, additional textual syntax is introduced.

3.6.4 Formalisation of the UML Profile for SDL

The UML Profile for SDL in [43] is defined in an informal fashion, similar to the UML language definition. The SDL language definition, on the other hand, includes a complete formalisation of static and dynamic aspects of the language. In order to carry over the mathematical precision of SDL to the subset of UML covered by the profile, it was proposed to create a formal definition of Z.109. A formal definition of Z.109 would have several advantages. For example, constraints formulated using the Object Constraint Language (OCL) can be automatically checked by most UML tools. From an operational formal definition of the mapping to the SDL abstract grammar, tool support can be automatically generated, as it is done in the case of the formal semantics of SDL.

Figure 3.9 provides an overview over the steps taken in the profile definition. The intent is to provide a mapping from model elements of UML, described by a meta-model (UML MM), to abstract syntax elements of SDL, described by an abstract grammar AS1 (SDL AG). This mapping consists of two orthogonal steps: a mapping from UML to SDL (M and M'), and a mapping between abstract syntax representations (m), from meta-models to abstract grammars. In addition to the mapping, constraints are defined on the meta-model, for example using OCL, and transformations (T) are performed on the UML side, since the mapping targets the already reduced abstract grammar AS1.

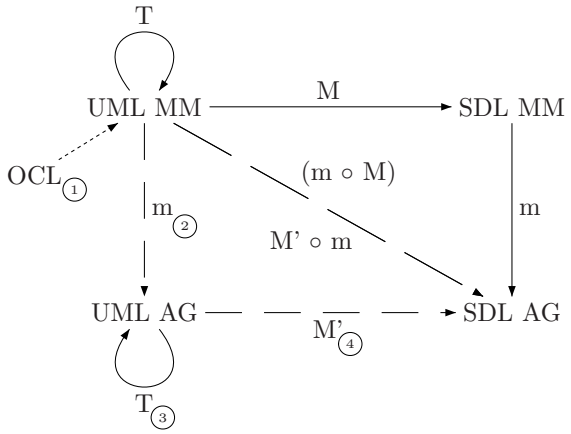


Figure 3.9: Mapping from UML meta-model to SDL abstract syntax

A way to perform the mapping is to map the UML meta-model to a UML abstract grammar first (m), then to map the UML abstract grammar to SDL (M'). Based on this, we have devised a formal approach to the definition of the UML profile for SDL, which we survey in Section 3.6.5. The mapping between the abstract syntax representations can be derived from the mapping we defined in Section 3.3. The mapping M' is a mapping between two different abstract grammars. Such a mapping can be found in the formalisation of the static semantics of SDL (Z100 Annex F Part 2 [41]), where the abstract grammar AS0 is mapped to the abstract grammar AS1. Generally, the mapping M' is more complicated than the mapping in Z100.F2, since the mapping is performed between two different languages.

The mapping of the UML Profile for SDL covers a subset of SDL. Core features of SDL that are not covered include *timers*, *exceptions*, *enabling conditions*, *entry-* and *exit-procedures*. This subset defines an *SDL Profile*, for which a tailor-made formal semantics can be extracted [21, 25].

3.6.5 Survey of an SDL-style Formalisation Approach

In this section, we present our partial formalisation of the Z.109 standard, with the focus on transitions. This work was published as technical report 350/06 [22]. The approach is to apply the techniques used for the formalisation of the static semantics of SDL-2000 [41] as much as possible. This gives us the advantage of using an approach that has been applied successfully before, and for which tool support is available, allowing us to concentrate on the formalisation itself. The sequence of steps taken in our approach is illustrated in Figure 3.9, with numbers 1) to 4). Each step is described in this section in a paragraph with a corresponding number.

1) Formalising the Constraints.

The stereotypes in the UML Profile for SDL introduce additional constraints on the meta-model classes they extend. For the formalisation of these constraints, the Object Constraint Language (OCL) [54], which is used thoughout the UML superstructure specification, is a self-evident choice. The OCL is tailor-made for specifying constraints for MOF-compatible [53] meta-models. It provides a logic that allows navigation over properties and association ends of classifiers. In the following are the constraints specified for the «Transition» stereotype formulated in OCL. Unless specified otherwise, all OCL expressions are formulated in the context of meta-class Transition.

- The Transition shall have kind == external or local. The UML concepts of internal transitions are not allowed.

  $self.kind = \#external$ **or** $self.kind = \#local$

- The port of the Trigger[1] shall be empty.

  $self.trigger{->}forAll(t \mid t.port{->}isEmpty())$

- In the Transition set defined by the outgoing properties of a State, the signal property of each event property that is a SignalEvent of each trigger shall be distinct.

  context *State*
  $self.outgoing{->}forAll(t1,t2: \text{TRANSITION} \mid t1.trigger{->}select(event.oclIsKindOf(\text{SIGNALEVENT})).event.signal{->}intersection(t2.trigger{->}select(event.oclIsKindOf(\text{SIGNALEVENT})).event.signal){->}isEmpty())$

- The event property of the trigger property shall be a MessageEvent or ChangeEvent.

  $self.trigger{->}forAll(t \mid t.event.oclIsKindOf(\text{MESSAGEEVENT})$ **or** $t.event.oclIsKindOf(\text{CHANGEEVENT}))$

- The effect property shall reference an Activity.

  $self.effect{->}notEmpty()$ **implies** $self.effect.oclIsKindOf(\text{ACTIVITY})$

While the informally specified constraints of the «Transition» stereotype are intuitive and easy to understand, three issues were discovered when specifying the constraints in OCL, two of them concerning multiplicities.

- A transition in UML can have an arbitrary number of triggers, while the stereotype constraints only assume a single trigger at most. The OCL constraints were formulated in a way that allows an arbitrary number of triggers, however, there should be an additional constraint that a transition should have at most one trigger.

[1] The UML meta-model actually defines a set of triggers for a transition.

- The informally specified constraints leave it unclear if the effect property is allowed to be empty. This has been clarified in the formalisation.
- The third constraint is formulated more naturally in the context of «State», since it deals with sets of transitions of a state. Therefore, the constraint should be part of the stereotype «State».

2) Mapping the Metamodel to an Abstract Grammar.

In order to apply the approach from the formalisation of the static semantics of SDL, which provides a mapping between two abstract grammars, we first have to provide a mapping from the meta-model of the UML profile to a UML abstract grammar (mapping m in Figure 3.9). Since UML does not define an abstract grammar, we apply the mapping defined in Section 3.3 to the UML meta-model. The result is an UML abstract grammar together with the mapping m between the meta-model and abstract grammar, which we document using OCL expressions.

The UML Profile for SDL constrains the meta-model defined in the UML superstructure specification to classes and associations that can be expressed in SDL. In some cases, elements are not constrained but no mapping to SDL is defined, since no semantics is associated with them. These elements can be omitted in the extracted abstract grammar, keeping it concise. In the mapping of meta-model class Transition to the non-terminal *Transition*, name and visibility of the Transition are omitted. The result is a production rule (concatenation):

Transition(TransitionKind, [Trigger], [Constraint], [Activity], Vertex−Identifier, Vertex−Identifier, **Integer**)

Apart from the production rule, we get the mapping m for class Transition, describing the mapping to the extracted production rule. Role names of the associations are used to navigate in the meta-model, and to define the relation between elements of the meta-model and elements of the abstract grammar. The auxiliary function *toId* maps meta-classes to identifiers as required by the non-terminal.

context TRANSITION::*m*
mk−Transition(*kind*, *trigger*−>*any*(), *guard*, *effect*, *source*.*toId*, *target*.*toId*, *priority*) `--tag definition priority--`

toId: METACLASS → IDENTIFIER

The mapping from meta-models to abstract grammars naturally maps general associations to identifiers and aggregation or composition to subtrees in an abstract syntax tree (see Section 3.3.3). In few cases, due to the different structure of the abstract syntax of SDL and UML, it is of advantage to map general associations in the same way as compositions. For example, in UML, *states* and *transitions* are related by general associations, while in SDL the transition is a part of the state. Because a UML transition has a unique source state, it can be mapped as a subtree of a state instead of an identifier, in the SDL fashion.

State(**String**, Trigger−set, Transition−set, ConnectionPointReference−set, Pseudostate−set, StateMachine−Identifier)
Pseudostate(**String**, PseudostateKind, Transition−set) `--Transition-Identifier replaced by Transition--`

context STATE::*m*
  mk−State(*name*, *deferrableTrigger*, *outgoing*, *connection*, *connectionPoint*, *submachine*)
context PSEUDOSTATE::*m*
  mk−Pseudostate(*name*, *kind*, *outgoing*)

Events do not have a counterpart in the abstract syntax of SDL. Therefore, as with transitions, we place events directly inside a trigger instead of an event-identifier. The abstract meta-class *MessageEvent* is merged with the abstract meta-class *Event*, since it doesn't introduce new attributes and associations.

Trigger(Event) `--Event-Identifier replaced by Event--`

context TRIGGER::*m*
  mk−Trigger(*event*)

Event = SignalEvent | CallEvent | ChangeEvent | AnyReceiveEvent

SignalEvent(Signal−Identifier)
CallEvent(Operation−Identifier)
ChangeEvent(ValueSpecification)
AnyReceiveEvent()

context SIGNALEVENT::*m*
  mk−SignalEvent(*signal.toId*)
context CALLEVENT::*m*
  mk−CallEvent(*operation.toId*)
context CHANGEEVENT::*m*
  mk−ChangeEvent(*changeExpression*)

3) Transformations on the UML Abstract Grammar.

To keep the language semantics concise, SDL distinguishes between core constructs of the language, for which the semantics is given directly, and additional constructs, which are expressed through the core constructs. In the abstract grammar AS1 of SDL, which is the target of the UML Profile for SDL, these additional constructs are already eliminated. UML constructs that correspond to additional SDL constructs are therefore transformed before the mapping to SDL is performed.

The transformations can be defined on the UML meta-model, using meta-model transformations, or on the UML abstract grammar. In order to apply the techniques from Z100 Annex F, transformations are defined on the abstract grammar, using rewrite rules on abstract syntax trees.

If the trigger event of a «Transition» Transition is an AnyReceiveEvent, the transition is expanded according to the Model in SDL 11.3 for transforming <asterisk input list> before applying the mapping that follows in this section.

The function collectTriggers computes the set of triggers for a state from the complete valid set of triggers of the enclosing class, minus transitions and deferred signals defined for the state, and minus remote procedures and remote variables. If at least one trigger exists in the set of triggers returned by collectTriggers, the set of transitions is expanded with a copy of the transition triggered by AnyReceiveEvent, but triggered by a trigger from collectTriggers. If the set of triggers is empty, the transition triggered by AnyReceiveEvent is removed.

{ pre, tany=Transition(kind,Trigger(AnyReceiveEvent()),grd,eff,src,trg,prio), rest}
=1=>
if ∃*trig* ∈ collectTriggers(tany.parent) **then**
  {pre, tany, Transition(kind,collectTriggers(tany.parent).take(),grd,eff,src,trg,prio),
    rest}
else
  {pre, rest}
endif

collectTriggers(s: State): Trigger-**set**$=_{\text{def}}$
  let ag: Class = enclosingAgent(s) **in**
    validTriggers(ag) \ (remoteProcedures(ag) ∪ remoteVariables(ag) ∪
      inputTriggers(s) ∪ deferredTriggers(s)
  endlet

4) Formalisation of UML to SDL Mapping.

In order to define the mapping from UML to SDL, we introduce a function *Mapping* from UML abstract syntax trees (DefinitionUML) to AS1 abstract syntax trees (DefinitionAS1) of SDL. The domain of Mapping contains all abstract syntax trees described by the abstract grammar of UML, and their subtrees. A detailed description of these domains is given in [40].

Mapping: DefinitionUML → DefinitionAS1
idToNode: Identifier → DefinitionUML

In the same way as in Z100 Annex F Part 2, the mapping function is a concatenation of cases. A case consists of a pattern on the left hand side, and a resulting syntax tree on the right hand side. The pattern can contain nodes of the UML abstract syntax tree, as well as variables, wildcards ('*'), and a **provided**-clause to constrain the matches. Additionally, we introduce the notation *var*! to express that *var* does not match *undef*. It is a shortcut for specifying *var* ≠ *undef* in the **provided**-clause.

The function *idToNode* provides the same functionality as the function *idToNodeAS1* in the static semantics of SDL. It maps an identifier to the node in the abstract syntax tree that corresponds to the definition the identifier refers to. The operators **s-** and **s2-** have the same semantics as in Z100 Annex F Part 2, selecting a subnode of a specified kind from a node. For example, s.**s-**Transition**-set** selects the set of outgoing transitions from STATE s.

Mapping UML Transitions to SDL. We provide a formalisation for the semantics of the stereotyped class «Transition» Transition by defining the mapping function to abstract grammar AS1. Depending on the properties of the transition, it is mapped to a *Spontaneous-transition*, *Input-node*, *Continuous-signal* or *Connect-node*.

If the trigger event of a «Transition» Transition is a SignalEvent and the name of the Signal is "none" or "NONE" (case sensitive therefore excludes "None"), the Transition is mapped to a *Spontaneous-transition-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Spontaneous-transition-node*.

Transitions triggered by the signal "none" or "NONE" are *Spontaneous-transition*s with undefined *On-exception* and *Provided-expression*. Here, we define the case where the guard of the transition is undefined. Transitions with guard have a more complicated mapping and are defined below.

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),undef,effect,*,target,*)
    provided signal.idToNode.**s-**String ∈ {"NONE", "none"}
  ⇒ Spontaneous-transition(undef,undef,Transition(*Mapping*(effect),
      $Mapping_{trg}$(target.idToNode)))
)

If the trigger event of a «Transition» Transition is a SignalEvent and the name of the Signal is neither "none" nor "NONE" (so it does not map to *Spontaneous-transition-node*), the Transition is mapped to an *Input-node*. The qualifiedName of the Signal maps to the *Signal-identifier* of the *Input-node* and for each ⟨attr-name⟩ in the ⟨assignment-specification⟩ (see the Notation given in UML SS 13.3.24) the qualifiedName of the attribute (with this name) of the context object owning the triggered behavior is mapped to the corresponding (by order) *Variable-identifier* of the *Input-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Input-node*.

Transitions triggered by all other kind of signals are *Input-node*s without *Priority*, *Provided-expression* and *On-exception*. To get the *Signal-identifier* of the Signal, the second subnode of kind string is selected with **s2-**String (**s-**String selects the signal name).

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),undef,effect,*,target,*)
    provided signal.idToNode.**s-**String $\notin$ {"NONE", "none"}
  $\Rightarrow$ Input-node(undef,signal.idToNode.**s2-**String,*Mapping*(assignment-spec),
      undef, undef, Transition(*Mapping*(effect), $Mapping_{trg}$(target.idToNode)))
)

If the trigger event of a «Transition» Transition is a ChangeEvent, the transition is mapped to a *Continuous-signal*. The changeExpression maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the Priority-name.

Transitions that are triggered by a ChangeEvent are mapped to *Continuous-signal*s. The changeExpression is a boolean expression that is mapped to a corresponding SDL expression.

Mapping(
  Transition(*,Trigger(ChangeEvent(changeExpression)),*,effect,*,target,priority)
  $\Rightarrow$ Continuous-signal(undef,*Mapping*(changeExpression),*Mapping*(priority),
      Transition(*Mapping*(effect), $Mapping_{trg}$(target.idToNode)))
)

If the «Transition» Transition has an empty trigger property and a non-empty guard property, the Transition is mapped to a *Continuous-signal*. The guard maps to the *Continuous-expression* of the *Continuous-signal*. The *effect* property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*.

Transitions without trigger but with guard are mapped to *Continuous-signal*s. In this case, the guard defines the condition of the *Continuous-signal*.

Mapping(
  Transition(*,undef,guard!,effect,*,target,priority)
  $\Rightarrow$ Continuous-signal(undef,*Mapping*(guard),*Mapping*(priority),Transition(
      Mapping(effect), $Mapping_{trg}$(target)))
)

If the «Transition» Transition has an empty trigger property and an empty guard property, the Transition is mapped to a *Connect-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Connect-node*. If the source of the Transition is a ConnectionPointReference, this maps to the *State-exit-point-name*. If the source is a State the *State-exit-point-name* should be empty.

Transitions without trigger and guard are mapped to Connect-nodes. The exact mapping depends on the source of the transition. The informal description is imprecise

with regard to how a ConnectionPointReference is mapped to a *State-exit-point-name*. In the formalisation, this is defined precisely as the name of the exit Pseudostate associated with the ConnectionPointReference.

Mapping(
  Transition(*,undef,undef,effect,source,target,*)
    provided source.idtoNode ∈ ConnectionPointReference
  ⇒ Connect-node(source.idToNode.**s2-**Pseudostate.idToNode.**s-**String,undef,
      Transition(*Mapping*(effect), $Mapping_{trg}$(target)))
  | Transition(*,undef,undef,effect,source,target,*)
    provided source.idtoNode ∈ State
  ⇒ Connect-node(undef,undef,Transition(*Mapping*(effect),
      $Mapping_{trg}$(target.idToNode)))
)

If a «Transition» Transition has a non-empty trigger property and non-empty guard property, the guard is mapped to the *Transition* as follows. A *Decision-node* is inserted first in the *Transition* with a *Decision-answer* with a Boolean *Range-condition* that is the *Constant-expression* true and another *Decision-answer* for false. The specification property of the guard property of the Transition maps to Decision-question of the *Decision-node*. The false *Decision-answer* has a *Transition* that is a *Dash-nextstate* without **HISTORY**. The effect property of the Transition maps to the *Graph-node* list of the *Transition* of the true *Decision-answer*.

Guards in UML and enabling conditions in SDL represent the same concept, but have incompatible semantics (see Section 3.5). Mapping guards to SDL is therefore not straightforward, except in the case of continuous signals (see above). To express UML-style guards in SDL, the transition is modified, inserting a decision node with the guard as condition as the first action of the transition.

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),guard!,effect,*,target,*)
    provided signal.idToNode.**s-**String ∈ {"NONE", "none"}
  ⇒ Spontaneous-transition(undef,undef,Transition(< >,
    Decision-node(*Mapping*(guard),undef,
    { Decision-answer(Range-condition(Constant-expression(false)),
        Transition(< >,Terminator(Dash-nextstate(undef)))),
    Decision-answer(Range-condition(Constant-expression(true)),
        Transition(*Mapping*(effect),$Mapping_{trg}$(target.idToNode))) },
  undef)))
)

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),guard!,effect,*,target,*)
    provided signal.idToNode.**s-**String ∉ {"NONE", "none"}
  ⇒ Input-node(undef,signal.idToNode.**s2-**String,*Mapping*(assignment-spec),

undef, undef, Transition(< >,Decision-node(*Mapping*(guard),undef,
{ Decision-answer(Range-condition(Constant-expression(false)),
Transition(< >,Terminator(Dash-nextstate(undef)))),
Decision-answer(Range-condition(Constant-expression(true)),
Transition(*Mapping*(effect),$Mapping_{trg}$(target.idToNode))) },
undef)))
)

A target property that is a State maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* without *Nextstate-parameters*, and where the qualifiedName of the State maps to the *State-name* of the *Nextstate-node*. A target property that is a ConnectionPointReference maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* with *Nextstate-parameters*, and where the qualifiedName of the state property of the ConnectionPointReference maps to the *State-name* of the *Nextstate-node*, and the qualifiedName of the entry property Pseudostate of the ConnectionPointReference maps to *State-entry-point-name*.

Mapping of states is ambiguous. A state can either be mapped as a state owned by a statemachine, or as the target of a transition. The former is a state in SDL, the latter a terminator. To differ between these mappings, we introduce a mapping function $Mapping_{trg}$ to map states as targets of a transition.

$Mapping_{trg}$(
State(name,*,*,*,*,*)
⇒ Terminator(Named-nextstate(name,undef))
| cpr=ConnectionPointReference(PseudoState(name,*,*),*)
⇒ Terminator(Named-nextstate(qualifiedName(state(cpr)),
Nextstate-parameters(< >,name))))

A target property that is a Pseudostate maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) as defined in section 8.6.

Transition targets that are pseudostates are mapped to corresponding terminators in SDL in a straightforward manner. Pseudostates of kind choice are mapped to decision nodes. The mapping to decision nodes is more complicated, because the UML Profile for SDL encodes the decision question in the guard expressions of the outgoing transitions, as the first operand of a two operand guard. The decision question is selected from the guard of a random transition by Constraint.**s-**Expression.**s-**Expression. The second operand, selected by Constraint.**s-**Expression.**s2-**Expression, defines the range condition of a transition originating from the choice pseudostate.

$Mapping_{trg}$(
PseudoState(*,deepHistory,*)

```
  ⇒ Terminator(Dash-nextstate(HISTORY))
  | Pseudostate(name,junction,*)
  ⇒ Terminator(Join-node(name))
  | Pseudostate(*,choice,outgoing)
  ⇒ Decision-node(
      Mapping(outgoing.take.s-Constraint.s-Expression.s-Expression),
      undef,
      {Decision-answer(
        Mapping(t.s-Constraint.s-Expression.s2-Expression),
        Transition(Mapping(t.s-Activity),
        Mapping_trg(t.s2-Vertex-Identifier.idToNode)
      ) | t ∈ outgoing},
      undef)
  | Pseudostate(name,exitPoint,*)
  ⇒ Terminator(Named-return-node(name))
  | Pseudostate(*,terminate,*)
  ⇒ Terminator(Stop-node())
)
```

3.6.6 Conclusions

In Section 3.6.4, we have presented an approach for the formalisation of the UML Profile for SDL. This approach applies techniques from the formalisation of the static semantics of SDL, for which tool support is available. We have applied our formalisation approach to the transition stereotype from the profile, which differs syntactically and semantically from SDL transitions. The formalisation covers all mappings defined in the Z.109 standard for the transition stereotype. Successfully applying our approach to transitions indicates its feasibility for less complicated cases.

The formal definition of the transformations, mappings and meta-model constraints helps detecting errors, omissions and ambiguities in the informal specification of the Z.109 standard (for example, in Z.109 it is unclear how ConnectionPointReference is mapped exactly). This became especially apparent with the meta-model constraints, where an error and an ambiguity were detected in the five informal constraints of the transition stereotype (see Section 3.6.5, Paragraph 1). The formalisations of the mappings and transformations lead to a number of suggested improvements concerning ambiguities in the informal semantics. For the transition stereotype, ten comments and suggested corrections were submitted, and resolved in a subsequent version of the standard.

3.7 Related Work

In [17], Fischer and others argue that support for generalisation and structure give meta-models an advantage over context-free syntax definitions for specifying language syntax. Therefore, a mapping of BNF style grammars to meta-models is provided. For

the resulting simple meta-models, transformations are defined that lead to a better use of meta-model features such as specialisation.

In [66], the semantics of the UML Profile for Communicating Systems is defined by a mapping of the UML meta-model to the abstract grammar AS1 of SDL. The mapping is defined by pre- and post-conditions on the meta-model and the abstract grammar, using OCL [54]. The correctness of a concrete mapping can be verified using these constraints.

The syntactic and semantic comparison is taken further by the new Z.109 standard [39], defined by the ITU. With the new standard, SDL is integrated into UML as a UML profile, which defines a mapping of a subset of UML to a subset of SDL-2000. Currently, this mapping is defined in an informal fashion, and covers most features of SDL-2000. A tool that complies to the UML Profile for SDL supports a subset of UML with precise semantics, which can be exchanged with other compliant tools while retaining the semantics of the model. This subset can be combined with other UML model elements not covered by the profile. However, the semantics of these model elements can differ between UML tools. This approach is taken by the Telelogic Tau Generation 2 tool, which supports a subset of UML that is mapped to SDL, using a proprietary profile. Apart from using SDL as the semantic foundation of the UML subset, the user can switch between the concrete syntax of SDL and UML for statemachines.

3.8 Summary and Conclusions

In this chapter, we have argued that the harmonisation of languages requires a common syntactic and semantic basis. Following this line, we have first defined conceptually sound and well-founded mappings from meta-models - used to define the abstract syntax of UML - to abstract grammars - used by SDL -, and vice versa. By applying these mappings, we have then extracted common grammar rules, arriving at a common abstract grammar for several language constructs. While the results were encouraging for structural language elements, it turned out that the coverage was below expectations for behavioural constructs. From this experience, we have drawn the conclusion that an extraction on a purely syntactical basis is not sufficient. Therefore, we have compared language elements on a semantic basis, too. Here, the results of the syntactical study provided valuable information and feedback.

Based on our mapping between meta-models and abstract grammars, and the mapping between abstract grammars defined in the formal language definition of SDL, we have proposed an approach to the formalisation of the Z.109 standard. The advantages of this approach are a well-defined and precise mapping from UML to SDL, and the ability to generate tool support from the formal definition for checking conditions and mapping specifications.

4 Profiling of Modelling Languages

In this chapter, we contribute the concept of language profiles as a way to define sublanguages of a language. We define consistency between language profiles, based on the dynamic semantics of the sublanguages defined by the profiles[1]. Focusing on formal semantics defined using Abstract State Machines, we describe the problems defining and verifying consistency for Distributed Abstract State Machines, propose solutions, and argue for deriving formal semantics definitions from the complete formal semantics, using language profiles, as a way to guarantee consistency. Finally, we introduce several language profiles for SDL - called SDL profiles - defined by tools and standards, in an informal fashion.

4.1 Language Profiles

In order to support a wide range of applications, system modelling languages are often complex and expressive. The complexity of the languages leads to language definitions that are long and hard to understand, and can limit their applicability in domains for which specialised, tailor-made languages are preferred. Another drawback is that tool support for complex languages usually covers only parts of the language. For example, there is no tool that supports the whole of SDL-96, and only a few of the language constructs introduced in SDL-2000 are supported.

A *language profile* defines a subset of a language, consisting of a language core (for example, the Kernel package of UML), and a set of language features that extend the core. A language profile can also be characterised by the features excluded, compared to the complete language definition. Using language profiles, it is possible to define sublanguages of a language that are of lesser complexity, and are tailor-made for certain application areas.

A way to define language profiles is to split a language definition into a language core and a set of language modules that can be used as language building blocks. The language core represents an essential subset of the language that each tool for the language should implement. This core is a profile that can be *extended* by language modules, yielding further language profiles that represent well-defined subsets of the language which a tool provider can implement. Defining language modules requires techniques to encapsulate language features in modules, and to compose these modules with the language core.

A language (profile) definition consists of the syntax and the semantics of the language (profile). Modelling languages usually have a *graphical syntax*, defined informally or using a special description language, and possibly a formal *textual syntax*.

[1] In this thesis, we consider only sublanguages that have a dynamic semantics.

Usually, the language definition of a modelling language also includes at least one *abstract syntax* definition, specified using a meta-language like MOF or Meta-IV. The context-sensitive part of the syntax is defined using *well-formedness conditions*. *Transformations* give semantics to parts of a language by mapping them to a subset of the language. Well-formedness conditions and transformations are also referred to as *static semantics*.

The semantics of a language is often defined informally, or using a mathematical formalism. There are several advantages associated with a formal definition of the language semantics. Formal semantics give a precise definition of the language and eliminate the ambiguities that come with an informal language definition. Furthermore, operational mathematical formalisms like Abstract State Machines can be executed and used to generate a compiler and runtime system [58], giving a reference for tool developers.

Defining language profiles, we have to consider both the syntax and the semantics of a language, and their extension or reduction. In order to formulate precise criteria for valid language extension and reduction, we focus on parts of a language definition that are defined formally, which includes the textual and abstract syntax, and for some languages the semantics.

As a shortcut, we refer to both the definition of a sublanguage and the sublanguage itself as a language profile in this thesis.

4.2 Consistency of Profiles

The goal is for a specification expressed in a sublanguage to yield the same behaviour as in all supersets of the sublanguage. In order to accomplish this goal, we need to assure *consistency* between language profiles. Two language profiles are *consistent*, if all specifications accepted by both profiles behave exactly the same way in each profile. A set of language profiles is consistent, if each pair of profiles from the set is consistent.

The dynamic semantics of SDL is defined using Abstract State Machines [42]. Consistency between language profiles defined using ASMs is based on the consistency of corresponding runs of the ASMs. A run of an ASM is a sequence of states, where each subsequent state is the result of firing the programs of one or more agents on the preceding state. For non-deterministic, multi-agent ASMs, the legal behaviour is given by a set of runs, each run in the set describing a possible execution of the system.

Definition. Given a dynamic semantics defined using ASMs, two language profiles are *consistent* if they yield the same set of runs of their respective ASMs for all specifications accepted by both profiles.

In the following sections, we examine consistency for two arbitrary profiles related by the superset relationship: $P_1 \supseteq P_2$. P_2 covers a subset of the specifications that can be expressed in P_1.

4.2.1 Consistency for Sequential Abstract State Machines

The notion of consistency between ASM runs of different language profiles is closely related to the *refinement relation* between runs of ASMs, which has been studied intensively [3, 14, 59]. The refinement relation relates runs of an "abstract" machine A to the runs of a "concrete" machine - or "implementation" - A'. In an m:n refinement (see Figure 4.1), starting from two states related by the refinement relation, both machines end up in states related by the refinement relation after m steps of A and n steps of A' ($m, n \geq 1$). $\approx$ is a data equivalence relation that relates states of A and A' (see [3]).

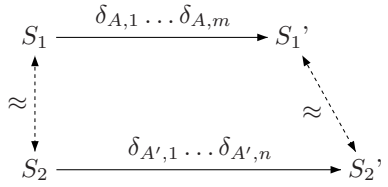


Figure 4.1: m:n refinement relation

A *consistency relation* between two runs is a 1:1 refinement relation, in which each state in the run of A is directly related to a state in the run of A' (see Figure 4.2), and one step of A and A' leads to another pair of related states. The data equivalence relation $\approx$ relates the contents of the locations of A and A' in all points of interest.

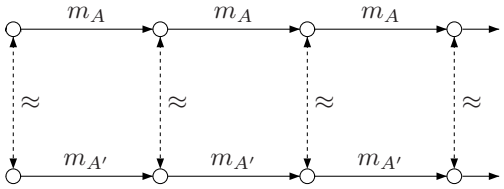


Figure 4.2: Consistency of language profiles (1:1 refinement)

The consistency relation can be relaxed so that the machine A' of the subset can take fewer steps than the machine A of the superset (see Figure 4.3). This is reasonable when a state update in the run of A only updates the state in locations outside the points of interest. For example, the SVM changes into mode *selectSpontaneous* if the predicate *Spontaneous* is true during transition selection. If no spontaneous transition exists in the current states, the mode is immediately changed back without modifying other locations. In an SDL profile without spontaneous transitions, it is reasonable to omit this intermediate state.

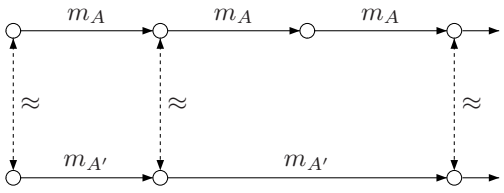


Figure 4.3: Consistency with stuttering steps (m:1 refinement)

4.2.2 Consistency for Distributed Abstract State Machines

Non-determinism and Initial States

Consistency for DASMs has to take *non-determinism* into account. Non-determinism can be introduced explicitly via the choose construct, or through the behaviour of the environment, via monitored and shared functions. A non-deterministic DASM program, fired on a state, can lead to a finite[2] but potentially large number of subsequent states.

To show consistency for two DASMs, runs starting from all *initial states* must be related by the consistency relation. The set of initial states of the machine A' is a subset of the set of initial states of machine A. The states of the smaller machine, usually having less locations, are related to states of the larger machine by the data equivalence relationship $\approx$. For each run starting in an initial state of the smaller machine, a consistent run of the larger machine starting from an equivalent initial state must exist[3].

To show consistency for DASMs, we can extend the definition of consistency from Section 4.2.1 from a consistency relation on simple sequential runs to a consistency relation of algebraic transition systems [15].

Definition. An algebraic transition system $(S, I, \rightarrow)$ is an unlabelled transition system with algebraic structures S as states, initial states $I \subset S$ and a transition relation $\rightarrow \subset S \times S$.

I_A is given by the set of initial states of DASM A, S_A by the states over the vocabulary Voc_A, modulo reserve permutation (states that only differ by a permutation of the elements of the infinite reserve). $\rightarrow_A$ is given as (s, s') for each $s \in S_A$ and each s' resulting from firing an action γ from the action family $\mathrm{NDen}(A, s)$, the set of actions of DASM A in state s (see [28]).

The consistency relation between DASMs A and A' is a bisimulation relationship R between the algebraic transition systems of A and A', with $I_{A'} \subset I_A$, and $\rightarrow_A, \rightarrow_{A'}$

[2] While DASM programs are limited to finite nondeterminism, infinite nondeterminism can be introduced by the environment [28].

[3] Runs starting from a pair of equivalent states does not imply consistency between the runs. Equivalent states can differ in locations outside the points of interest, which can affect locations in points of interest during a run of the ASM.

restricted to the states reachable from $I_{A'}$. The bisimulation R shall only relate states that are data equivalent: $R(s,t) \rightarrow s \approx t$.

Partially-ordered Runs

The execution semantics of Distributed Abstract State Machines (see Section 2.3.4) is based on the notion of partially-ordered runs, in which moves of the agents are ordered by a partial-order relation (see Figure 4.4). The *coherence condition* ensures that every linearisation of the set of moves results in the same state when executed. Given a set of moves M and a partial order $<$, a move $m \in M$ is activated in a well-defined state, resulting from firing all the moves $m' \in M$ with $m' < m$.

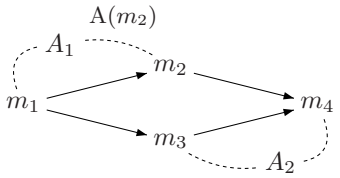


Figure 4.4: Partially ordered run

Two moves m_2, m_3 that are not ordered by $<$ can be fired in parallel or concurrently. Possible executions of a partially-ordered run can differ in the order in which these moves are executed, resulting in differing intermediate states (if any) in these executions (see Figure 4.5). The well-defined states, resulting from firing all moves $\{m' | m' < m\}$ for a move m, are equal for all executions. This follows from the coherence condition.

Figure 4.4 shows a partially-ordered set of moves m_1, m_2 of agent A_1 and m_3, m_4 of agent A_2. After agent A_1 fires m_1, moves m_2 and m_3 can be fired concurrently or in parallel. If the set describes a valid partially-ordered run, move m_4 is fired in the same state regardless of the execution order of m_2 and m_3.

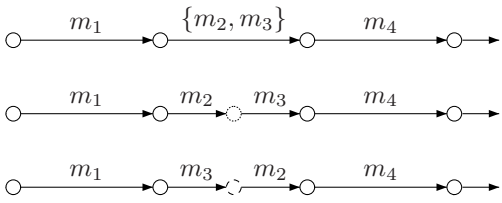


Figure 4.5: Possible executions of a partially-ordered run

Figure 4.5 shows the possible executions for the partially-ordered set in Figure 4.4. The states of the executions are identical, with the exception of a possible intermediate state resulting from firing m_2 before m_3, or vice versa. One possibility is to modify the consistency relation so that the intermediate state is ignored, assuming m_2 and m_3

are fired in parallel. In this example, the number of executions to be related between the abstract and the concrete machine is reduced from three to one.

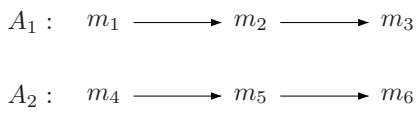


Figure 4.6: Two agents performing causally unrelated moves

Figure 4.6 shows another example, with two agents performing their moves independently with no causal dependency between the moves of the agents. Figure 4.7 shows the possible executions for this partially-ordered run. While the coherence condition ensures that all executions end in the same final state, move m_6 of agent A_2 can be fired in four different states, depending on how many moves were already fired by agent A_1.

Firing all activated moves in parallel, as in the previous example, results in a single execution with four states. However, several executions exist that have only the initial and final state in common with this execution (for example, by firing all moves of A_1 before a move of A_2 is fired). In this example, consistency would only be guaranteed for the initial and final state, resulting in a simple input/output consistency. Therefore, it is not generally possible to reduce the number of executions that have to be related by the consistency relation, by assuming that agents perform their activated rules in parallel. Furthermore, the number of possible executions grows exponentially with the number of agents performing causally unrelated moves. This makes any naive approach to proving consistency for partially-ordered runs infeasible.

Figure 4.6 reflects the situation in the dynamic semantics of SDL[4]. In the formal semantics, ASM agents - representing SDL agents, SDL agent sets and links - perform most of their moves independently, with few causal relationships to moves of other agents. Dependencies between moves occur only when agents communicate using signals.

4.2.3 Verifying Consistency for Distributed Abstract State Machines

In order to verify the consistency of two DASMs, an approach that abstracts from initial states, nondeterminism and execution order of agents is needed. *Induction* is a suitable proof technique for this problem. DASMs A and A' are assumed to start in arbitrary, equivalent initial states. In the *induction step*, it is shown that for two equivalent, reachable states of A and A', firing a valid, arbitrary subset of the activated agents leads to another pair of equivalent states, given equivalent behaviour of the environment (i.e. nondeterminism).

To make this approach feasible, we make two assumptions:

[4]For an example of partially-ordered runs of the dynamic semantics of SDL, see Annex B.

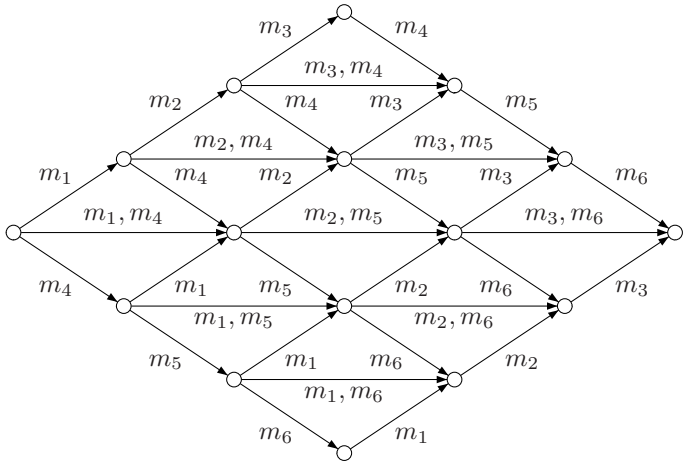


Figure 4.7: Possible executions for causally unrelated moves

- For verification purposes, nondeterminism introduced by the choose construct is either replaced by nondeterminism introduced by the environment, through monitored functions, or corresponding choose-constructs whose guards can be compared exist in both DASMs.
- An isomorphism exists between the sets of agents of A and A'. From this follows that A and A' have the same number of agents, and that each agent of A is uniquely related to an agent of A'. Consistency proofs are therefore restricted to DASMs that are structurally equivalent.

Resolving Non-determinism. Non-determinism is introduced in Abstract State Machines via the choose construct and via monitored functions, which are modified by the environment. Choose constructs can be replaced by equivalent monitored functions, which makes the verification of consistency easier. For each choose construct, a monitored function with a unique name x_n is introduced. This function is constrained by the guard g from the choose rule, so that only elements are returned that satisfy the condition of the choose construct. Within the constraint g, each occurrence of variable x is replaced by the monitored function x_n. In rule R, variable x is bound to the content of x_n in the current state (see Listing 4.1).

```
1 /* choose rule */
2 choose x: g(v_1, ..., v_n) in
3   R
4 endchoose
5
6 /* replacement */
7 monitored x_n: D_1 × ... × D_n → D
8 constraint mode → ∀p_1 ∈ D_1, ..., p_n ∈ D_n.g[x/x_n(p_1, ..., p_n)]
9 let x = x_n(v_1, ..., v_n) in
```

```
10   R
11 endlet
```

Listing 4.1: Replacing choose constructs

The guard of the choose construct can contain locally bound variables that become free variables when placed into a global constraint. These variables must be added as parameters to the monitored function, and bound in the constraint by a universal quantifier. Since the constraint only has to hold in states where the choose construct is evaluated, it is implied by predicate *mode*, which holds only for these states. Otherwise, the constraint may be false in a state that does not lead to the evaluation of the choose construct, making a valid execution of the environment impossible.

In Listing 4.2, choose selects an agent that is directly contained in the currently executing agent set, and is the target of the signal instance *si* under observation. The monitored function *sa_1* gets a signal instance as parameter, and returns for every signal instance the target agent of the signal in the currently executing agent set, while in the mode *DeliverSignal*.

```
1  /* choose rule */
2  choose sa: sa ∈ SDLAgent ∧sa.owner = Self ∧sa.self = si.toArg
3    R
4  endchoose
5
6  /* replacement */
7  monitored sa_1: SignalInst → SDLAgent
8  constraint DeliverSignal → ∀si ∈ SignalInst.sa_1(si) ∈ SDLAgent ∧sa_1(si).owner = Self ∧
       sa_1(si).self = si.toArg
9
10 let sa = sa_1(si) in
11   R
12 endlet
```

Listing 4.2: Application of the choose replacement

A difference between the choose construct and monitored functions is that a monitored function always yields the same value for a given set of parameters in the same state. Executing a given choose rule more than once in the same state, for example when combining it with bounded parallelism (do forall), would yield the same element for each execution with the monitored function, but possibly different elements with the choose construct, if there is true nondeterminism. In general, this problem doesn't occur, since the monitored function will be called with different parameters each time. If this is not the case, an additional parameter that increases with each call has to be introduced. Furthermore, with the new semantics of choose constructs introduced in [28], the approach described above fails if no element matches the constraint. If such cases exist, they must be explicitly handled in the constraint on the monitored function.

An easier approach is to identify corresponding choose constructs of both DASMs and to compare their guards directly. Corresponding choose constructs are choose constructs that are evaluated in the same equivalent states of DASMs A and A'.

In the induction step, the premise is that abstract machines A and A' behave equally with regard to non-determinism. Given equivalent constraints g and g', monitored

functions x_n of A and x_n' of A' are assumed to be equal in every equivalent state of A and A', and corresponding choose constructs are assumed to select equivalent elements.

Agent Execution.

Definition. Let $\mathrm{Ag} = \mathrm{Agents}_A = \mathrm{Agents}_{A'}$ the set of agents of DASMs A and A'. $\mathrm{Active}_a(n) \subset \mathrm{Ag}$ is the set of agents that can perform non-trivial moves in the current state of DASM a (the n-th step of the execution of a). $\mathrm{Exec}_a(n) \subset \mathrm{Active}_a(n)$ is the set of agents firing their moves on the current state, in accordance with the coherence condition of partially ordered runs.

For consistent abstract machines A and A', the set of active agents must be equal in each step of the execution. Otherwise, there is an inconsistent execution resulting from firing only the non-trivial move of an agent activated only in one of the machines, while no move is performed in the other machine, since there is no matching active agent.

To prove consistency, we compare runs of A and A' that execute the same subset of agents in each step of the execution:

$$\forall n \in \mathbb{N}.(\mathrm{Active}_A(n) = \mathrm{Active}_{A'}(n) \wedge \mathrm{Exec}_A(n) = \mathrm{Exec}_{A'}(n))$$

In case the machine representing the superset (A) can perform stuttering steps, we modify this condition to compare the n-th step of A' with the $f(n)$-th step of A, where $f : \mathbb{N} \to \mathbb{N}$ is a strictly increasing function, and $f(0) = 0$.

$$\forall n \in \mathbb{N}.(\mathrm{Active}_A(f(n)) = \mathrm{Active}_{A'}(n) \wedge \mathrm{Exec}_A(f(n)) = \mathrm{Exec}_{A'}(n))$$

A similar approach is taken in [14], where a function $A(k)$ is used to refer to the agent executed in the k-th state of the run. The run of a lower-level machine that implements termination detection is then proven to be a run of a higher level machine, with the same agent execution sequence. This assumes a linear execution of the partially-ordered run. The functions described above extend this notion so that a set of agents can be executed in each step.

Verification. In the induction step, it has to be proven that from any reachable pair of equivalent states, A and A' end up in another pair of equivalent states after firing the moves of any subset of their active agents. In this proof, the assumptions on monitored functions and the execution of agents derived above are used. The induction step consists of a case distinction over all classes of states that lead to the execution of different rules of the ASMs, by satisfying guards of certain if-rules, while not satisfying guards of others. For large DASMs A and A', this case distinction can become very large. Furthermore, a case distinction is needed for every combination of agents that are active in a certain class of states. Since these agents usually work on independent locations, this is generally a trivial step.

4.2.4 Using Structural Information for Verifying Consistency

Generally, we show consistency for DASMs that are related to each other. In the case of language profiles, one DASM is defined for a smaller subset of the language, and can therefore omit certain parts of the specification that the DASM defined for the superset must include, or parts of the specification can be written in a less complex fashion (for example, a simplified transition selection).

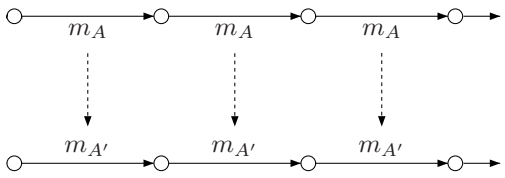


Figure 4.8: Relating update sets of two DASMs

For example, Listing 4.3 shows excerpts of the semantics definition for two language profiles, with language features A and B. The second profile does not support feature B and therefore omits rule R_2 with associated guard g_B. To prove consistency, we show that g_B holds only for specification that use feature B. In that case, specifications using only feature A are not affected by removing (or adding, when extending the language) the second if-statement.

```
1 if (g_A) then                    if (g_A) then
2   R_1                              R_1
3 if (g_B) then
4   R_2
```

Listing 4.3: Structurally related ASMs

4.3 SDL Profiles

SDL-2000, as a complex, sophisticated language with a complete formal semantics [40, 41, 42], defined using ASMs, is well-suited for the definition of language profiles. SDL is used in many different application areas, and systems specified with SDL are deployed on a wide range of target platforms, from servers to microcontrollers. Several tools exist that support a subset of SDL to generate optimised code for specific applications and platforms, for example Telelogic Tau and SDL-RT. This use of SDL profiles is today's state-of-the-practice. However, unlike in UML, their definition is not reflected in the SDL standard.

In this section, we describe several SDL profiles, defined by SDL tools and standards.

4.3.1 Core, Static and Dynamic

The transpiler ConTraST [18] introduces four SDL profiles, as subsets of SDL-96. The smallest profile - included in all other profiles described in this section - is *Core*, which

contains only basic features of SDL. *Static*$_1$, *Static*$_2$ and *Dynamic* extend *Core*, each profile adding additional features to the preceding one, with *Dynamic* being roughly the equivalent of SDL-96.

- SDL profile *Core* is a minimal subset of SDL, supporting only the most elementary features of the language. *Core* retains architectural concepts of SDL such as *package*, *system*, *block* and *process*, plus communication structure using channels. Processes are limited to single instances and contain simple state machines, which are restricted to simple *states*, *input* and *output* of signals. Signals can be addressed explicitly by process name or implicitly by the channel structure. *Core* models structured, asynchronously communicating automata. Excluding support for data makes this subset academical in nature, with the advantage of a concise semantics, efficient execution, and the ability to apply verification techniques, such as model checking.

- Support for data is added in SDL profile *Static*$_1$, making it suitable for practical applications. It supports all data types defined in SDL excluding *string*, *array*, *octetstring* and *powerset*, which require dynamic memory allocation. *Static*$_1$ features commonly used language elements of SDL such as *timer*, *save*, *decision* and *task* symbols. Processes may have multiple instances, but dynamic instantiation is not supported.

- *Static*$_2$ extends *Static*$_1$ with more complex, static features of SDL. On the architectural level, *services* (or their replacement in SDL-2000, *state aggregation*) give additional structure to state machines, while *inheritance* gives better support for reuse. *Static*$_2$ introduces a more expressive transition semantics for SDL, with additions such as *priority input*, *spontaneous transition*, *continuous signal* and *enabling condition*. These additions significantly increase the complexity of transition selection.

- *Dynamic* extends *Static*$_2$ with dynamic aspects of SDL. That includes *procedures*, dynamic *creation* and *termination* of process instances, *context parameters* for process instances, and dynamic data structures (*string*, *array*, *octetstring* and *powerset*). SDL profile *Dynamic* is roughly equivalent to SDL-96 and is completely supported by ConTraST.

Apart from the linear hierarchy of SDL profiles defined by ConTraST, further SDL profiles could have practical relevance. For example, *Static*$_1$, containing the most common static SDL features, could be extended by dynamic SDL features.

4.3.2 Cmicro and Cadvanced

Two SDL profiles are defined by the Telelogic Tau code generators *Cadvanced* and *Cmicro*. While *Cadvanced* supports the majority of SDL-96 features, *Cmicro* has several restrictions in order to produce efficient code for target platforms with limited resources, such as microcontrollers.

Procedures with states have semantics comparable to composite states in SDL-2000, with similar complexity. *Cmicro* only supports *stateless procedures*, leading to flat state machines and simplified transition selection. Transition selection is further simplified by excluding SDL features such as *priority input*, *continuous signal* and *enabling condition*. Processes may not contain *services*.

4.3.3 SDL+/Safire

The language SDL+ [61], defined by the SDL Task Force and implemented in the SAFIRE tool, aims at being the simplest, practically useful subset of SDL. However, SDL+ is not a pure subset of SDL, since it includes several extensions of the language. Most notably, SDL+ includes extensions for testing, and a different handling of gates. As an SDL profile, we consider the reduction of SDL+ to features that exist in SDL[5].

The core of the SDL+ language is formed by a simplified version of SDL statemachines. Statemachines have only simple states (as in SDL-96) and transitions triggered by input signals. Other forms of triggers, such as *priority input*, *spontaneous transition* and *continuous signal*, as well as *enabling condition* and *save* are not supported.

Other features not included are *composite state*, *exception*, *specialisation* and *dynamic instantiation* of agents.

4.3.4 UML Profile for SDL

The UML Profile for SDL ([39], see Section 3.6.3) gives a precise meaning to a subset of UML by mapping elements of UML to elements of SDL with compatible semantics. UML is an extensive language that aims at being a universal language for modelling systems. Therefore, the mapping of the UML subset covers a large portion of the language scope of SDL. However, not every language element of SDL is naturally represented in UML, and therefore the mapping does not cover SDL completely. Therefore, the UML profile implicitly defines an SDL profile.

Language features not covered by this implicitly defined profile are *exception*, *priority input*, *enabling condition* (guards are mapped to decision symbols) and *entry-* and *exit-procedure*. Furthermore, SDL agents of type *system* are not covered, since the UML specification is mapped to an agent of type "block".

4.3.5 Hierarchy of SDL Profiles

Figure 4.9 shows the superset relationship between the language profiles defined in the previous sections. The SDL profiles *Core*, $Static_1$, $Static_2$ and *Dynamic* form a linear hierarchy of profiles, with each profile being a superset of the preceding one. *Dynamic* is included in the UML Profile for SDL, a subset of SDL-2000.

The profiles *Cmicro* and *Cadvanced* include $Static_1$, and $Static_2$ as subsets, respectively. *Cmicro* itself is a subset of *Cadvanced*. SAFIRE, which does not include common SDL features such as *save*, has *Core* as the largest subset. Since it contains dynamic

[5]The complete language of SDL+ can be considered as a profile of the unification of SDL and SDL+

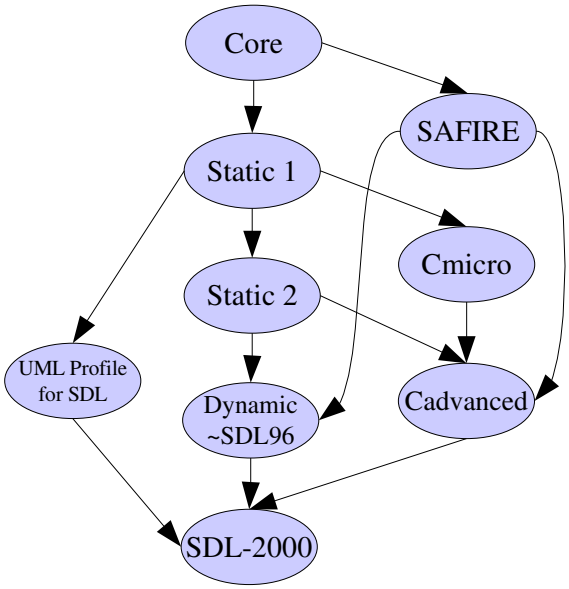


Figure 4.9: Superset relationship between SDL profiles

elements and procedures with states, the smallest supersets of SAFIRE are *Dynamic* and *Cadvanced*.

SDL-2000 is a superset of all SDL-96 based profiles and of the UML Profile for SDL, which covers most of SDL-2000. While some features of SDL-96 were removed in SDL-2000, SDL-2000 is the more powerful language, and these features can be expressed by corresponding SDL-2000 features. For example, services in SDL-96 can be directly mapped to the more powerful state aggregations in SDL-2000.

4.4 Related Work.

A modular language definition can be found in the language specification of UML [52]. The abstract syntax of UML is defined using a meta-model approach, using classes to define language elements and packages to group language elements into medium-grained units. The core of the language is defined by the Kernel package, specifying basic elements of the language such as packages, classes, associations and types. UML defines four compliance levels (L0 - L4, see [52], Section 2.2), defining language subsets by including specific packages only. However, each meta-model class has only an informal description of its semantics, limiting a precise definition of subsets to the language syntax.

UML has a profile mechanism that allows metaclasses from existing metamodels to be extended and adapted, using stereotypes. Semantics and constraints may be

added as long as they are not in conflict with the existing semantics and constraints. The profile mechanism has been used, for example, to define a UML profile for SDL, enabling the use of UML 2.0 as a front-end for SDL-2000 [43]. UML profiles define sublanguages of UML.

Pfahler and Kastens introduce a component-based approach for domain-specific languages [56, 55]. Here, domain and language experts develop a collection of configurable components describing certain properties of a language. The domain expert can define a tailor-made language for a domain by selecting and composing these components. The selections of the domain expert are checked automatically for consistency and completeness - for example, if the domain expert chooses to include while-statements in the language, type boolean is automatically included. In case boolean has been deactivated, the domain expert is notified about the inconsistent selections. The scope of this work is restriced to imperative languages.

Consistency of Abstract State Machines is closely related to refinement of ASMs. In [3] a general notion of ASM refinement is introduced and compared with other refinement approaches, like conservative extension, procedural refinement and data refinement. In [59], a generic framework for the verification of refinements of deterministic ASMs, using generalised forward simulation, is presented. [60] explores the relationship between ASM refinement and data refinement, and behavioural data refinement is shown to be a specific instance of ASM refinement. [14] presents a case study for the refinement and verification of distributed, multi-agent ASMs, using a termination detection algorithm as example.

4.5 Summary and Conclusions

In this chapter, we have introduced the concept of language profiles, and outlined the problems that can occur concerning the static semantics. For SDL, we have identified a hierarchy of profiles, defined by standards, commercial tools and the ConTraST transpiler. We have defined consistency between the behaviour of profiles, given a semantics using abstract transition systems. Finally, we have outlined an approach to verify consistency of profiles defined using Distributed Abstract State Machines.

From this, we conclude that structural information is needed for the practical, efficient verification of consistency for language profiles. Moreover, consistency can be guaranteed by deriving profiles from a common semantics definition, removing only parts of the semantics definition which do not apply to specifications contained in the sublanguage defined by a profile. Ideally, consistency is guaranteed by an extraction process that uses formal criteria to reduce the formal semantics definition for a given language profile. Consistency is either guaranteed directly by the extraction process, or, if heuristics are used, by verifying proof obligations generated by the extraction.

5 Extraction of Language Profiles

In this chapter, we contribute an extraction approach for tailor-made formal language definitions for language profiles. We examine the effect of defining syntactical subsets of a language on the static semantics. For dynamic semantics defined using Abstract State Machines, we formalise our approach for extracting the formal semantics of language profiles from the dynamic semantics definition. This approach is based on identifying invariants for the abstract machine state, derived from the language syntax and the profile definition. Based on these invariants, we extract a formal semantics definition with ASM rules that are never evaluated removed. Results of this work have been published at the SAM workshop [21], the FASE conference [26] and as a technical report [25].

5.1 Motivation

Language profiles define subsets of a language. For the language user, these subsets can be seen as syntactical subsets - the tool that supports the subset only provides certain language features, or language features not included in the profile are rejected by the parser. This is standard practice in the tool industry. For example, the graphical SDL editor of the Telelogic Tau suite only supports drawing SDL specifications with features included in the *Cadvanced* subset. The more restrictive *Cmicro* subset rejects SDL specifications using features not supported by the parser.

An open question is the language definition for language profiles. The state of the practice is to base tools that support a subset of the language on the complete language definition. For example, SDL profiles *Cadvanced* and *Cmicro* are based on the Z.100 standard of SDL, which does not define any compliance levels or valid subsets. Our aim is to provide smaller, tailor-made language definitions for specific language profiles, that are easier to understand, leading to fewer errors and faster development of tool support. Specifically, we want to provide tailor-made formal semantics definitions for language profiles. Formal semantics definitions for complex languages with many features, like SDL, are generally large and take a lot of effort to be understood completely.

5.2 Language Profile Definition

Language profiles characterise subsets of the set of valid language specifications. These subsets are defined syntactically. We propose two ways to define language subsets:

- The syntax of the language defined by a language profile is given explicitly. Here, the complete syntax of the language has to be specified.
- The language profile specifies constraints on the syntax (concrete or abstract), in the same way UML profiles constrain the meta-model of UML using OCL expressions. For example, the constraint

$$\forall s \in \textit{State-node}.(s.\textbf{s-}\textit{Save-signalset}.\textbf{s-}\textit{Signal-identifier}\textbf{-set} = \emptyset)$$

  on the abstract syntax of SDL removes the save feature of SDL syntactically.

In both cases, the effects on the static semantics of the language have to be evaluated (see Section 5.4).

5.3 Approach for Defining Semantics for Language Profiles

One approach to provide a tailor-made formal semantics definition for language profiles is to define the semantics for each profile from scratch. This leads to concise, specialised language definitions for each language profile. However, this approach is infeasible, since defining a formal semantics from scratch takes a lot of effort, even when factoring in reuse from related profile definitions. Changes due to new language versions have to be applied to each profile individually. Furthermore, defining the formal semantics for each profile from scratch makes statements about the consistency of profiles harder to prove, as we have argued in the previous chapter.

For these reasons, we start from a single formal semantics definition, and derive the semantics definition for language profiles. We consider two approaches:

- The *composition* approach, where a tailor-made semantics definition for a language profile is created by composing a language core with predefined language modules.
- The *extraction* approach, where, starting from a formal semantics definition for the complete language, a semantics definition is extracted by removing parts corresponding to language features not included in the profile.

5.3.1 The Composition Approach

Here, language profiles are defined by composing a language core and selected language modules. The *language core* can be understood as the smallest useful subset of the language, for instance, SDL reduced to a set of elementary communicating finite state machines, without any extensions. A *language module* encapsulates a language feature, defining its syntax, semantics, and dependencies to other language modules. For SDL, examples of language modules are timer, exception, save, and inheritance. Note that language core as well as language modules consist of syntax *and* semantics. For the composition to be feasible, it is crucial that the semantics of modules

can be encapsulated and composed with the semantics of the language core and other modules.

The formal semantics definition of SDL is already defined in a modular fashion. Rule macros and derived functions are used extensively to structure the dynamic semantics. The language core is formed by the signal flow model, together with the definition of SDL agents and basic actions such as signal output. Language modules are defined by behaviour primitives, which define the actions of the virtual machine executing SDL specifications. Transition selection is split up into different agent modes for each input kind SDL supports (for example, priority input or spontaneous signals). However, composition of language modules with the language core requires a substantial amount of "glue code", which means that language features have an impact on the complete formal semantics definition in a lot of different places.

From a methodological point of view, the composition approach seems more appealing. However, there is the difficulty of encapsulating the formal semantics of language modules such that composition is supported - while maintaining readability -, which we have not been able to overcome.

5.3.2 The Extraction Approach

Here, language profiles are defined by extracting the profile definition from the complete (formal) language definition. As in the composition approach, language modules, each consisting of a set of language constructs, can be identified. To obtain a particular language profile, these language modules are then removed from the complete language definition. Different from the composition approach, it is not necessary that the semantics of modules can be encapsulated. Instead, it suffices to characterise modules by their language constructs. With this information, it is straightforward to identify corresponding grammar rules, and to reduce or remove them.

To extract the formal semantics of a language profile from the complete formal semantics, given a dynamic semantics defined using Abstract State Machines, we have considered two approaches:

- **ASM rule coverage.** With each language profile, an ASM rule coverage comprising all ASM rules of the dynamic semantics that may be evaluated in some execution of some specification written in that language profile can be associated. While this approach is semantically sound, it is practically infeasible. The concurrent, non-deterministic nature of most modelling languages leads to a very large number of possible executions. Furthermore, the number of specifications that can be written in a given language profile is extremely large, even for small subsets of a language. Therefore, the worst-case complexity of an algorithm for ASM rule coverage is far too high to be of any use for practical purposes.

- **Dead ASM rule recognition.** Instead of computing the ASM rule coverage of a set of specifications, we can develop safe criteria to recognise ASM rules that are never evaluated for a given language profile. For instance, if the SDL language module *timer* is to be removed, we can safely remove all ASM rules that are used

for setting and resetting SDL timers, including the corresponding ASM domains, functions, and relations. It is important here that ASM dead rule recognition works in a conservative way, meaning ASM rules must only be removed if it can be proven that they are never evaluated for a given language profile. The degree of reduction that can be achieved this way thus depends on the completeness of the criteria that can be defined. Unlike the ASM rule coverage approach, dead code recognition is practically feasible. Therefore, we have followed this approach, and will present safe criteria as well as some heuristics below.

5.4 Static Semantics

Language profiles, from the point of view of the language user, can be seen as syntactical subsets of a language. Language features not included in the profile are not supported syntactically, and the formal syntax definition can be reduced accordingly. The context-free syntax of a language is usually defined using a variation of the Backus-Naur Form (BNF). Similar meta-language exist for the definition of the abstract syntax, like Meta-IV for SDL. Figure 5.1 shows the abstract syntax of states for SDL. The non-terminals Spontaneous-transition and Continuous-signal are removed from the definition of the non-terminal State-node, resulting in states without these kind of transitions. The resulting abstract syntax definition is consistent with the original one, since the non-terminals appear as sets, which can be empty. Removing State-name from the syntax rule or making the State-name optional are examples of non-consistent reductions.

State-node ::=(::) State-name
[On-exception]
Save-signalset
Input-node-**set**
Spontaneous-transition-**set**
Continuous-signal-**set**

State-node ::=(::) State-name
[On-exception]
Save-signalset
Input-node-**set**

Figure 5.1: Reduced production rule of the SDL abstract syntax [35]

Consistent Reduction A reduced grammar is *consistent* if all specifications complying to the reduced grammar also comply to the original grammar, that is the language described by reduced grammar L_r is a subset of the language described by the original grammar: $L_r \subset L_o$.

A reduction of a grammar rule leads to a consistent grammar, if for all elements `x` on the right hand side of a production rule the multiplicity of the element is restriced or stays the same. The multiplicity is restriced by raising the lower bound (but not beyond the new upper bound) and lowering the upper bound (but not below the new lower bound). Applied to the abstract grammar of SDL (see Section 3.2.1), this leads to the following rules:

- An optional element $[x]$ has the multiplicity 0..1, and can be omitted (multiplicity 0..0) or be required (x, multiplicity 1..1).
- A set of elements x-**set** has the multiplicity 0..$*$, and can be omitted (multiplicity 0..0) or replaced by an optional element $[x]$.
- A non-empty sequence of elements $x+$ has the multiplicity 1..$*$ and can be replaced by a single element x (multiplicity 1..1).
- A sequence of elements $x*$ has the multiplicity 0..$*$ and can be omitted, or replaced by a single element x, a single optional element $[x]$ or a non-empty sequence $x+$.

The *minimal* consistent reduced grammar is the grammar with the smallest number of elements on the right hand sides of production rules that is consistent with the original grammar.

Well-formedness conditions. Well-formedness conditions are static conditions that a valid specification has to satisfy. They form the context-sensitive part of the language syntax. Reducing the context-free grammar can restrict specifications in a way that certain well-formedness conditions are always violated, and thus no valid specifications exist. Therefore, reduction of the context-free grammar has to take well-formedness conditions into account. For example, the WFC in Figure 5.2 - defined on the abstract syntax of SDL - ensures that two states in the same context do not have the same State-name. Under the assumption that selecting a removed grammar element yields a unique, undefined value, this WFC is always false, since sn.s-State-name $= sn_2$.s-State-name $= \mathit{undefined}$. Note that we need to make this assumption because reducing State-name from State-node is an inconsistent reduction.

$$\forall sn, sn_2 \in \text{State-node} : (sn \neq sn_2) \wedge (sn.parentAS1 = sn_2.parentAS1) \Rightarrow (sn.\text{s-State-name} \neq sn_2.\text{s-State-name})$$

Figure 5.2: Well-formedness condition for state names [41]

In general, there are two ways to resolve this problem:

- Allow only reductions of the formal syntax that do not violate the WFCs. The WFCs introduce dependencies between elements of the syntax that have to be taken into account. Usually, but not generally, these syntax elements belong to the same language feature.
- Reduce the WFCs accordingly, or remove the conflicting condition entirely. This leads to inconsistencies since the reduced language now accepts specifications that are not accepted by the complete language. We have to assure that the dynamic semantics covers these specifications. This approach is unsuitable for language profiles, which we have defined as subsets of a language.

Transformations. Transformations rewrite syntax elements of the specification to core constructs of the language, for example remote procedure calls to local procedure calls and signal exchange in SDL. Transformation rules match certain syntax elements, and become superfluous if these syntax elements are removed. Some rules still apply but must be adapted, in order to fit the reduced syntax. For example, in the static semantics of SDL, the pattern $<$ state $>$ $(< s >^{\cap}$ rest, exc, triggers), with $<$ onexception $>$ exc, must be reduced to $<$ state $>$ $(< s >^{\cap}$ rest, triggers) if exceptions are removed - the transformation rule can't be removed entirely, since it matches states regardless of the existence of exceptions. The variable exc must consequently be removed from the transformation rule - usually, this happens automatically by reducing the abstract syntax expressions on the right hand side of the transformation rule. In other cases, such variables can be replaced by `undefined` (for elements) or $\emptyset$ (for sets).

Reduction of the syntax that affects core constructs of the language affects every construct that is mapped to this core construct. For example, in SDL, reducing syntax for local procedures will affect remote procedures, which are mapped to local procedures. These dependencies between features can be obscure: in SDL, a language profile without exceptions must also exclude remote procedures, since the mapping of remote procedures to local procedures and signal exchange introduces exceptions.

The affected transformation rules must be removed together with corresponding syntax elements, for which no semantics is defined in the reduced language.

Mappings. Mappings are affected in a similar way as transformations. While transformations rewrite syntax elements of a grammar, mappings map between different grammars, for example from abstract grammar AS0 to abstract grammar AS1 in SDL. The same rules as for transformations apply. If an element of the target abstract grammar is removed, all elements from the source abstract grammar that map to this element are affected. If an element of the source abstract grammar is removed, the mapping rule must be reduced accordingly.

In SDL, there is a 1:1 relationship between elements of the abstract grammar AS0, after applying all transformations, and abstract grammar AS1. Consequently, the grammars are not reduced independently, but corresponding elements are removed. This simplifies the reduction of the mapping function. In Figure 5.3, the second parameters of $<$ state $>$ in the AS0 and *State-node* in the AS1 are omitted, both referring to the exception handler for this state node.

$$< \text{state} > (<< \text{statelistitem} > (name, empty) >, \underline{exc}, triggers)$$
$$\Rightarrow \mathbf{mk}\text{–}State\text{-}node(Mapping(name), \underline{Mapping(\overline{exc})}, \ldots, \emptyset, undefined)$$

Figure 5.3: Mapping states from AS0 to AS1 [41]

A formal reduction of the conditions, transformations and reductions depends on the concrete specification technique used to define them. For SDL, which uses denotational aspects of ASMs to define the static semantics, techniques we introduce in Section 5.7 can be applied.

5.5 Reduction Profile

Language profiles characterise subsets of the set of valid specifications, by defining subsets of the concrete and abstract syntax of the language. The abstract syntax of a language influences the dynamic semantics, which is the focus of our work. For SDL, this happens in two ways:

- The abstract syntax yields part of the SVM data structure (ASM signature, see Figure 5.4). For each element of the abstract grammar, a domain of the same name is introduced in the ASM signature. For example, the following non-terminals of the abstract grammar, which are only relevant for SDL specifications with timers, are also domains in the signature of the ASM: *Timer-name*, *Timer-identifier*, *Timer-definition*, *Timer-active-expression*, *Set-node*, and *Reset-node*.
- In the case of SDL actions (for example, assignments, setting timers), a compilation function maps parts of the abstract syntax to domains of the formal semantics definition that form the SVM. For example, the compilation of a *Set-node* in the abstract syntax tree leads to the creation of an element of the domain SET in the ASM signature.

These observations will be used in our approach to identify invariants of the machine state over the run of the dynamic semantics. These invariants are used when extracting a formal semantics definition for a language profile.

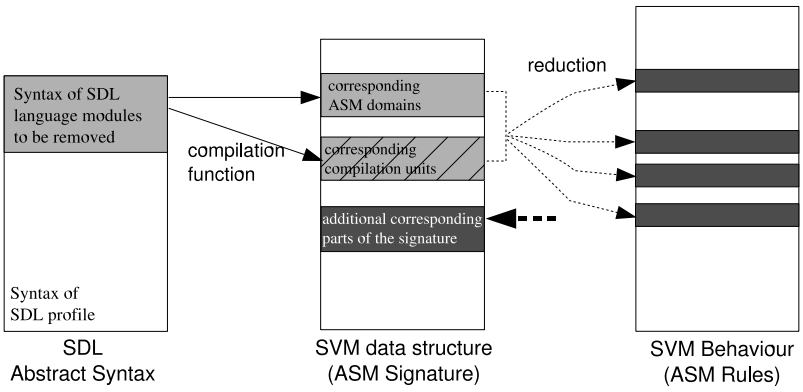


Figure 5.4: Concept of the extraction process (for SDL)

Following the extraction approach, we remove language modules from the formal language definition. Language modules consist of sets of language constructs, and

their corresponding grammar rules. These grammar rules are removed from the formal syntax definition. Furthermore, they form the starting point for the reduction of the formal semantics definition (see Figure 5.4). Starting from the removed parts of the formal syntax definition, we can identify corresponding domains in the ASM signature, as described above. These domains are empty in the initial state of the ASM, and, since they are not modified by the machine, will be empty in all reachable states, too. This observation is fundamental for recognising dead ASM rules of the dynamic semantics.

Apart from domains corresponding to elements of the abstract grammar of a language module, other domains, functions and predicates in the ASM signature correspond to specific language modules. For example, in the dynamic semantics of SDL, *SignalSaved* is a predicate that corresponds to the *save* feature in SDL. If it holds, the signal being examined is not discarded, if no valid transition is found. These elements of the signature are removed in addition to domains corresponding to elements of the abstract grammar. However, we need to prove that these elements are not needed for the given language profile. For example, we can show that *SignalSaved* is always false if *save-signalset* is empty. This is an invariant for all reachable states in the run of the SVM for specifications without save.

```
Reduction−profile ::= Invariant∗
Invariant ::= (Function−name | Relation−name | Domain−name | Rule−name)
              Default−value
Default−value ::= true | false | undefined | ∅ | empty
```

Listing 5.1: Abstract Syntax of Reduction Profiles

In order to perform ASM dead code recognition, we specify all parts of the ASM signature that correspond to language modules not included in the language profile in a *reduction profile* (see Listing 5.1). The reduction profile is a list of domains, functions and predicates from the SVM signature to be removed in the extraction process. This list can be derived from the abstract syntax and, if one exists, the compilation function. However, domain knowledge is still required. We specify a default value for predicates (`true` or `false`), functions (`undefined`, the empty set or the empty sequence), and domains (the empty set). These elements are removed from the formal semantics definition according to a set of extraction rules - mapping between ASM definitions - formally defined in the following sections. The complete set of extraction rules is listed in Annex C.

Default values are invariants specified for elements of the ASM signature. The reduction assumes the content of a domain/function in the reduction profile corresponds to the default value in any state of the dynamic semantics. This is a proof obligation that has to be verified. The reduction profile can also contain names of derived functions and rule macros, used by the SDL-profile tool (see Chapter 6) when iterating the remove step. In the reduction profile provided by the language expert, these elements should only be included if it is certain that calls to these elements are never executed.

Figure 5.5 shows the smallest possible reduction profile corresponding to a language feature. It specifies all grammar elements and predicates used to defer the consumption of input signals.

Save-signalset = ∅ *SignalSaved* = false

Figure 5.5: Reduction Profile for 'Save' Feature

Correct Reduction Profiles. In a correct reduction profile, default values must correspond to the type of the function, predicate or domain in the SVM. Predicates must have default value true or false, and all other functions included in the profile must have type **-set** or include `undefined`. In Listing 5.2, functions *inheritedStateNode* and *stateNodesToBeRefined* can be included in a reduction profile, defaulting to `undefined` and the empty set, respectively. Function *stateName* must not be included in a reduction profile, since it must contain a valid state name.

controlled *inheritedStateNode*: StateNode → [StateNode]
controlled *stateNodesToBeRefined*: SdlAgent → StateNode**-set**

controlled *stateName*: StateNode → *State−name*

Listing 5.2: Functions of the SVM

Given a correct reduction profile, we can exclude certain cases of our dead ASM rule recognition, defined in the following sections, that lead to inconsistent reductions. For correct reduction profiles, proving consistency can be reduced to proving the invariants specified in the reduction profile. Additionally, the reduction covers incorrect reduction profiles, such as assigning the default value *undefined* to a predicate. In most cases, this leads to the extraction of semantics definitions inconsistent with the formal semantics of the complete language.

5.6 Formalisation Signature

We now formalise our approach for extracting the formal semantics of language profiles from the complete language semantics. The formalisation gives a precise definition of the removal process, which leads to deterministic results, and provides the foundation for tool support for the removal process. Finally, a formal definition is necessary in order to make precise statements about the consistency of language profiles. Since the formal syntax definition can be easily defined in a modular fashion, making its reduction straightforward, we focus on the reduction of the formal semantics definition.

For the formal definition of the extraction process, we have decided to use a functional approach, defining functions that recursively map the original formal semantics to the reduced formal semantics. These functions are based on a concrete grammar for Abstract State Machines [20]. The input of the reduction is the formal semantics definition, and a reduction profile r, as described in the previous section.

To formalise the extraction, we define a function $remove_r$, which maps a term from the grammar G of ASMs and a set of variables V - an initially empty set of locally undefined variables from the ASM formal semantics - to a reduced term from the grammar G. Additionally, we introduce three *mutually exclusive* binary predicates,

namely $undefined_r$, $true_r$ and $false_r$. These predicates hold for expressions of the ASM that are determined as true, false or undefined/empty, respectively, in any state, given the information in the reduction profile. The language profile is specified by a globally defined set of elements r from the ASM signature of the formal semantics definition, annotated by default values `true` and `false` for predicates. This set represents the elements to be removed from the formal semantics definition, and is therefore called the *reduction profile*. For all elements in the reduction profile, $undefined_r$ ($true_r$ or $false_r$ for predicates) holds.

$$\begin{aligned} remove_r : G \times V &\rightarrow G \\ undefined_r : G \times V &\rightarrow \mathsf{Boolean} \\ true_r : G \times V &\rightarrow \mathsf{Boolean} \\ false_r : G \times V &\rightarrow \mathsf{Boolean} \end{aligned}$$

The $remove_r$ function is defined on all elements of the grammar G. Predicates $true_r$ and $false_r$ are explicitly defined on boolean and first-order logic expressions. On all other elements of G, the predicates do not hold.

The function $remove_r$ is defined recursively - a given term is mapped to a new term by applying the mapping defined by $remove_r$ to the subterms. In case none of the predicates $undefined_r$, $true_r$ and $false_r$ holds, the current term is not reduced any further. This assures in particular that $remove_r$ corresponds to the identical mapping if the signature of the ASM is not reduced. In other cases, subterms can be replaced or omitted depending on which of the predicates hold.

In the following sections, we omit the index r from *remove* and the predicates.

5.7 Formal Reduction of ASMs

In this section, we define parts of the formal reduction of ASMs. The complete definition can be found in Annex C.

5.7.1 Formal Reduction of ASM Definitions

This section describes the *remove* function for ASM definitions, namely domain, function and rule macro definitions. The *remove* function does not remove a definition completely. If, by the reduction process, the definition becomes trivial or is not referenced anymore, it is removed in a subsequent cleanup step. The following variables are used to define removal:

$D, D_1, D_2 \in$ domain, dn $\in$ `DomainName`, exp $\in$ formula, $R, R_1, R_2 \in$ rule, ps $\in$ paramSeq

Domain definitions are not affected by removal, unless they are derived definitions. For derived domain definitions, removal continues with the domain expression that defines the derived domain.

$remove(\text{mode } \textbf{domain } \text{dn}, \mathcal{V}) = \text{mode } \textbf{domain } \text{dn}$
$remove(\textbf{domain } \text{dn}, \mathcal{V}) = \textbf{domain } \text{dn}$
$remove(\text{dn} =_{def} D, \mathcal{V}) = \text{dn} =_{def} remove(D, \mathcal{V})$

For function definitions, removal continues with the function signature to remove any undefined domains. For derived functions, removal also continues with the formula that defines the function, taking into account all formal parameters that were removed. After removal, unreferenced functions or functions with no target domain can be removed in a subsequent step.

$remove(\text{mode } f \text{ ':' } D_1 \to D_2, \mathcal{V}) = \text{mode } f \text{ ':' } remove(D_1, \mathcal{V}) \to remove(D_2, \mathcal{V})$
$remove(\text{mode } f \text{ ':' } \to D_2, \mathcal{V}) = \text{mode } f \text{ ':' } \to remove(D_2, \mathcal{V})$
$remove(f \text{ ':' } D_1 \to D_2, \mathcal{V}) = f \text{ ':' } remove(D_1, \mathcal{V}) \to remove(D_2, \mathcal{V})$
$remove(f \text{ ':' } \to D_2, \mathcal{V}) = f \text{ ':' } \to remove(D_2, \mathcal{V})$

$remove(f \text{ ':' } D =_{\text{def}} \text{exp}, \mathcal{V}) = f \text{ ':' } remove(D, \mathcal{V}) =_{\text{def}} remove(\text{exp}, \mathcal{V})$
$remove(f(\text{ps}) \text{ ':' } D =_{\text{def}} \text{exp}, \mathcal{V}) =$
$\quad f(remove(\text{ps}, \mathcal{V})) \text{ ':' } remove(D, \mathcal{V}) =_{\text{def}} remove(\text{exp}, \mathcal{V} \cup remfpar(\text{ps}))$

A macro definition consists of a rule name, a sequence of formal parameters and a rule body. Domains of formal parameters may be undefined, and the corresponding parameters must be removed. Removal continues with the rule body and the undefined formal parameters added to the list of undefined variables.

$remove(\texttt{RuleName} \equiv R, \mathcal{V}) = \texttt{RuleName} \equiv remove(R, \mathcal{V})$
$remove(\texttt{RuleName}(\text{ps}) \equiv R, \mathcal{V}) =$
$\quad \texttt{RuleName}(remove(\text{ps}, \mathcal{V})) \equiv remove(R, \mathcal{V} \cup remfpar(\text{ps}))$

Removal on constraints equates to removal on the constraint formula. Removal of program definitions continues with removal of the rule body of the program.

$remove(\textbf{constraint } \text{exp}, \mathcal{V}) = \textbf{constraint } remove(\text{exp}, \mathcal{V})$
$remove(\textbf{initially } \text{exp}, \mathcal{V}) = \textbf{initially } remove(\text{exp}, \mathcal{V})$

$remove(\texttt{ProgramName} \text{ ':' } R, \mathcal{V}) = \texttt{ProgramName} \text{ ':' } remove(R, \mathcal{V})$
$remove(\texttt{ProgramName} \text{ ':'}, \mathcal{V}) = \texttt{ProgramName} \text{ ':'}$

5.7.2 Macros, Functions and Parameters

This section describes the removal of formal parameters of rule macros and functions, and the removal of corresponding parameters from calls to these macros and functions. The following variables are used in this section, in addition to the ones introduced above:

fcs $\in$ `formulaCommaSeq`, $n, p \in \mathbb{N}$

Formal parameters are removed from a list of formal parameters if *undefined* holds for their domain (the type). Removal of formal parameters starts with the rightmost parameter.

$remove(\text{ps }\text{','}\ x\ \text{':'}\ D, \mathcal{V}) =$

$remove(\text{ps}, \mathcal{V})$	iff	$undefined(D)$
$remove(\text{ps}, \mathcal{V})\ \text{','}\ x\ \text{':'}\ remove(D, \mathcal{V})$		else

$remove(x\ \text{':'}\ D, \mathcal{V}) =$

"	iff	$undefined(D)$
$x\ \text{':'}\ remove(D, \mathcal{V})$		else

The function *numfpar* counts the number of formal parameters in a formal parameter sequence. The function is used when removing parameters from a parameter sequence (see below).

$numfpar(\text{fcs }\text{','}\ \text{exp}) = numfpar(\text{fcs}) + 1$
$numfpar(\text{exp}) = 1$
$numfpar(\text{ps }\text{','}\ x\ \text{':'}\ D) = numfpar(\text{ps}) + 1$
$numfpar(x\ \text{':'}\ D) = 1$

The function *remfpar* returns a set of names of formal parameters. The set includes all names of a formal parameter sequence for which *undefined* holds for the corresponding domain.

$remfpar(\text{ps }\text{','}\ x\ \text{':'}\ D) =$

$\{x\} \cup remfpar(\text{ps})$	iff	$undefined(D)$
$remfpar(\text{ps})$		else

$remfpar(x\ \text{':'}\ D) =$

$\{x\}$	iff	$undefined(D)$
$\{\}$		else

Formal parameters removed in a macro definition must be removed from the argument lists of macro calls. *count* assigns a code to a macro that describes which parameters have been removed. The code function gets a list of parameters and a number n (initially the number of arguments minus one) as arguments. If the domain of the rightmost argument is undefined, 2^n is added to the code of the remaining parameters with the number $n-1$. E.g. for a sequence of four parameters, with the first and the third undefined, the code is $2^0 + 2^2 = 5$.

$code(\text{ps }\text{','}\ x\ \text{':'}\ D, n) =$

$code(\text{ps}, n-1) + 2^n$	iff	$undefined(D)$
$code(\text{ps}, n-1)$		else

$code(x\ \text{':'}\ D, 0) =$

1	iff	$undefined(D)$
0		else

$count(\texttt{MacroName}) = code(\text{ps}, numfpar(\text{ps}) - 1)$

The function *removepar* removes arguments from a macro call corresponding to undefined formal parameters in the macro definition. The number n corresponds the position of the argument (initially the number of arguments minus one), the number p to the code of the macro definition. If p is larger than 2^n, the argument is removed and removal is continued with the remaining parameters.

$$
\begin{array}{lll}
\mathit{removepar}(\text{fcs ',' exp}, n, p) = & & \\
\quad \mathit{removepar}(\text{fcs}, n-1, p-2^n) & \text{iff} & p - 2^n > 0 \\
\quad \mathit{removepar}(\text{fcs}, n-1, p) \text{ ',' exp} & & \text{else} \\
 & & \\
\mathit{removepar}(\text{exp}, n, p) = & & \\
\quad \text{''} & \text{iff} & p = 2^n \ // \text{ n should be 0} \\
\quad \text{exp} & & \text{else}
\end{array}
$$

A sequence of formulas is undefined if each formula in the sequence is undefined.

$$\mathit{undefined}(\text{fcs}, \text{exp}, \mathcal{V}) \quad \text{iff} \quad \mathit{undefined}(\text{fcs}, \mathcal{V}) \vee \mathit{undefined}(\text{exp}, \mathcal{V})$$

5.7.3 Formal Reduction of ASM Rules

Rules specify transitions between states of the ASM. The basic rule is the *update rule*, which updates a location of the state to a new value. Alltogether, there are seven kinds of rules for ASMs, for all of which we have formalised the reduction.

The left hand side of an update rule specifies a location of the ASM. The location consists of a function f from the ASM signature and a tuple of elements fcs. If *undefined* holds for either, the location lies outside the scope of the reduced ASM, and the update rule is omitted. If *undefined* holds for the expression on the right hand side of the update rule, we remove the update rule, retaining the previous value of the location.

$$
\begin{array}{lll}
\mathit{remove}(f(\text{fcs}) := \text{exp}, \mathcal{V}) = & & \\
\quad \text{skip} & \text{iff} & \mathit{undefined}(f, \mathcal{V}) \vee \mathit{undefined}(\text{fcs}, \mathcal{V}) \vee \\
 & & \mathit{undefined}(\text{exp}, \mathcal{V}) \\
\quad f(\mathit{remove}(\text{fcs}, \mathcal{V})) := \mathit{remove}(\text{exp}, \mathcal{V}) & & \text{else}
\end{array}
$$

The mapping of the **if**-rule depends on which predicate holds for the guard *exp* of the rule. If the guard always evaluates to `true` (`false`), the **if**-rule can be omitted, and removal continues with subrule R_1 (R_2). If the guard is undefined, the rule is syntactically incorrect, and should not be reachable[1]. Since a valid **if**-rule can not be constructed with an invalid guard, we map this case to the skip-rule. If none of the predicates hold, the removal is applied recursively to the guard and the subrules of the **if**-rule, leaving the rule itself intact.

[1] This is a proof obligation that we have to verify manually. However, so far this has only occurred in very few cases, which turned out to be errors in the reduction profile.

remove(**if** exp **then** R_1 **else** R_2 **endif**, $\mathcal{V}$) =

remove$(R_1, \mathcal{V})$	iff	*true*(exp, $\mathcal{V}$)
remove$(R_2, \mathcal{V})$	iff	*false*(exp, $\mathcal{V}$)
skip	iff	*undefined*(exp, $\mathcal{V}$)
if *remove*(exp, $\mathcal{V}$) **then** *remove*$(R_1, \mathcal{V})$		else
else *remove*$(R_2, \mathcal{V})$ **endif**		

The **let**-rule is a shortcut that binds the evaluation result of an expression in the current state to a variable, which can be used inside the **let**-rule. In case the expression exp is undefined, so is the variable x. The result is the mapping of the contained rule R, with the variable x included in the set of locally undefined names $\mathcal{V}$. The result of the removal is the same as if the expression *exp* had been used directly in the rule R instead of the variable x.

remove(**let** $x : D =$ exp **in** R **endlet**, $\mathcal{V}$) =

remove$(R, \mathcal{V} \cup \{x\})$	iff	*undefined*(exp, $\mathcal{V}$) $\vee$ *undefined*(D)
let x : *remove*$(D, \mathcal{V})$ = *remove*$(exp, \mathcal{V})$ **in**		else
remove$(R, \mathcal{V})$ **endlet**		

The **extend**-rule dynamically imports a fresh ASM element from the reserve (an infinite store of unused ASM elements), binding it to a variable x in the context of the subrule R and including it in the ASM domain dn (given by name). In case the domain name dn is undefined, i.e. has been removed from the ASM signature, the **extend**-rule can be omitted, since elements of domain dn belong to a removed feature. However, the subrule R might still contain parts not related to this feature - although it would be a better style to move these parts outside the **extend**-rule. Therefore, the subrule is not omitted by default, but replaced with its mapping by the remove function, including the now unbound variable x in the set of locally undefined variables. This leads to all occurrences of x being removed from the rule R.

remove(**extend** dn **with** x R **endextend**, $\mathcal{V}$) =

remove$(R, \mathcal{V} \cup \{x\})$	iff	*undefined*(dn, $\mathcal{V}$)
extend dn **with** x *remove*$(R, \mathcal{V})$ **endextend**		else

The **choose**-rule nondeterministically takes an element from the finite set defined by the constraint exp and binds it to the variable x. If no element satisfies the constraint, as in the case where *false* holds, choose is equivalent to skip [28]. Futhermore, if *undefined* holds for the constraint, we assume that no element matches it. If *true* holds for the constraint, the **choose**-rule is invalid since it ranges over a potentially infinite set.

remove(**choose** x : exp R **endchoose**, $\mathcal{V}$) =

skip	iff	*false*(exp, $\mathcal{V}$) $\vee$ *true*(exp, $\mathcal{V}$) $\vee$ *undefined*(exp, $\mathcal{V}$)
choose x : *remove*(exp, $\mathcal{V}$) *remove*$(R, \mathcal{V})$		else
endchoose		

Using typed ASMs, it is sensible to restrict the element x to a domain D as the type of x. In the formal semantics of SDL-2000, all constraints of **choose**-rules have the form "$x \in D \wedge$ *constraint*" (*constraint* being optional). From the definition of *true* and *undefined* on expressions (see Section 5.7.5) follows that these predicates will not hold for a constraint of this form. Tables 5.3 and 5.4 show that only *false* can hold for $x \in D$ and $x \in D \wedge$ *constraint*.

The **do forall**-rule performs a parallel update of the state, firing the rule R with x bound to the element a, for all $a \in \{x \mid \text{exp}\}$, $\{x \mid \text{exp}\}$ being a finite set. Removal follows the same principles as with **choose**, as both **choose**-rule and **do forall**-rule use elements from a finite set defined by a constraint. If *false* holds for the constraint, the rule is equivalent to skip. As described above, predicates *true* and *undefined* do not hold if exp has the form "$x \in D \wedge$ *constraint*".

remove(**do forall** x ':' exp R **enddo**, $\mathcal{V}$) =

skip	iff	*false*(exp, $\mathcal{V}$) $\vee$ *true*(exp, $\mathcal{V}$) $\vee$ *undefined*(exp, $\mathcal{V}$)
do forall x ':' *remove*(exp, $\mathcal{V}$) *remove*(R, $\mathcal{V}$) **enddo**	else	

Rule blocks in ASMs are fired in parallel. A sequence of rule blocks is broken down to the mappings of the sub-rule blocks. This may result in a sequence of skip-rules which can be reduced to a single skip. However, this is not part of the remove mapping, but is done in a subsequent cleanup step.

remove(R_1 R, $\mathcal{V}$) = *remove*(R_1, $\mathcal{V}$) *remove*(R, $\mathcal{V}$)

5.7.4 Formal Reduction of ASM Domains

This section defines the *remove* function for expressions describing ASM domains, e.g. union or tuple domains. The following variables are used, in addition to the variables defined in previous sections:

$s, s_1, s_2 \in$ simpledomain, $t \in$ tupledomain, $u \in$ uniondomain, ics $\in$ itemCommaSeq.

Removal is applied to domains in expressions that contain them, in formal parameter lists and in function declarations. A domain is removed if it is undefined. A union domain is removed if all of the subdomains are undefined (i.e., evaluating to the empty set), otherwise only the undefined subdomains are removed. A tuple domain is removed if any of the subdomains is undefined.

remove($s_1 \times s_2$, $\mathcal{V}$) =

nodomain	iff	*undefined*($s_1 \times s_2$, $\mathcal{V}$)
remove(s_1, $\mathcal{V}$) $\times$ *remove*(s_2, $\mathcal{V}$)	else	

remove($t \times s$, $\mathcal{V}$) =

nodomain	iff	*undefined*($t \times s$, $\mathcal{V}$)
remove(t, $\mathcal{V}$) $\times$ *remove*(s, $\mathcal{V}$)	else	

$remove(\text{'(' ')'}, \mathcal{V}) = \text{'(' ')'}$

$remove(s_1 \cup s_2, \mathcal{V}) =$

nodomain	iff	$undefined(s_1 \cup s_2, \mathcal{V})$
$remove(s_1, \mathcal{V})$	iff	$undefined(s_2, \mathcal{V})$
$remove(s_2, \mathcal{V})$	iff	$undefined(s_1, \mathcal{V})$
$remove(s_1, \mathcal{V}) \cup remove(s_2, \mathcal{V})$		else

$remove(u \cup s, \mathcal{V}) =$

nodomain	iff	$undefined(u \cup s, \mathcal{V})$
$remove(u, \mathcal{V})$	iff	$undefined(s, \mathcal{V})$
$remove(s, \mathcal{V})$	iff	$undefined(u, \mathcal{V})$
$remove(u, \mathcal{V}) \cup remove(s, \mathcal{V})$		else

The predicate undefined on domains specifies if a domain expression is undefined (that is, empty), given the basic domains that have been defined as being empty. A domain expression is undefined if the domain name it comtains is undefined. In case of union of two domains, both domains must be undefined - if only one is undefined, a valid domain definition can be extracted by removing the undefined domain from the expression. A tuple domain is undefined if one of its subdomains is undefined.

$$\begin{aligned} undefined(s_1 \times s_2, \mathcal{V}) &\text{ iff } undefined(s_1, \mathcal{V}) \vee undefined(s_2, \mathcal{V}) \\ undefined(t \times s, \mathcal{V}) &\text{ iff } undefined(t, \mathcal{V}) \vee undefined(s, \mathcal{V}) \end{aligned}$$

$$\begin{aligned} undefined(s_1 \cup s_2, \mathcal{V}) &\text{ iff } undefined(s_1, \mathcal{V}) \wedge undefined(s_2, \mathcal{V}) \\ undefined(u \cup s, \mathcal{V}) &\text{ iff } undefined(u, \mathcal{V}) \wedge undefined(s, \mathcal{V}) \end{aligned}$$

5.7.5 Formal Reduction of ASM Expressions

Expressions are terms over the signature of the SVM. Additionally, ASMs include common mathematical structures like boolean algebra, or natural numbers. Our formal reduction covers all operations defined in [20]. In the truth tables defined in this section, we use the following shortcuts:

T	Predicate *true* holds
F	Predicate *false* holds
U	Predicate *undefined* holds
-	$\neg$T $\wedge \neg$F $\wedge \neg$U

Furthermore, we use the variables $e, e_1, e_2, e_3 \in$ formula, nseq $\in$ nameCommaSeq and pcs $\in$ primaryCommaSeq.

Boolean Operators

Boolean Operators take boolean expressions as arguments, therefore the predicates *true*, *false* and *undefined* apply. With binary boolean operators, we have to consider sixteen different combinations of predicates holding for the subexpressions - four for each subexpression. In order to improve readability, we combine the definitions of *true*, *false*, *undefined* and *remove* for boolean operators in a four-valued truth table. Valid boolean expressions always evaluate to either `true` or `false`. Therefore, it is undesirable that the predicate *undefined* holds for such an expression. However, this can not be avoided in every case.

We define truth tables for all boolean operators from the concrete syntax of ASMs: negation ($\neg$), disjunction ($\vee$), conjunction ($\wedge$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). In order to ensure consistent results, we derive the definition of conjunction, implication and equivalence from the definitions of negation and disjunction. For the predicates *true* and *false*, the subtables match the truth tables for the corresponding boolean operators with the truth values `true` and `false`, respectively. If all subexpressions of the operator are undefined, so is the composite expression.

The truth table for the negation directly follows from these considerations (see Table 5.1). In case no predicate holds for the boolean expression e_1 (-), removal maps to the original term, with removal applied to the subexpression e_1.

e_1	$\neg$	T	F	U	-
		F	T	U	$\neg e_1$

Table 5.1: Truth table for negation

If *true* holds for one of the subexpressions e_1 or e_2, *true* holds for $e_1 \vee e_2$. If *undefined* holds for one of the subexpressions, it is omitted and the result depends exclusively on the other subexpression. If *false* holds for one of the subexpressions, the subexpression is omitted but can still influence the final result (as in the case *false* and *undefined*).

	e_2				
e_1	$\vee$	T	F	U	-
	T	T	T	T	T
	F	T	F	F	e_1
	U	T	F	U	e_1
	-	T	e_2	e_2	$e_1 \vee e_2$

Table 5.2: Truth table for disjunction

We can define the other boolean operators with the operators $\vee$ and $\neg$ defined above. For example, $e_1 \wedge e_2$ is defined as $\neg(\neg e_1 \vee \neg e_2)$, the truth table is derived accordingly (see Table 5.3).

e_1	e_2 $\wedge$	T	F	U	-
	T	T	F	T	e_1
	F	F	F	F	F
	U	T	F	U	e_1
	-	e_2	F	e_2	$e_1 \wedge e_2$

Table 5.3: Derived truth table for conjunction

Relational Operators

Binary relational operators form boolean expressions, comparing two subexpressions. Unlike boolean or arithmetical operators, it is not possible to omit the operator and retain one of the subexpressions, since the subexpressions are not boolean expressions. In our approach, we do not evaluate the relational operators $>, <, \geq, \leq$ in respect to their truth-value. Therefore, these expressions are undefined if one of their subexpressions is undefined. Removal is defined accordingly.

A special relational operator is the element-of operator $e_1 \in e_2$, where e_1 denotes an element and e_2 denotes a set. The element-of operator appears frequently in the guard of **if**-rules. The expression e_2, denoting a set, is interpreted as the empty set if *undefined* holds. Therefore, *false* (*true*) holds for the element-of (not element-of) expression if e_2 is undefined. Likewise, an undefined expression should not be an element of any set. Note that according to this definition, *undefined* can not hold for an element-of expression.

e_1	e_2 $\in$	U	-
	U	F	F
	-	F	-

e_1	e_2 $\notin$	U	-
	U	T	T
	-	T	-

Table 5.4: Truth table for element-of operator

The equality operator is as significant as the element-of operator. For the equality operator, we take three special ASM elements into account - the element `undefined`, the empty set ($\emptyset$) and the empty sequence (*empty*). We interprete an undefined expression e as `undefined`, empty set or empty sequence, depending on the context. Therefore, *true* holds if an undefined e is equated with one of these elements. Likewise, *false* holds if an undefined expression is said to be unequal to one of these elements. Note that equality is symmetric, so if *true* holds for $e =$ `undefined`, it also holds for `undefined` $= e$.

Excluding the cases addressed above, two expressions should never be equal if one expression is undefined and the other expression is not.

$$
\begin{array}{rll}
\mathit{true}(e = \texttt{undefined}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \texttt{undefined}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{true}(e = \emptyset, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \emptyset, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{true}(e = \mathit{empty}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \mathit{empty}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e_1 = e_2, \mathcal{V}) & \text{iff} & \neg \mathit{true}(e_1 = e_2, \mathcal{V}) \wedge \\
 & & (\mathit{undefined}(e_1, \mathcal{V}) \wedge \neg \mathit{undefined}(e_2, \mathcal{V}) \vee \\
 & & \neg \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V})) \\
\mathit{true}(e_1 \neq e_2, \mathcal{V}) & \text{iff} & \neg \mathit{false}(e_1 \neq e_2, \mathcal{V}) \wedge \\
 & & (\mathit{undefined}(e_1, \mathcal{V}) \wedge \neg \mathit{undefined}(e_2, \mathcal{V}) \vee \\
 & & \neg \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}))
\end{array}
$$

Quantification

Quantification consists of two subexpressions - the expression e_1 representing the set of elements in the range of the quantification, and the boolean-valued expression e_2 as the predicate. In the context of quantification, we interprete e_1 as the empty set if the predicate *undefined* holds.

$$Qx \in e_1 : e_2,\ Q \in \{\forall, \exists, \exists_1\}$$

Quantification over an empty set (i.e., *undefined* holds for e_1) is always true in case of universal quantification, and always false in case of existential quantification. Furthermore, universal quantification is always true of the boolean expression e_2 is always true, and existential quantification is always false iff e_2 is always false. This leads to the following definitions of the *remove* function and respective predicates (see Tables 5.5, 5.6).

	e_1				
e_2	$\forall$	T	F	U	-
	U	T	T	T	T
	-	T	-	U	-

Table 5.5: Truth table for universal quantification

$$
\begin{array}{lll}
\mathit{remove}(\forall \text{nseq} \in e_1\text{`} : \text{`}e_2, \mathcal{V}) = & & \\
\quad \texttt{true} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{true}(e_2, \mathcal{V}) \\
\quad \texttt{undefined} & \text{iff} & \neg \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V})
\end{array}
$$

$\forall \text{nseq} \in remove(e_1, \mathcal{V})\text{`} : \text{`}remove(e_2, \mathcal{V})$ else

e_2	e_1 $\exists$	T	F	U	-
	U	F	F	F	F
	-	-	F	U	-

Table 5.6: Truth table for existential quantification

$remove(\exists \text{nseq} \in e_1\text{`} : \text{`}e_2, \mathcal{V}) =$

`false`	iff	$undefined(e_1, \mathcal{V}) \vee false(e_2, \mathcal{V})$
`undefined`	iff	$\neg undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V})$
$\forall \text{nseq} \in remove(e_1, \mathcal{V})\text{`} : \text{`}remove(e_2, \mathcal{V})$		else

Arithmetic Operators

In order to define removal for arithmetic operators precisely, the reduction profile needs to be extended so that static values, constants or expressions can be defined as default values for functions that produce natural or real numbers. Here, we define a simple reduction that assumes an undefined subexpression of an arithmetic operation evaluates to zero, which is suitable for the reduction of the dynamic semantics of SDL.

$remove(-e, \mathcal{V}) =$

`undefined`	iff	$undefined(e, \mathcal{V})$
$-remove(e, \mathcal{V})$		else

$remove(e_1 + e_2, \mathcal{V}) =$

`undefined`	iff	$undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V})$	iff	$undefined(e_2, \mathcal{V})$
$remove(e_2, \mathcal{V})$	iff	$undefined(e_1, \mathcal{V})$
$remove(e_1, \mathcal{V}) + remove(e_2, \mathcal{V})$		else

$remove(e_1 * e_2, \mathcal{V}) =$

`undefined`	iff	$undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V}) * remove(e_2, \mathcal{V})$		else

$$
\begin{aligned}
undefined(-e, \mathcal{V}) &\text{ iff } undefined(e, \mathcal{V}) \\
undefined(e_1 + e_2, \mathcal{V}) &\text{ iff } undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V}) \\
undefined(e_1 * e_2, \mathcal{V}) &\text{ iff } undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})
\end{aligned}
$$

Sets and Sequences

Set (or sequence) composition constructs a set (sequence) by applying expression e_1 to elements from e_2, an expression describing a set (sequence), for which the boolean expression e_3 holds. If expression e_1 or e_3 are undefined, so is the set/sequence, resulting in a malformed specification. If *undefined* holds for e_2 (i.e., e_2 is interpreted as empty) or *false* holds for e_3, the resulting sequence is empty, and the resulting set is the empty set.

$$
\begin{array}{lll}
remove(\{e_1 \mid x \in e_2 : e_3\}, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & undefined(e_1, \mathcal{V}) \vee undefined(e_3, \mathcal{V}) \\
\quad \emptyset & \text{iff} & (false(e_3, \mathcal{V}) \vee undefined(e_2, \mathcal{V})) \\
 & & \wedge \neg undefined(e_1, \mathcal{V}) \\
\quad \{remove(e_1, \mathcal{V}) \mid x \in remove(e_2, \mathcal{V}) & & \text{else} \\
\quad : remove(e_3, \mathcal{V})\} & &
\end{array}
$$

$$
undefined(\{e_1 \mid x \in e_2 : e_3\}, \mathcal{V}) \quad \text{iff} \quad (false(e_3, \mathcal{V}) \vee undefined(e_2, \mathcal{V})) \wedge \neg undefined(e_1, \mathcal{V})
$$

Function and Macro Calls

A macro call is removed if one of the parameters passed is undefined, or the macro name itself has been marked as undefined. Otherwise, parameters that correspond to removed formal parameters from the rule macro definition are removed from the parameter list of the macro call. This is done with the function *remfpar* defined in Section 5.7.2. *count* is a natural number that holds the information which formal parameters were removed.

$$
\begin{array}{lll}
remove(\texttt{MkName}\underline{()}, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & undefined(\texttt{MkName}) \\
\quad \texttt{MkName}\underline{()} & & \text{else} \\
remove(\texttt{MkName}\underline{(}\text{fcs}\underline{)}, \mathcal{V}) = & & \\
\quad \text{skip} & \text{iff} & undefined(\texttt{MkName}) \vee undefined(\text{fcs}, \mathcal{V}) \\
\quad \texttt{MkName}\underline{(}removepar(\text{fcs}, numfpar(fcs) - 1, count(\texttt{MkName}))\underline{)} & & \\
 & & \text{else}
\end{array}
$$

$$
\begin{array}{rlr}
undefined(\texttt{MkName}\underline{()}, \mathcal{V}) & \text{iff} & undefined(\texttt{MkName}) \\
undefined(\texttt{MkName}\underline{(}\text{fcs}\underline{)}, \mathcal{V}) & \text{iff} & undefined(\texttt{MkName}) \vee undefined(\text{fcs}, \mathcal{V})
\end{array}
$$

Function calls are similar to rule macro calls. A function call can either refer to a location of the ASM, or a derived function defining an expression. Removal for

function calls is identical to removal for macro calls, removing the call if either the parameters or the function itself are undefined.

$$
\begin{array}{lll}
remove(f, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & undefined(f) \\
\quad f & & \text{else} \\
remove(exp.f, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & undefined(f) \vee undefined(exp, \mathcal{V}) \\
\quad remove(exp, \mathcal{V}).f & & \text{else} \\
remove(f\underline{(}\text{fcs}\underline{)}, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & undefined(f) \vee undefined(\text{fcs}, \mathcal{V}) \\
\quad f\underline{(}removepar(\text{fcs}, numfpar(fcs) - 1, count(f))\underline{)} & & \\
 & & \text{else}
\end{array}
$$

$$
\begin{array}{rll}
undefined(f\underline{(}\text{fcs}\underline{)}, \mathcal{V}) & \text{iff} & undefined(f) \vee undefined(\text{fcs}, \mathcal{V}) \\
undefined(f, \mathcal{V}) & \text{iff} & undefined(f) \\
undefined(exp.f, \mathcal{V}) & \text{iff} & undefined(f) \vee undefined(exp, \mathcal{V})
\end{array}
$$

5.8 Verifying Correctness of the Extraction

5.8.1 Proof Obligations

In order to prove consistency, it is sufficient to show that only dead ASM rules are removed. This property does not follow automatically from the formally defined operations for removal, since they rely on the invariants of the reduction profile. However, based on these operations, it is possible to derive proof obligations that have to be verified in order to prove consistency.

For example, during removal, an **if**-rule can be replaced by the subrule in the **then**-block of the rule, if the predicate *true* holds for the guard. To prove consistency, it is sufficient to prove that for all specifications of the SDL profile, the guard evaluates to `true` in all reachable states[2]. Likewise, if the predicate *false* holds for the guard, we have to prove that for all specifications of the SDL profile, the guard evaluates to `false` in all reachable states. In case *undefined* holds for the guard, we have to prove that the **if**-statement can not be reached at all.

Figure 5.6 shows a part of the formal language definition that was removed as part of the save feature of SDL, which is used to defer the consumption of input signals. For SDL profiles that do not contain the save feature, no grammatical elements of *Save-signalset* exist. Therefore, selecting the *Save-signalset* for any state yields `undefined`, and selecting *Signal-identifier**-set*** for the element `undefined` yields the empty set. Since *Save-signalset* is not modified in the formal language definition, this holds for

[2] This condition is stronger than necessary. It would suffice to show that the guard is always `true` for all reachable states that lead to the firing of the **if**-rule.

```
1 if Self.signalChecked.signalType ∈
2       sn.stateAS1.s−Save−signalset.s−Signal−identifier−set then
3    Self.SignalSaved := True
4 endif
```

Figure 5.6: Removed part of semantics definition

any reachable state of the ASM. An element can not be contained in an empty set, therefore the guard is always false, and omitting the **if**-statement leads to a consistent definition for specifications without save.

Choose. Choose nondeterministically selects an element that satisfies the constraint given by expression *exp*. If a **choose**-rule is removed, we have to prove consistency by proving the expression *exp* to be `false` in any reachable state, and therefore - according to the semantics of ASMs - the **choose**-rule equates to an empty update set. Alternatively, we can prove that the **choose**-rule can not be reached.

Extend. **extend**-rules are removed if they extend a domain that has been removed from the ASM signature. For a domain that is associated with a language feature, an extension of the domain must not be reached if that feature is removed. In order to prove consistency, we therefore have to prove that such a rule can not be reached.

Rule Macro Definition. A rule macro can be removed without affecting consistency if no corresponding rule call exists in a reachable part of the ASM, or if the body of the rule macro can be reduced to skip while maintaining consistency.

Boolean Expressions Parts of boolean expressions are removed if they have no influence on the final result, for example if *true* holds for a subexpression of a conjunction. In this case, the proof obligation is to show that the subexpression is always true for specifications of the SDL profile.

Proof obligations on boolean expressions can be split into proof obligations on subexpressions, as shown for $\wedge$ and $\vee$ below. For example, in order to prove consistency for predicate *true* on $e_1 \wedge e_2$, we can prove consistency for predicate *true* on e_1 and e_2.

$$true(e_1 \wedge e_2) \quad \text{iff} \quad true(e_1) \textbf{ and } true(e_2) \tag{5.1}$$
$$false(e_1 \wedge e_2) \quad \text{iff} \quad false(e_2) \textbf{ or } false(e_2) \tag{5.2}$$
$$true(e_1 \vee e_2) \quad \text{iff} \quad true(e_1) \textbf{ or } true(e_2) \tag{5.3}$$
$$false(e_1 \vee e_2) \quad \text{iff} \quad false(e_1) \textbf{ and } false(e_2) \tag{5.4}$$

Proof obligations for ASM rules and expressions can be inserted into the reduced formal semantics definition by the SDL-profile tool described in the following chapter.

Relational Operators Relational Operators like $>, <, \geq$ and $\leq$ are undefined if one of their subexpressions is undefined. To prove consistency, we have to prove that the expression can not be reached.

In case of the element-of operator $e_1 \in e_2$, two heuristics were used. If *undefined* holds for the expression on the right hand side, the set is interpreted as empty and *true* holds for the expression. In this case, we have to prove that the right hand side always equates to the empty set. In case *undefined* holds for the expression on the left hand side, we have to prove that the element described by this expression is not contained in the set on the right hand side in any reachable state.

5.8.2 Proofs over Distributed Abstract State Machines

In order to prove the proof obligations that are created during the extraction, we need to reason over the structure of the dynamic semantics, and over the updates fired in certain states. To make these proofs more precise, we define first-order formulae that hold for an ASM. These formulae define what locations are updated depending on the guards of if-rules, and other ASM rules. Here, we only consider ASMs that produce consistent update sets, such as the dynamic semantics of SDL.

Function Update maps an ASM rule to a corresponding update formula that describes which locations are affected by updates depending on guards and constraints in the current state. Function NUpdate maps an ASM rule to a corresponding update formula that described which locations are *potentially* affected by updates. To describe updates of locations, we define the following predicates:

$\mathbf{upd}(f, x, t)$ is defined as "the content of $f(x)$ is changed in the next state to the current value of t"

$\mathbf{nupd}(f, x, t)$ is defined as "the content of $f(x)$ is potentially changed in the next state to the current value of t"

We now define Update and NUpdate for the different kinds of ASM rules.

Update. The basic update instruction $f(x) := t$ writes the value of term t into the location described by function f and parameter x. Inconsistent update sets excluded, the value of this location in the next state will be the value of t in the current state.

$$\text{Update}(f(x) := t) := \mathbf{upd}(f, x, t)$$

Skip. Skip does not produce any updates, and is defined in the update formula as true.

$$\text{Update}(\text{skip}) := \text{true}$$

If-rules. Generally, update instructions are guarded by several if-rules. That means that the update instruction is only fired if a guard g holds. This can be expressed in a formula using implication: g, a first-order formula, implies that the update formula described by Update(R) holds. If g doesn't hold, Update(R) can still be true, if the respective locations are updated in other parts of the ASM.

$$\text{Update}(\textbf{if } g \textbf{ then } R \textbf{ endif}) := g \rightarrow \text{Update}(R)$$

$$\text{Update}(\textbf{if } g \textbf{ then } R_1 \textbf{ else } R_2 \textbf{ endif}) := (g \rightarrow \text{Update}(R_1)) \wedge (\neg g \rightarrow \text{Update}(R_2))$$

Parallel. In a parallel rule block with consistent update sets, the resulting update set is the unification of the update sets of the subrules. The resulting update formula is the conjunction of the update formulae of the subrules.

$$\text{Update}(\textbf{do in-parallel } R_1\ R_2 \textbf{ enddo}) := \text{Update}(R_1) \wedge \text{Update}(R_2)$$

Do Forall. For all x for which the guard $g(x)$ holds, the update formula Update($R(x)$)) holds.

$$\text{Update}(\textbf{do forall } x : g(x)\ R(x) \textbf{ enddo}) := \forall x.(g(x) \rightarrow \text{Update}(R(x)))$$

Let. The let-rule defines variable x as a shortcut for a term t. The let-rule can be replaced by replacing every occurrence of x in the rule body with term t. In the update formula, every occurrence of x is substituted by term t, which is safe because t does not contain free variables.

$$\text{Update}(\textbf{let } x = t \textbf{ in } R(x) \textbf{ endlet}) := \text{Update}(R(x)))[x/t]$$

Extend. Extend imports a fresh element from the infinite reserve and includes it in domain D, by setting $D(x)$ to true.

$$\text{Update}(\textbf{extend } D \textbf{ with } x\ R(x) \textbf{ endextend}) :=$$
$$\exists x.(x \in \text{Reserve} \wedge \textbf{upd}(D, x, \textbf{true}) \wedge \text{Update}(R(x)))$$

Rule Macro Call. A rule macro call fires the body of the rule macro definition with the formal parameters replaced by name by the parameters passed during the rule macro call. For a rule macro definition Rule($a_1, \ldots, a_n$) $\equiv R$, the update formula of the call is defined as follows:

$$\text{Update}(\text{M}(x_1, \ldots, x_n)) := \text{Update}(R[a_1/x_1, \ldots a_n/x_n])$$

In order to prevent the update formula from growing too large, we can replace the rule macro call with a macro representing the update formula for the rule macro definition.

Choose. Choose fires the update of $R(x)$ for an x for which the guard $g(x)$ holds. If no such x exists, the choose rule has no effect.

$$\begin{aligned}&\text{Update}(\textbf{choose}\, x : g(x)\; R(x)\; \textbf{endchoose}) :=\\ &\qquad \exists x.(g(x) \rightarrow \text{Update}(R(x))) \vee \forall x.\neg g(x)\end{aligned}$$

For proofs about consistency, we are usually interested in what locations are potentially affected by updates. NUpdate produces a formula that includes all locations potentially affected by the choose-rule.

$$\begin{aligned}\text{NUpdate}(\textbf{choose}\, x : g(x)\; R(x)\; \textbf{endchoose}) &:= \forall x.(g(x) \rightarrow \text{NUpdate}(R(x)))\\ \text{NUpdate}(f(x) := t) &:= \textbf{nupd}(f, x, t)\end{aligned}$$

For all other rules, NUpdate is defined as Update.

Distributed Abstract State Machines. Update and NUpdate are applied to all programs defined for a DASM. The formula of a program defined by Update holds for an agent with this program if the agent performs a move in the current state. The formula of a program defined by NUpdate holds for an agent with this program if the agent *can* perform a move in the current state. Since the proofs don't rely on a fixed partially-ordered run, we assume that an agent can perform a move in any state, or produces a trivial update set.

Example. The following example shows the result of NUpdate for the rule macro SignalOutput from the dynamic semantics of SDL, generated by a tool which implements the definitions described above.

```
1  SignalOutput(s: Signal,vSeq: Value*,toArg: [ ToArg ],viaArg: ViaArg) ≡
2    let invReference=(if (toArg ∈ PId) then (idToNodeAS1(s) ∉ toArg.s-Interface-definition.s-
         Signal-definition-set) else False endif) in
3      if invReference then
4        RaiseException(InvalidReference, empty)
5      else
6        choose g: g ∈ (Self.outgates ∪ Self.ingates) ∧ Applicable(s,toArg,viaArg,g,undefined)
7          extend PlainSignalInst with si
8            si.plainSignalType := s
9            si.plainSignalValues := vSeq
10           si.toArg := toArg
11           si.viaArg := viaArg
12           si.plainSignalSender := Self.self
13           Insert(si, now, g)
14         endextend
15       endchoose
16     endif
17   endlet
```

```
1  SignalOutput(s,vSeq,toArg,viaArg) ≡
2    ((invReference → RaiseException[eid/InvalidReference, vSeq/empty])
3    ∧ (¬ invReference →
4      ∀ g.(g ∈ (Self.outgates ∪ Self.ingates) ∧ Applicable(s, toArg, viaArg, g, undefined) →
5        ∃ si.( si ∈ Reserve ∧nupd(PlainSignalInst,si,true) ∧
6        nupd(plainSignalType, <si>, s) ∧
7        nupd(plainSignalValues, <si>, vSeq) ∧
8        nupd(toArg, <si>, toArg) ∧
9        nupd(viaArg, <si>, viaArg) ∧
10       nupd(plainSignalSender, <si>, Self.self) ∧
11       Insert[si/si, t/now, g/g])
12     ))
13   )[invReference/(if (toArg ∈ PId) then (idToNodeAS1(s) ∉ toArg.s-Interface−definition.s-
         Signal−definition−set) else False endif)]
```

5.8.3 Case Study: Proving Correctness of Extraction

In this section, we prove the correctness of the extraction for three SDL profiles, a profile without the *save* feature, a profile without the *timer* feature, and a profile without the *exception* feature. The proof obligations are generated by the SDL-profile tool, described in Chapter 6, and inserted into the extracted semantics definition as comments (see Section 6.3.6). A proof obligation is generated for every statement that is removed based on heuristics (see Section 5.8.1).

Profile Without Save

Save-signalset = ∅
SignalSaved = false

Figure 5.7: Reduction Profile for save Feature

Saving is activated for a signal in a state by including it in the < save part > of a state. The < save part > is mapped by the static semantics to the *Signal-identifier*-**set** of the *Save-signalset* of the corresponding *State-node*. Since no other element is mapped to this set, for specifications without save, the following property holds in the initial state:

$$\forall s \in \textit{State-node}.(s.\mathbf{s}\text{–}\textit{Save-signalset}.\mathbf{s}\text{–}\textit{Signal-identifier}\textbf{-set} = \emptyset) \qquad \text{(PS1)}$$

Since sets of *Signal-identifier*s are not modified in the dynamic semantics, this property holds in every state of the SVM.

```
1  ...
2  elseif (stateNodeKind(sn) = stateNode) then
3    let curSigId: Identifier = Self.signalChecked.signalType in
4      Self.stateNodeChecked := sn
5      Self.stateNodesToBeChecked := (Self.stateNodesToBeChecked \ { sn })
6      Self.transitionsToBeChecked := { t ∈ stateTransitions(sn).inputTransitions: (t.s-Signal =
           curSigId) }
```

```
7     /* if: prove expression false : (Self.signalChecked.signalType ∈ stateAS1(sn).s-
           Save-signalset.s-Signal-identifier-set) */
8   endlet
9 endif
```

The first proof obligation comes from removing an **if**-rule with a guard that was determined as false in the extraction, given the reduction profile. The proof follows directly from (PS1), from which we can derive that the set on the right hand side of the element-of operator is always empty.

Self.signalChecked.signalType ∈ *stateAS1*(*sn*).**s-***Save-signalset*.**s-***Signal-identifier*-**set**
implies$_{\text{PS1}}$ *Self.signalChecked.signalType* ∈ ∅
implies false

```
1 if (Self.stateNodesToBeChecked ≠ ∅) then
2   SelectNextStateNode
3 else
4   /* if: prove expression true : ¬(Self.SignalSaved) */
5   ...
6 endif
```

The second proof obligation comes from removing the **else**-part of an **if**-expression, since ¬(*Self.SignalSaved*) was determined as true. We show that *Self.SignalSaved* is always false in states leading to the execution of the **if**-rule. First, we find all potential updates of *SignalSaved* to true, for an arbitrary agent. NUpdate contains two potential updates of *SignalSaved*, occurring in subformulas of the form $g \rightarrow \mathbf{nupd}(\mathit{SignalSaved}, < \mathit{Self} >, \text{true})$, g being the expression proved false in the first proof obligation. Since a controlled location that is not updated keeps its value in the next state, *SignalSaved* is only updated to true if g holds. Therefore, an update of *SignalSaved* to true implies that g holds: $\mathbf{nupd}(\mathit{SignalSaved}, < \mathit{Self} >, \text{true}) \rightarrow g$. From this follows (with $g \equiv$ false as shown above):

$g \equiv \mathbf{nupd}(\mathit{SignalSaved}, < \mathit{Self} >, \text{true})$ **implies**
$\text{false} \equiv \mathbf{nupd}(\mathit{SignalSaved}, < \mathit{Self} >, \text{true})$ **implies**
$\neg\mathbf{nupd}(\mathit{SignalSaved}, < \mathit{Self} >, \text{true})$

For a specification without save, *SignalSaved* is therefore never set to true. Since *SignalSaved* can have any value in the initial state, we must also show that the function is set to false for an agent before that agent evaluates the reduced rule. This can be shown over the sequence of modes the agent goes through. Rule macro SelInputStartPhase, which sets *SignalSaved* to false, is always executed before SelInputSelectionPhase, which contains the proof obligation.

Profile Without Timer

Timers in SDL can be set, reset, and checked if they are active. In the abstract grammar, this corresponds to the abstract syntax elements *Set-node*, *Reset-node* and

TIMER = ∅, TIMERINST = ∅, TIMELABEL = ∅, SET = ∅, RESET = ∅, TIMERACTIVE = ∅, *Set-node* = ∅, *Reset-node* = ∅, *Timer-active-expression* = ∅

Figure 5.8: Reduction profile for 'Timer' feature

Timer-active-expression, which do not occur for profiles without timers. The *compilation function* maps these elements to the SVM domains SET, RESET and TIMERACTIVE, respectively. Since these domains are not modified elsewhere, the following property holds in all states of the SVM:

$$\text{SET} = \text{RESET} = \text{TIMERACTIVE} = \emptyset \qquad \text{(PT1)}$$

This property directly satisfies the proof obligations generated during the reduction:

```
1 EVAL(a: ACTION) ≡
2   ...
3   else
4     /* if: prove expression false : (a ∈ Set) */
5     /* if: prove expression false : (a ∈ Reset) */
6     /* if: prove expression false : (a ∈ TimerActive) */
7      ...
8   endif
```

The domain TIMERINST holds all active instances of timers and is initially empty. We show that it is empty in every state by collecting all potential updates of domain TIMERINST, and by evaluating the conditions that lead to the update. From this follows, that a potential update of TIMERINST only occurs if and only if an action of kind set, reset or timer-active exists.

$$\exists a.(a \in \text{SET} \vee a \in \text{RESET} \vee a \in \text{TIMERACTIVE}) \equiv \exists x.(\textbf{nupd}(\text{TIMERINST},\text{x},\text{true}))$$
$$\textbf{implies}_{\text{PT1}} \quad \nexists x.\textbf{nupd}(\text{TIMERINST},\text{x},\text{true})$$

```
1 signalType(si: SIGNALINST):SIGNAL =def
2   if (si ∈ PlainSignalInst) then plainSignalType(si)
3   elseif /* if: prove expression false: si ∈ TimerInst */
4     (si ∈ ExceptionInst) then si.s-EXCEPTION
5   endif
```

Profile Without Exceptions

Exceptions are an extension to signals introduced in SDL-2000. A transition that terminates with a raise node generates an exception instance that is consumed in a handle node (a special input node) in the active exception handler node (a special state node).

Several elements of the abstract syntax correspond to the exception feature. Most notably *Raise-node*, which terminates a *Transition* and generates an exception instance, *On-exception*, which sets the active exception handler, and *Exception-handler-node* and *Handle-node*, which define the exception handler.

EXCEPTION $= \emptyset$, EXCEPTIONINST $= \emptyset$,
EXCEPTIONHANDLERNAME $= \emptyset$,
EXCEPTIONHANDLERNODE $= \emptyset$,
SETHANDLER $= \emptyset$, RAISE $= \emptyset$
Exception-handler-name $= \emptyset$,
Exception-definition $= \emptyset$,
Exception-handler-node $= \emptyset$, *On-exception* $= \emptyset$,
Handle-node $= \emptyset$, *Raise-node* $= \emptyset$
selectException $=$ undefined

Figure 5.9: Reduction profile for 'Exception' feature

The compilation function maps *Raise-node*s and *On-exception*s to domains RAISE and SETHANDLER, respectively. These domains are not modified elsewhere in the SVM, therefore the following property holds in all states.

$$\text{RAISE} = \text{SETHANDLER} = \emptyset \tag{PE1}$$

As with the removal of timers, the proof obligations inserted into the EVAL rule macro are directly satisfied by this property.

```
1 EVAL(a: ACTION) ≡
2   ...
3   else
4     /* if: prove expression false : (a ∈ SetHandler) */
5     /* if: prove expression false : (a ∈ Raise) */
6     ...
7   endif
```

For profiles without exceptions, no exception handler nodes are included in the abstract syntax, therefore property PE2 holds.

$$\forall s \in \textit{State-transition-graph}.(s.\textbf{s-}\textit{Exception-handler-node} = \emptyset) \tag{PE2}$$

```
1 INITSTATEMACHINE ≡
2   if ( Self.stateNodesToBeCreated ≠ ∅) then ...
3   elseif ( Self.statePartitionsToBeCreated ≠ ∅) then ...
4   else
5     /* if: prove expression false : (Self.ehNodesToBeCreated ≠ ∅) */
6   endif
```

To prove *ehNodesToBeCreated* equates to the empty set for any agent, we check the possible updates of the location. The SVM specifies one potential update of *ehNodesToBeCreated* to a value unequal the empty set:

nupd(*ehNodesToBeCreated*, < *Self* >, *csdg*.**s**−*State-transition-graph*.**s**−*Exception-handler-node*−**set**) **implies**$_{\text{PE2}}$
nupd(*ehNodesToBeCreated*, < *Self* >, ∅)

Therefore, all potential updates of *ehNodesToBeCreated* write the empty set to this location. To prove that *ehNodesToBeCreated* is always initialised to the empty set for an agent, we can show that the creation of an agent is always followed by the execution of the rule macro InitStateMachine for the agent, which sets the location to the empty set. We omit the details of this proof here.

```
1 SelectTransition ≡
2   if ( Self.agentMode3 = startSelection) then ...
3   else
4     /* if: prove expression false : (Self.agentMode3 = selectException) */
5     ...
6   endif
```

```
1 SelectTransitionStartPhase ≡
2   /* if: prove expression false : (Self.currentExceptionInst ≠ undefined) */
3   ...
```

In the reduction profile for exceptions (see Figure 5.9), we have specified that agent mode *selectException* can not be reached, which leads to the omission of transition selection for exceptions. From the update formula follows, that setting the agent mode to *selectException* is guarded by a boolean term stating that the current exception instance is not undefined (see above). Furthermore, for the current exception instance not to be undefined, domain ExceptionInst must not be empty.

However, this assumption does not hold. Rule macro RaiseException creates an exception instance, and is called within the SVM for example when a division by zero occurs, or when a value in a decision is out of range. In these cases, selection of an exception handler is triggered, which results in the agent program being set to undefined behaviour, since no exception handlers exist in a profile without exceptions. The resulting profile is only consistent for specifications that don't lead to exceptions at runtime.

Therefore, agent mode *selectException* must not be removed in a profile without exceptions, or rule macro RaiseException must be modified to set the program to the undefined behaviour program directly. Due to the complexity of the transition selection, this has to be done manually.

5.9 Related Work.

ConTraST [65, 18] is an SDL to C++ transpiler that generates a readable C++ representation of an SDL specification by preserving as much of the original structure as

possible. The generated C++ code is compiled together with a runtime environment that is a C++ implementation of the formal semantics defined in Z100.F3. ConTraST is based on the textual syntax of SDL-96, and supports SDL profiles syntactically through deactivation of language features, and semantically by suppressing unreachable parts of the runtime environment for a given profile, as identified by the formal extraction.

A logic for ASMs is introduced in [64], similar to the logic introduced in Section 5.8.2. The logic takes consistency into account, but does not cover indeterminism or the import of elements from the reserve.

In [47], the concept of program slicing is extended to Abstract State Machines. For an expressive class of ASMs, an algorithm for the computation of a minimal slice of an ASM, given a slicing criterion, is presented. The slicing criterion is a list of locations of the ASM that are of interest with regard to the slice. The resulting ASM is correct with regard to the slicing criterion, and minimal for an expressive subclass of ASMs. Possible applications of ASM slicing are testing and error detection. While the complexity of the slicing algorithm is acceptable in the average case, the worst case complexity is exponential. In [48], a polynomial slicing algorithm is introduced that produces a possibly non-minimal slice for every ASM. However, indeterminism and dynamic memory allocation are not covered. Furthermore, for our application, the number of locations of interest - locations of the state relevant for the dynamic semantics of a language profile - is much larger than the number of irrelevant locations.

In [6], a modular definition of C♯ using ASMs is presented. In the paper, C♯ is divided into layers of orthogonal language features, starting from an imperative core, and subsequently adding classes, object-orientation, exception handling, delegates and events, concurrency and pointer arithmetic. The semantics of C♯ is defined by specifying an interpreter for an annotated abstract syntax tree representing a C♯ program. For each layer, the abstract syntax is extended, and two new rule macros are specified defining the semantics for new statements and expressions of the layer. The rule macros define the effect of computing the program construct associated with an abstract syntax node. The close integration of syntax elements and their semantics achieved in this fashion leads to pure incremental extensions that can be easily modularised by executing the rule macros for different layers in parallel. Similar approaches can be found for the formal semantics of the Java Virtual Machine [7] and C [29], both defined using ASMs. While this approach is suitable for imperative languages like C and Java, a complex specification language with a high degree of parallelism and asynchronous communication like SDL is better defined using a virtual machine approach, for which an extraction approach - as defined in Section 5.7 - is more suitable.

5.10 Summary and Conclusions

In this chapter, we have introduced an approach to extract the formal semantics definition for a language profile, given a dynamic semantics defined using ASMs. The extraction is based on recognising and removing dead ASM rules from the formal semantics definition, starting from a reduced ASM signature. The reduction of the

ASM signature is derived from the abstract syntax of removed language modules. To achieve deterministic results, we have formalised the approach using functions defined recursively over the concrete syntax of ASMs.

The concept behind the extraction defined in this chapter is to identify parts of the ASM signature that are not modified in a run of the ASM for specifications in the given language profile. For these parts of the signature, we specify default values, derived from the static syntax of the language and the compilation function of the dynamic semantics. These default values can be used in the reduction of the formal semantics. We found the following set of default values to be sufficient for the dynamic semantics of SDL: the special ASM element `undefined`, the empty set, the empty sequence, and the boolean values `true` and `false`. Further default values can be added as needed, for example for locations holding natural or real numbers.

The advantage of this approach is that the extraction can be performed efficiently, requiring only a single traversal of the formal semantics definition. The approach is sufficiently powerful, leading to significant reduction of dead ASM rules for common language features. For SDL, the reduction achieved is less significant where transition selection is effected, due to the structure of the SDL formal semantics. For transition selection, only small reductions are possible, unless complete parts of transition selection defining certain transition types, such as priority input, are removed. Results of the application of the extracttion to the formal semantics of SDL, using the SDL-profile tool, are discussed in Chapter 6.

While the approach is efficient, specifying incorrect default values or including functions and domains that can be changed by a specification in the profile in the reduction profile can lead to the extraction of formal semantics definitions that are inconsistent with the complete formal semantics. Here, a subsequent verification step is needed to show the consistency of the extracted formal semantics with regard to specifications included in the corresponding profile.

5.11 Future Work

Language Modules and Composition Approach

An open question is the feasibility of defining a modular semantics by composition of language modules on top of a language core, if the virtual machine approach is used (as in case of SDL). It is important that language modules encapsulate all parts of the specification corresponding to a language feature. For the instruction set of the abstract machine, this can be achieved in a similar fashion as for interpreter semantics. In Listing 5.3, the domain ACTION is defined incrementally (line 2), with each module extending the set of actions. For each type of action defined by the module, a behaviour primitive is encapsulated in a rule macro. The intructions in the rule macro are fired when the current label of the executing agent points to a behaviour (line 5), and the action of the behaviour has the correct type (for example, SET, line 7). The rule macros for different action types are mutually exclusive, and can be composed using the parallel rule composition of ASMs.

```
1  module Timer requires Data
2    Action' =def Action ∪ Set ∪ Reset ∪ TimerActive
3
4    EvalSet ≡
5      choose b: b ∈ behaviour ∧ b.s-Label = Self.currentLabel
6        let a = b.s-Action in
7          if a ∈ Set then
8            SetTimer(a.s-Timer, values(a.s-ValueLabel-seq, Self), semvalueReal(value(a.s-
                 TimeLabel, Self)))
9            Self.currentLabel := a.s-ContinueLabel
10         endif
11       endlet
12     endchoose
13
14   ...
```

Listing 5.3: Modular definition of timer instruction set

Language features may affect various parts of the SVM (for example, routing or transition selection), where modularisation and composition is hard to achieve. For example, when creating an agent instance, functions for procedures and exceptions are initialised. A possible solution is to add insertion points to the ASM, that can be referred to from a module definition. ASM rule fragments tied to an insertion point in a module are executed in parallel with ASM rules at the insertion point. Possible insertion points are the agent modes of the SVM, described in Section 2.4. Open questions are the handling of local variables, sequential rule fragments, and feature interaction. Drawbacks of this approach are limited extensibility - depending on the number of insertion points - and possibly a negative effect on readability.

```
1  EnterStateNodesEnterPhase ≡
2    if Self.stateNodesToBeEntered ≠ ∅ then
3      choose snwen: snwen ∈ Self.stateNodesToBeEntered
4        ...
5        if snwen.s-StateNode.stateNodeRefinement = undefined then
6          enterstate: RefinementUndef(snwen)
7        elseif snwen.s-StateNode.stateNodeRefinement = stateAggregationNode then
8          enterstateaggr: RefinementStateAggrNode(snwen)
9        elseif snwen.s-StateNode.stateNodeRefinement = compositeStateGraph then
10         entercompstate: RefinementCompStateNode(snwen)
11       endif
12     ...
13
14 module EntryProcedure requires Procedure
15   entercompstate:
16     let cstd : Composite−state−type−definition = snwen.s-StateNode.stateDefinitionAS1 in
17       let comp : Composite−state−graph = cstd.s-implicit in
18         if comp.s-Entry−procedure−definition ≠ undefined then
19           CreateProcedure(comp.s-Entry−procedure−definition, undefined, undefined)
20         endif
21       endlet
22     endlet
23
24   enterstateaggr:
25     let cstd : Composite−state−type−definition = snwen.s-StateNode. stateDefinitionAS1 in
26       let aggr : State−aggregation−node = cstd.s-implicit in
```

```
27          if aggr.s-Entry−procedure−definition ≠ undefined then
28            CreateProcedure(aggr.s-Entry−procedure−definition, undefined, undefined)
29          endif
30        endlet
31      endlet
```

Listing 5.4: Modular definition of entry procedures

Listing 5.4 shows the SVM rule macro EnterStateNodesEnterPhase, defining entry for plain states, state aggregations and composite states, with corresponding insertion points enterstate (line 6), enterstateaggr (line 8) and entercompstate (line 10). Module EntryProcedure (line 14) - containing rule parts of Entry-/Exit-procedure from Table 6.4 - defines rule fragments for the execution of entry procedures, that are tied to insertion points entercompstate (line 15) and enterstateaggr (line 24). The rule fragments use local variables defined in the scope enclosing the insertion points. Module EntryProcedure requires procedures because the rule macro CreateProcedure is called (lines 19, 28).

6 The SDL-Profile Tool

Based on the formalisation provided in Chapter 5, we have implemented a tool called SDL-profile tool, in order to automate the extraction process, providing visible results. The tool reads the formal semantics definition[1], performs the *remove* operation based on a *reduction profile*, and outputs a reduced version of the formal semantics. The reduction profile is a list of domain names, function names, and macro names that are removed from the ASM signature (or from the set of rules, in the case of macro names). For predicates, a default value (`true` or `false`) is defined explicitly.

In this chapter, we give an overview over the sequence of steps of the SDL-profile tool, describe the term processor kimwitu, which we use for defining and modifying abstract syntax trees, and describe the implementation of the tool in detail. Finally, we show the results of applying the formal extraction, using the SDL-profile tool, for several SDL profiles.

6.1 Sequence of Steps of the SDL-profile Tool

This section gives a short overview of the steps taken by the SDL-profile tool. Figure 6.1 shows the sequence of steps performed during the extraction, and the tools used for each step.

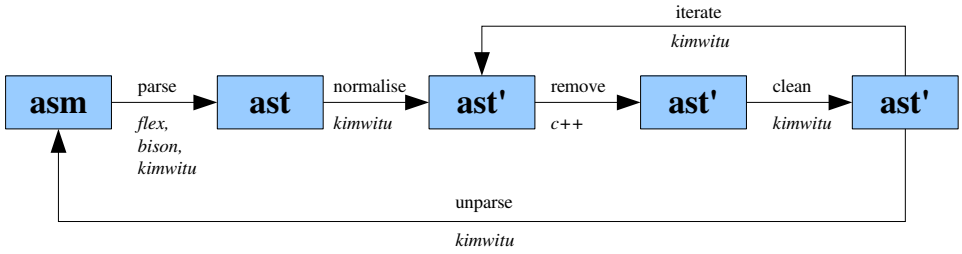


Figure 6.1: Sequence of steps of the SDL-profile tool

6.1.1 Parse

The *parse* step takes a semantics definition using ASMs as input and creates an abstract syntax tree representation of the ASM as output. It is generated out of specifications of the ASM syntax (lexis, concrete and abstract syntax), as defined for the

[1] The SDL-profile tool can be applied to every formal semantics definition specified using ASMs. However, the concrete syntax of ASMs is given by the dynamic semantics of SDL.

formal semantics of SDL-2000 [20]. The specification of the abstract syntax is translated by kimwitu++ (see Section 6.2) to a data structure for the abstract syntax tree, where each node type corresponds to a C++ class. Scanner and parser are generated by flex and bison, respectively. The parser is a modified version of the parser defined in [58], and accepts the same language.

6.1.2 Normalise

The *normalise* step transforms the abstract syntax tree to a pre-removal normal form. The transformation is specified by rewrite rules on the abstract syntax tree. The rewrite rules are translated to C++ functions by the kimwitu tool. The main function of the normalisation step is to split up complicated abstract syntax rules, in order to simplify the definition of the remove function.

6.1.3 Remove

The *remove* step is the implementation of the remove function formalised in Section 5.7. For each type of node (called *phylum*) in the ASM abstract syntax, a C-function called 'remove' is introduced. The remove function performs removal for each term of the respective node type, for example the terms `IfThenElse`, `Choose`, and `Extend` for the *rule* node type. It returns a term of the respective node type as result - for example, the remove function for rules always returns a term of type rule.

6.1.4 Clean

The *clean* step transforms superfluous rules resulting from the remove step to a post-removal normal form. The normal form is achieved by defining term rewrite rules in kimwitu. Unlike removal, the rewrite rules apply anywhere where their left hand side matches, and are applied as long as a match is found. The clean step only removes trivial parts of the formal semantics definition. The resulting specification is semantically equivalent to the specification before the clean step.

6.1.5 Iterate

Given a complete reduction profile, only one run of the SDL-profile tool is needed to generate the reduced formal semantics definition. In case the reduction profile is incomplete, the profile tool can identify further names in the ASM signature that can be removed, and iterate the remove and clean steps[2]. For example, a function in the ASM signature with a target domain that has been removed during the previous remove step, is included in the reduction profile of a subsequent iteration. Primarily, definitions (for example, rule macros) that become unreferenced in the extracted formal semantics are removed in subsequent iterations.

[2] The intent is to keep the reduction profile as small as possible, including only parts of the signature directly related to features that should be removed, and let the tool identify the remaining parts in iteration steps.

6.1.6 Unparse

The *unparse* step traverses the abstract syntax tree and outputs a string representation of every node. The result is a textual representation of the abstract syntax tree in the original input format. Therefore, the output of the profile tool can be used as the input for a subsequent run of the profile tool. We also provide different output formats, for example a Latex document of the formal semantics, and a translation to C++.

6.2 The Term Processor Kimwitu

Kimwitu++[3] [45, 46] is a tool for the definition and handling of abstract syntax trees, an important part of compiler generation. It supports the definition of an abstract syntax in a BNF-style notation, additional control structures, and rewrite and printing ("unparse") rules. These definitions and rules are translated by kimwitu into C++. Kimwitu is used extensively for the generation of the SDLC compiler [58].

6.2.1 Defining the Abstract Syntax.

In kimwitu, an abstract syntax is defined by a set of node types, called *phyla*. Each node type has a name (for example, 'rule') on the left hand side of a colon, and a set of alternative operators - separated by '|' - on the right hand side, the nodes of the abstract syntax tree. Each operator has a possibly empty list of node types as argument, representing the subnodes of this node. Node types correspond to non-terminals in the BNF, and operators to terminals.

```
rule:   ASSIGN(casestring argumentList expr)
      | IFTHENELSE(expr rule rule)
      | ...

expr:   VARIABLE(casestring)
      | ...
```

Listing 6.1: Excerpt from the abstract syntax definition of ASMs

From the abstract syntax definition, *terms* representing abstact syntax trees can be constructed. For example, given the abstract syntax of ASMs in Listing 6.1, the term `Assign("f", NilargumentList(), Variable("y"))`, corresponding to the concrete syntax "f() := y", can be constructed. These terms can be constructed manually, or as the output of a parser generator like bison.

For a node type, a corresponding list type can be defined. Two operators are implicitly introduced for each list type: an operator representing the empty list, and an operator concatenating an element of the node type with a list of elements. Below is the definition of a list type, and the equivalent right-recursive definition in kimwitu, without using the list type. Both definitions produce the same abstract syntax, but kimwitu generates special list handling functions for the list type.

[3] Kimwitu++ is a C++ based derivative of kimwitu [12]. In this document, we abbreviate kimwitu++ to kimwitu, referring to the C++ based tool.

```
exprList: list expr /* list type */

exprList: NilexprList() /* equivalent definition */
        | ConsexprList(expr exprList)
```

Listing 6.2: Definition of list types

Finally, attributes can be added to node types of the abstract syntax definition. In the abstract syntax tree, information can be stored in these attributes for each node of the respective node type. For example, an attribute can be added to the node type 'expr', in which the result of the evaluation of the expression can be stored.

```
expr:   Plus(expr expr)
      | Minus(expr expr)
      | Value(integer)
      { int value = 0; }
```

Listing 6.3: Defining attributes for node types

6.2.2 Rewriting Abstract Syntax Trees.

Rewrite rules specify *transformations* on the abstract syntax. A rewrite rule consists of a pattern on the left hand side, and the resulting term on the right hand side. Rewrite rules are applied as long as terms matching patterns are found in the abstract syntax tree, replacing the matches with the corresponding right hand side of the rewrite rule. To ensure termination of the rewrite process, the rewrite rules must be confluent, resulting in a normal form of the abstract syntax tree.

```
Plus(a,Plus(b,c))
-> <trans: Plus(Plus(a,b),c) >;
```

Listing 6.4: Transforming expressions with rewrite rules

Listing 6.4 shows a rewrite rule that transforms the addition operator according to the law of commutativity. The resulting normal form is a left-associative term. Identifier 'trans' specifies a *rewrite view*, which is used to group rewrite rules into rewrite systems.

6.2.3 Unparsing Abstract Syntax Trees.

Unparsing traverses and outputs the abstract syntax tree in a textual representation. Listing 6.5 shows an unparse rule for the addition operator. Kimwitu prints the string tokens and subterms a, b and c in the order specified. The string tokens are sent directly to a printer function; to the subterms, matching unparse rules are applied.

```
Plus(Plus(a,b),c)
-> [print: "(" a "␣+␣" b ")␣+␣" c ];
```

Listing 6.5: Unparsing the addition operator

Kimwitu allows using arbitrary C++-code on the right hand side of the unparse rule, enclosed in braces, which makes unparsing a powerful operation. Listing 6.6 shows how unparse rules can be used for computation. First, subterms a and b are traversed, in that order, storing the result of the evaluation in the value field of these terms (not specified here). Then, the C++-code is evaluated, adding the values of the subterms and storing the result in the value field of the expression exp.

```
exp=Plus(a,b)
-> [compute: a b { exp->value = a->value + b->value } ];
```

Listing 6.6: Computation using unparse rules

6.2.4 Kimwitu Control Structures.

Kimwitu offers additional control structures for handling abstract syntax trees, which can be used in combination with C++-code. The **with**-statement is the equivalent to a switch-statement for abstract syntax tree terms. The cases are a list of term patterns, with associated code enclosed in braces. In the example below, the **with**-statement is used to switch over expression terms, computing a value of the expression.

```
int compute(expr ex) {
  with (ex) {
    Plus(a,b): { return compute(a) + compute(b); }
    Minus(a,b): { return compute(a) - compute(b); }
    Value(a): { return a; }
    default: { return 0; }
  }
}
```

The **foreach**-statement implements iteration over list terms. The first argument of **foreach** is a variable that represents the current list element, the second argument holds the complete list. In the example below, a list of expressions (for example, the arguments of a function call) is evaluated, computing and storing the value of each expression.

```
int computelist(exprList ex) {
  foreach (e; exprList ex) {
    e->value = compute(e);
  }
}
```

6.3 Implementation Decisions

6.3.1 ASM Syntax Definition

For the syntax definition of ASMs, as used in the dynamic semantics of SDL, modified versions of the concrete and abstract syntax definitions of SDLC ([58]) are used. The abstract syntax is defined using kimwitu, introducing node types for each entity kind of an ASM specification: definitions, domains, rules, expressions, as well as patterns

and parameter/expression lists. For each entity of an entity kind, an operator is introduced in the respective node type. For example, the rule type (see Listing 6.7) has an operator for each kind of rule defined for ASMs.

```
rule:     Assign(casestring argumentList expr)
    | IfThenElse(expr rule rule)
    | Empty()
    | Skip()
    | Parallel(ruleList)
    | ForAll(casestring expr rule)
    | Choose(casestring expr rule)
    | Extend(domain nameList rule)
    | Let(letStatements rule)
    | RuleCall(casestring argumentList)
    ;
```

Listing 6.7: Abstract syntax for ASM rules

The concrete syntax of ASMs is defined using flex and bison. The generated parser accepts the same set of ASM specifications as the ASM parser used for the generation of the runtime environment for SDLC. Therefore, we can use the textual representation of the dynamic semantics, extracted from Z100.F3 [42] for the SDLC tool.

6.3.2 Implementing the Predicates

For the predicates *true*, *false* and *undefined* (see Chapter 5), three boolean-valued functions *eval_true*, *eval_false* and *eval_undef* are introduced. These functions are overloaded for every node type the respective predicate applies to (see Listing 6.8): expressions for *eval_true* and *eval_false*; expressions, domains, definitions, patterns and argument lists for *eval_undef*. Each predicate has a set of undefined variable names V as second parameter, as in the formal definition of the predicates.

```
1 bool eval_true(expr ex, SET V);
2 bool eval_false(expr ex, SET V);
3 bool eval_undef(expr e, SET V);
4 bool eval_undef(domain dom, SET V);
5 bool eval_undef(definition def, SET V);
6 bool eval_undef(pattern p, SET V);
```

Listing 6.8: Interface of the predicates

These predicates are implemented with a case switch over all applicable operators of the respective node type, using the kimwitu control structure **with**. For *eval_true* and *eval_false*, all operators describing first-order expressions are relevant. For each operator, a boolean value is returned as defined in the formal definition of the predicates. For example, for the conjunction operator (&) *eval_true* (*eval_false*) returns `true` for all cases which yield T (F) in the corresponding truth table (see Table 5.3).

```
eval_true(expr ex, SET V) {
  with (ex) {
    BinOp("&", e1, e2): { return (eval_true(e1,V) && eval_true(e2,V)) ||
                                 (eval_true(e1,V) && eval_undef(e2,V)) ||
                                 (eval_undef(e1,V) && eval_true(e2,V)); }
```

```
    ...
    default: { return false; }
  }
}
```

Listing 6.9: Implementation of predicate *true* for conjunction

6.3.3 Implementing the Remove Function

Remove traverses the abstract syntax tree, starting from the topmost node. The rewrite operation of kimwitu is not well suited for this approach, since rewrites are performed anywhere in the abstact syntax tree where a rewrite pattern matches. The remove function is therefore implemented using C++-functions and kimwitu control structures.

For each node type, a *remove* function is introduced that returns a possibly reduced term of the same node type. For each operator of the node type, the returned term is determined by evaluating the predicates *eval_true*, *eval_false* and *eval_undef*, as defined in the formal definition of the extraction. If none of these predicates holds, a copy of the original term - possibly with reduced subterms - is returned.

The *remove* function always returns a copies of a term, leaving the original abstract syntax tree intact. This means that attributes calculated for the original abstract syntax tree (for example, references) have to be recalculated for the new abstract syntax tree. Listing 6.10 shows the general structure of the remove functions for each node type (phylum).

```
phylum remove(phylum p, SET V) {
    with (p) {
       Pattern: { /* term handled explicitly */
            if (/*check predicates*/) return transformed copy
            ...
            apply removal to children, return copy
        }
        ...
        /* terms not handled explicitly */
        default: { return (phylum) p->copy(true); }
    }
}
```

Listing 6.10: General definition of remove functions

6.3.4 Referencing Definitions

Calling the unparse function with unparse view `reference` on the abstract syntax tree traverses the tree and sets a reference for each node that references a definition. These references are needed primarily for two purposes: detecting unreferenced definitions that can be removed, and determining the type of parameters. Table 6.1 lists all nodes with corresponding definitions.

In order to store the references in the abstract syntax tree, we introduce an attribute of type definition to the node types rule (ASSIGN, RULECALL) and expr (PROGRAM,

Node	Referenced Definition
Assign (update)	FunctionDecl (location)
RuleCall	RuleDef
Program	ProgramDef
FunCall	FunctionDecl or FunctionDef (derived function)
MKCall (tuple constructor)	DomainDef (derived domain)

Table 6.1: Abstract syntax nodes and corresponding definitions

FunCall, MKCall). Additionally, we introduce an attribute to node type definition, which is used to store the number of references that point to a definition.

```
rule: ...                    expr: ...                    definition : ...
  { definition def; }          { definition def; }          { int refcounter=0; }
```

The unparse function for unparse view `reference` traverses the abstract syntax tree, and sets references for applicable nodes. Below is the unparse rule that sets the reference for the assign rule, which sets the location described by name and expression list args to the value of expression e. First, args and e are traversed, setting references for contained function calls and tuple constructors. Second, getFunDecl looks up the function declaration with the corresponding name and number of arguments. TheSpec refers to the root of the abstract syntax tree, which contains the list of global definitions as subterms; rul refers to the assign rule and is used to find local definitions. The definition returned is stored in the assign node.

```
rul=Assign(name,args,e)
-> [reference: args e { rul->def = getFunDecl(name,args,TheSpec,rul); } ];
```

The function *getFunDecl* returns a definition that is a function declaration (location) with corresponding name and number of arguments. First, the list of local definitions is searched, then the list of global definitions. Local definitions are definitions defined in the context of the rule macro in which rule rul occurs. Since kimwitu can not return the parent node for a given node of the abstract syntax tree, the list of local definitions is looked up starting from the topmost node of the abstract syntax tree.

```
definition getFunDecl(casestring name, argumentList args, asmSpec as, rule rul) {
    defList dl = (defList) as->subphylum(0); /* global definitions */
    defList dl2 = getLocalDefinitions(dl, rul); /* local definitions */
    foreach ($dcl; defList dl2) { /* check local definitions first */
        d=FunctionDecl(*, fname, dom, *): {
            if (fname->eq(name) && (args->length() == length(dom))) return d;
        }
        default: { /* do nothing */ }
    }
    foreach ($dcl; defList dl) { /* check global definitions next */
        d=FunctionDecl(*, fname, dom, *): {
            if (fname->eq(name) && (args->length() == length(dom))) return d;
        }
```

```
        default: { /* do nothing */ }
    }
    /* no definition found */
    return NULL;
}
```

Corresponding functions *getRuleDef*, *getFunDef*, *getDomainDef* and *getProgDef* are introduced for the definitions listed in Table 6.1.

Given an abstract syntax tree in which all references are set, the reference counter can be computed for each definition. This is done by traversing the tree, and incrementing n->def->refcounter for applicable nodes n where def is unequal to NULL. Definitions with refcounter = 0 can be removed with no effect on the semantics of the ASM.

6.3.5 Transforming Trivial Rules and Expressions

In the formal semantics definition extracted by *remove*, usually several trivial rules and expressions remain. Calling the rewrite function with rewrite view `clean` removes these parts of the semantics definition. Table 6.2 shows several trivial rules and expressions that are covered by rewrite rules, and the result of the transformation of these rules.

Pattern	**Result**
if e **then** r **else** r **endif**	r
let $x = t$ **in** `skip` **endlet**	`skip`
choose $x : g$ **in** `skip` **endchoose**	`skip`
do forall $x : g$ `skip` **enddo**	`skip`
if $x = a$ **then** a **else** x **endif**	x
let $x = t$ **in** x **endlet**	t
let $x = t$ **in** **if** $x = t$ **then** x **else** y **endif** **endlet**	x
$take(\emptyset)$	`undefined`

Table 6.2: Semantically equivalent transformations

Note that an **extend**-rule with rule body `skip` can not be transformed to `skip`, since **extend** modifies a domain of the ASM.

6.3.6 Generating Proof Obligations

Proof obligations can be generated by the SDL-profile tool in places where parts of the formal semantics definition were removed based on heuristics, as described in Section 5.8.1. A proof obligation is inserted at the place where the corresponding reduction took place, since it only applies if this part is executed. To insert proof obligations into rules, the rule node type of the abstract syntax is extended by a node ProofObligation (see Listing 6.11). ProofObligation has a casestring, a description of

the proof obligation, and an expression, the proof obligation, as subterms. Currently, the tool generates four kinds of proof obligations: the expression must be proven true, false, the place of the proof obligation must not be reached, and the value assigned to a location in the reduction profile equals the default value of that location.

```
rule:       ...
    | ProofObligation(casestring expr)
    ;
```

Listing 6.11: Abstract syntax for rules with proof obligation

Remove is modified to insert the proof obligation into the abstract syntax tree, as described in Section 5.8.1. In the example below, an **if**-rule is removed because expression e is evaluated as true. Expression e being true in all states that lead to the execution of the **if**-rule is a proof obligation that is inserted into the reduced abstract syntax by the remove function.

```
1 IfThenElse(e, r1, r2):  {
2   if (eval_true(e,V)) {
3     if (proofob) {
4       return Parallel(ConsruleList(remove(r1,V),ConsruleList(ProofObligation(mkcasestring("if:␣
              prove␣expression␣true"),e),NilruleList())));
5     } else { return remove(r1,V); }
6   }
7   ...
```

The implementation can be refined to generate more precise proof obligations. In the example above, if *eval_true* holds because e has the form $x \in D$, and D was evaluated as empty (*undefined*), $D \equiv \emptyset$ can be returned as proof obligation instead.

6.3.7 Output of the Reduced Abstract Syntax Tree

Creating a LaTeX-Document

The SDL-profile tool can generate output in latex format of the resulting ASM specification using the listings-package, which provides a latex environment for formatting specification and programming languages. The output format is identical to the textual representation of the ASM specification used as the input format, enclosed in the `lstlistings` environment, with the language set to `ASM`, and an appropriate LaTeX header and trailer. The main advantage of using textual output together with the listings package, as opposed to directly generating latex code, is that the listings class automatically handles line breaks, which would otherwise require substantial effort for long formulae.

The listings class needs a format definition for ASMs to be able to display them properly. This definition consists of a list of keywords and a description of the delimiters for comments and strings. Two different types of "keywords" can be defined, one being used for standard keywords of ASMs, the other for domains and macros. The latter are added to the definition during the unparsing process. Symbols given in textual form in the output, for example `\land` for $\wedge$, are replaced by the listings environment using "literate programming", defining the textual string and its replacement in the compiled LaTeX-document: `{\\land}{{$\wedge$}}`.

Compilation to Runtime Environment

ConTraST [18] is a transpiler for SDL developed in the Distributed Systems Group at the University of Kaiserslautern. ConTraST translates SDL-96 systems to C++, preserving the structure of the original SDL system and producing readable C++ code. This enables the developer to work with the generated code. To execute the generated system, a runtime environment is provided, which, compiled together with the generated code, produces an executable.

The runtime environment (SdlRE) is based directly on an extracted dynamic semantics of SDL, which covers features of SDL-96. The ASM rules of the SVM are manually translated into a C++ representation. The resulting runtime environment closely resembles the dynamic semantics of SDL.

ConTraST supports SDL profiles by allowing the user to deactivate language features of SDL. Given an extracted dynamic semantics by the SDL-profile tool based on the deactivated features, we can generate a reduced version of the SDL runtime environment, resulting in a more efficient execution of the generated code. Currently, the SDL-profile tool generates the runtime environment semi-automatically. The resulting code can be adapted and optimised by the developer of the compiler.

ConTraST defines its own data structures for the abstract syntax of SDL and the runtime environment. Domains of the SVM are mapped to C++ classes in the runtime environment, functions and relations of the SVM to class methods. The SDL-profile tool generates C++ code for rules and derived functions of the SVM compatible to the data structures of ConTraST. In the following paragraphs, we provide an overview of the translation of several ASM rules, performed by the SDL-profile tool.

Update. An update of a location in the SVM is translated to an assignment in C++. If the location consists of a function name f and a single parameter x, the function is defined in the class corresponding to the type of the parameter, and the assignment is to x->f. For locations with multiple parameters, the C++ map class has to be used. However, this case does not occur in the subset of the dynamic semantics that is translated to C++.

ASM	**C++**
$f := t$	$f = t;$
$f(x) := t$	$x{-}{>}f = t;$

ConTraST defines data structures for sets and sequences of elements, using templates. These templates define special methods for modifying these containers, which we use when updating locations holding sets/sequences. For example, assigning the empty set/sequence to such a location is translated to a call of the method *clear* of the corresponding class. Other methods defined for sets and sequences include adding and deleting elements, and checking if an element is contained in a set/sequence.

$f := \emptyset$	$f->clear();$
$f(x) := \emptyset$	$x->f->clear();$

Parallel Update. Parallel updates with the **do-forall**-rule of ASMs can be translated to C++ by using iterators from the standard template library (STL). The constraint of the **do-forall**-rule always has the form $x \in exp \wedge g$ - in some cases just $x \in exp$ -, where *exp* is an expression generating a set. Let d, a parametrised type based on STL template set, be the type of expression *exp*[4]. The **do-forall**-rule is translated as follows:

do forall x: $x \in exp \wedge g$	d:: *iterator* $x = exp.begin();$
R	**while** $(x$!= $exp.end())$ {
enddo	**if** (g) { `/* rule body */` }
	x++;
	}

The translation defined above replaces the parallel update of the ASM with a sequential update in C++. Care must be taken that assignments in an iteration step do not affect subsequent iteration steps. In the dynamic semantics of SDL, locations modified during parallel updates are generally parametrised by variable x. Therefore, the sequentialisation described above is valid.

Note that in the translation defined above, the scope of variable x goes beyond the rule body R. Usually this causes no problems, but if variable x is used in rules executed within the same scope, the variable definitions will clash. This can be avoided by enclosing variable definition and rule body inside the C++ scope operators { and }.

Choose. ConTraST implements the **choose**-rule by defining a take-operator on domains (classes in ConTraST), and by keeping track of the elements of a class. We translate **choose**-rules that select an element from a domain d (the type of expression *exp*) to a local variable of type d, which is initialised using the take-operator. If no element is found, take returns *undefined* (a null pointer), and the rule body R is not executed.

choose x: $x \in exp$	$d*$ $x = exp->take();$
R	**if** $(x$!= *undefined*) {
endchoose	`/* rule body */`
	}

In cases where a guard g further constrains the domain, special *choose_if* functions are introduced, which take a predicate as argument. Currently, these predicates have to be implemented manually. The translation generates a generic "PRED" macro, together with the guard g, as an argument to *choose_if*.

[4] d is computed by the function *getDomainFromExpr*.

```
choose x: x ∈ exp ∧ g
  R
endchoose
```

```
d* x = exp->choose_if(PRED(g));
if (x != undefined) {
  /* rule body */
}
```

Extend. ConTraST implements domains of the SVM as classes. Extending a domain with an element translates to the creation of an object of the corresponding class.

```
extend d with x
  R
endextend
```

```
d* x = new d();
/* rule body */
```

Example. Below is the translation of the FORWARDSIGNAL rule macro to C++. The rule macro is mapped to a method of class *Link* (line 1). The **let**-rule is replaced by a local variable definition (line 3). *Delete* and *Insert* are methods of class *Gate*, and are called on the gates passed as the last parameter of DELETE and INSERT in the ASM (lines 5,6). Deleting elements from the set *viaArg* is done with the function *erase* defined on sets. The update of *viaArg* in the ASM rule is split up into two method calls in the C++ code (line 7,8).

```
1  ForwardSignal ≡
2    if Self.from.queue ≠ empty then
3      let si = Self.from.queue.head in
4        if Applicable(si.signalType, si.toArg, si.viaArg, Self.from, Self) then
5          DELETE(si, Self.from)
6          INSERT(si, now+Self.delay, Self.to)
7          si.viaArg := si.viaArg \ {Self.from.gateAS1.nodeAS1ToId,
8                                    Self.channelAS1.nodeAS1ToId}
9        endif
10     endlet
11   endif
```

```
1  void Link::ForwardSignal(void) {
2    if (this->from->queue->empty()==false) {
3      SignalInst* si = this->from->queue->head();
4      if (Applicable(si->signalType, si->toArg, si->viaArg, this->from,this)) {
5        this->from->Delete(si);
6        this->to->Insert(si, (now) + (this->delay));
7        si->viaArg.erase(this->from->Gate_name);
8        si->viaArg.erase(this->Agent_name);
9      }
10   }
11 }
```

6.4 Implementation of the SDL-Profile Tool

6.4.1 Executing the SDL-Profile Tool

The SDL-profile tool accepts several command line options to configure the extraction process. The reduction profile and the output format can be set, iteration and generation of proof obligations can be triggered, and steps of the SDL-profile tool can be

skipped. Table 6.3 shows the most important command line options of the SDL-profile tool. Executing the tool with option `-h` prints the complete list of options.

-c	Disables the rewrite rules of the clean step.
-l	Generate proof obligations during remove step.
-p	Pretty print only, skip normalise, remove and clean steps.
-f file	File containing the reduction profile.
-r type	Set output format to `asm`, `sdlre`, `latex` or `logic`.
-t	Iterate removal, for example to remove unreferenced definitions.

Table 6.3: Important command line options of the SDL-profile tool

The ASM specification is read from standard input, and the result is written to standard output. The SDL-profile tool can therefore be used in a sequence of commands by redirecting the output (for example, using pipes). If the selected output format is `asm`, the specification can go through repeated executions of the SDL-profile tool.

6.4.2 Parsing the Reduction Profile

A reduction profile is a list of names together with information about default values (see Section 5.5). Reduction profiles have a simple syntax. Names following token `True:` have default value `true`, names after token `False:` have default value `false`, and names after token `Undef:` have default value `undefined` or $\emptyset$. All names specified before any of these tokens have default value `undefined`/$\emptyset$. Additionally, literal names and their replacement can be specified in the reduction profile, with the notation *litold/litnew*. All occurrences of *litold* on the right hand side of update rules are replaced by *litnew*. This can be utilised to skip agents modes. For example, to skip priority input selection, we include `selectPriorityInput/selectInput` in the reduction profile.

Figure 6.2 shows the reduction profile for the save feature of SDL.

```
    d-Save-signalset
False:
    f-SignalSaved
```

Figure 6.2: Reduction profile for save feature (tool syntsx)

The SDL-profile tool scans the reduction profile, using *flex*, and stores the names in the sets *cs_true*, *cs_false* and *cs_undef*.

6.4.3 Processing the Semantics Definition

The SDL-profile tool implements the sequence of steps described in Section 6.1. In the *parse* step, the formal semantics definition is read from the standard input by calling

the *bison*-generated function `yyparse()`. The parser, generated from the specification of the concrete and abstract syntax, reads the semantics definition, and stores the root of the resulting abstract syntax tree in the variable `TheSpec` of node type `asmSpec`.

The *normalise* step is triggered by calling the rewrite function on the root of the abstract syntax tree, with rewrite view `normal`. The rewrite rules transform **extend**-rules, and **let**-rules and -expressions. The number of transformations is equal to the sum of all names in **extend**-rules plus the sum of all let-statements $x = t$ in **let**-rules and -expressions.

```
1 if (!opts.pretty) {
2   fprintf(stderr,"␣normalising...\n");
3   TheSpec=TheSpec->rewrite(kc::normal);
4 }
```

After normalising the abstract syntax tree, references to definitions, which are needed for the subsequent *remove* step, are set as described in Section 6.3.4. The abstract syntax tree is traversed twice, first with unparse view `reference` to set the references, then with the unparse view `refcounter` to count the references to each definition. Output is suppressed by a dummy printer function.

```
1 fprintf(stderr,"␣setting␣references ...\n");
2 TheSpec->unparse(dummy_printer, reference);
3 TheSpec->unparse(dummy_printer, refcounter);
```

On the normalised abstract syntax tree with references set, the remove function is called in the *remove* step. Remove traverses the tree and returns an extracted copy of the abstract syntax tree, as described in Sections 6.3.2 and 6.3.3.

On the resulting copy of the abstract syntax tree, the transformations of the *clean* step are performed, by calling the rewrite function with rewrite view `clean` (see Section 6.3.5). The number of transformations performed is variable, but always terminates since each transformation leads to a smaller abstract syntax tree. The *clean* step is skipped if the corresponding option is set (see Section 6.4.1).

The copy of the abstract syntax tree returned by the remove function does not contain any references to definitions. The references are set again after the clean step. Additionally, the unparse function is called with view `unreferenced`, filling the set *it_undef* with unreferenced definitions that can be removed in subsequent iteration steps.

```
1  asmSpec TheOldSpec = TheSpec;
2  TheSpec = remove(TheSpec);
3
4  if (opts.clean) {
5    TheSpec = TheSpec->rewrite(clean);
6  }
7
8  TheSpec->unparse(dummy_printer, reference);
9  TheSpec->unparse(dummy_printer, refcounter);
10 TheSpec->unparse(dummy_printer, unreferenced);
```

If the iteration option is set, the SDL-profile tool calculates a new reduction profile in each step, and repeats the *remove* and *clean* steps until the new reduction profile

is empty. The new reduction profile is contained in the sets *it_undef*, *it_true* and *it_false*. These sets are filled when resetting references, as described above, and when traversing the abstract syntax tree with unparse view `iterate`. Included in the new reduction profile are derived functions that equate to `false`, `true`, `undefined` and ∅, functions with empty target domains, and rule macros that are equivalent to `skip`. The old reduction profile is deleted, since it does not affect the subsequent *remove* step, and replaced by the new reduction profile in the next iteration.

```
1  do {
2    nextstep = false;
3
4    /** remove, clean, reset references **/
5
6    if (opts.iterate) {
7      TheSpec->unparse(dummy_printer, kc::iterate);
8
9      /* calculate reduction profile for subsequent step */
10     if (!it_undef->empty() || !it_true->empty() || !it_false->empty()) { nextstep = true; }
11     delete cs_undef; delete cs_true; delete cs_false;
12     cs_undef = it_undef; cs_true = it_true; cs_false = it_false;
13     it_undef = new std::set<casestring>(); it_true = new std::set<casestring>(); it_false = new
           std::set<casestring>();
14   }
15 } while (opts.iterate && nextstep);
```

After the *iterate* step, the resulting abstract syntax tree is transformed to a textual representation, and written to the standard output, as described in Section 6.3.7.

```
1  if (opts.type == f_asm) {
2    TheSpec->unparse(prettyprinter, kc::ast2asm);
3  } else { if (opts.type == f_latex) {
4    TheSpec->unparse(prettyprinter, kc::ast2latex);
5  } else { if (opts.type == f_sdlre) {
6    TheSpec=TheSpec->rewrite(kc::sdlre);
7    TheSpec->unparse(prettyprinter, kc::ast2sdlre);
8  } else {if (opts.type == f_logic) {
9    TheSpec->unparse(prettyprinter, kc::ast2logic);
10 } } } }
```

6.5 Application of the SDL-Profile Tool

Given an ASM formal semantics definition and a reduction profile, the SDL-profile tool generates a reduced formal semantics definition in the original format. In order to validate the extraction process, we compare the original semantics definition with the reduced version. For this, we have used graphical diff-based tools (for example, tkdiff) to highlight the differences between the versions. Using the SDL-profile tool, we have created reduction profiles for several language modules, such as timer, exception, save, composite state and inheritance. We have also created reduction profiles for language profiles like SDL+ and Core, resulting in formal semantics definitions that, with small modifications, match these SDL profiles.

ASM Listings 6.12 and 6.13 show an excerpt of the formal semantics definition before and after applying the SDL-profile tool, using a reduction profile for SDL exceptions (see Figure 5.9). The reduction profile contains, besides other function and macro names, the function name *currentExceptionInst*, which is interpreted as *undefined* in the context below. Therefore, the predicate *false* holds for the guard of the **if**-rule, and the first part of the **if**-rule is removed.

```
1  SelectTransitionStartPhase ≡
2    if ( Self.currentExceptionInst ≠ undefined) then
3      Self.agentMode3 := selectException
4      Self.agentMode4 := startPhase
5    elseif ( Self.currentStartNodes ≠ ∅ ) then
6      ...
7    else
8      ...
9    endif
```

Listing 6.12: Macro SelectTransitionStartPhase before reduction

```
1  SelectTransitionStartPhase ≡
2    if ( Self.currentStartNodes ≠ ∅ ) then
3      ...
4    else
5      ...
6    endif
```

Listing 6.13: Macro SelectTransitionStartPhase after reduction

6.5.1 Extracting a Profile without Timers

In this section, we present the results of applying the SDL-profile tool to the formal semantics definition of SDL, with a partial reduction profile for the timer feature. Exemplarily, we restrict the reduction profile for timers to the domain Set, with default value ∅ (see Figure 6.3). Domain Set holds the behaviour primitives for setting timers, and is always empty for specifications without timers.

Figure 6.3: Excerpt of reduction profile for timer feature

We examine the extraction for the rule macro for evaluating behaviour primitives, Eval, and related rule macros EvalSet and SetTimer (see Listing 6.14).

```
1  Eval(a: Action) ≡
2    if a ∈ Var then EvalVar(a)
3    elseif ...
13   elseif a ∈ Set then EvalSet(a)
14   elseif ...
28   endif
29
30 EvalSet(a: Set) ≡
```

```
31   SetTimer(a.s-Timer, values(a.s-ValueLabel-seq, Self), semvalueReal(value(a.s-TimeLabel,
         Self)))
32   Self.currentLabel := a.s-ContinueLabel
33
34 static duration: Timer → Duration
35
36 SetTimer(tm: Timer, vSeq : Value*, t:[Time]) ≡
37   let tmi = mk-TimerInst(Self.self, tm, vSeq) in
38     if t = undefined then
39       Self.inport.schedule := insert(tmi, now + tm.duration, delete(tmi, Self.inport.schedule))
40       tmi.arrival := now + tm.duration
41     else
42       Self.inport.schedule := insert(tmi, t, delete(tmi, Self.inport.schedule))
43       tmi.arrival := t
44     endif
45   endlet
```

Listing 6.14: Rule macros for setting timers [42]

Iteration Step 1

Remove: Domain Set defaults to the empty set, and the tool evaluates the expression $a \in$ Set as false. Remove subsequently removes the corresponding **then**-part, containing a rule macro call to EvalSet.

```
1  Eval(a: Action) ≡
2    if a ∈ Var then EvalVar(a)
3    elseif ...
13   elseif a ∈ Set (False) then EvalSet(a)
14   elseif ...
28   endif
```

Rule macro EvalSet has a parameter of type Set. Since no elements of Set exist according to the reduction profile, the rule body is reduced to `skip` by the tool. Rule macro calls to EvalSet would contradict the reduction profile, and lead to inconsistencies.

```
1 EvalSet(a: Set) ≡
2   skip
```

Iterate: In the iterate step, the SDL-profile tool detects that no calls to rule macros EvalSet and SetTimer exist in the reduced semantics definition, and includes the rule macros in the set of undefined names.

Iteration Step 2

Remove: In the second remove step, the tool removes the definitions of EvalSet and SetTimer, for which no rule macro calls exist. Therefore, no other parts of the semantics definition are affected.

Iterate: In the iterate step, the SDL-profile tool detects that the static function *duration* is unreferenced. The function definition is removed in a subsequent and final iteration step.

Listing 6.15 shows the formal semantics definition, extracted by the SDL-profile tool with reduction profile in Figure 6.3. One line of specification was removed from the EVAL rule macro, leading to 14 removed lines of unreferenced rule macros and functions.

```
1  EVAL(a: ACTION) ≡
2    if a ∈ VAR then EVALVAR(a)
3    elseif ...
27   endif
```

Listing 6.15: Extracted semantics definition

6.5.2 Extracting a Profile without Inheritance

In this section, we present the results of applying the SDL-profile tool to the formal semantics definition of SDL, with a partial reduction profile for inheritance. We restrict the reduction profile to a single function: *inheritedStateNode* (see Figure 6.4). Controlled function *inheritedStateNode*(s) returns for a state node s the state node s' that s inherits from, or `undefined`. In the reduction profile, *inheritedStateNode* defaults to `undefined`. This is an invariant used in the reduction. A correctness criteria is that no assignment unequal to `undefined` remains in the semantics definition - for any such assignment a proof obligation is generated.

We show how several parts of the SVM are reduced by iterated application of the *remove* and *clean* steps.

Figure 6.4: Excerpt of reduction profile for inheritance feature

Listing 6.16 shows an excerpt of the SVM, which defines derived functions for inheritance and refinement. *DirectlyInheritsFrom*, *InheritsFrom*, *DirectlyRefinedBy*, *DirectlyInheritsFromOrRefinedBy* and *InheritsFromOrRefinedBy* are predicates for the direct or indirect inheritance or refinement relation between two state nodes. Derived functions *selectNextStateNode* and *selectInheritedStateNode* are used during transition selection and return state nodes in a specific order. Derived function *inheritedStateNodes* is the transitive closure of *inheritedStateNode*.

```
1  DirectlyInheritsFrom(sn1: StateNode, sn2: StateNode): BOOLEAN =def
2    if sn2.parentStateNode = undefined then False
3    elseif sn1.parentStateNode = undefined then False
4    elseif sn2.parentStateNode ∈ sn1.parentStateNode.inheritedStateNodes ∧sn1.stateName = sn2.
         stateName ∧(¬ ∃sn3 ∈ StateNode:
5        if sn3.parentStateNode = undefined then
6          False
7        else
8          sn3.parentStateNode ∈ sn1.parentStateNode.inheritedStateNodes ∧
9          sn2.parentStateNode ∈ sn3.parentStateNode.inheritedStateNodes ∧
10         sn3.stateName = sn2.stateName) then True
11   else False
```

```
12   endif
13
14 InheritsFrom(sn1: StateNode, sn2: StateNode): Boolean =def
15   if sn2.parentStateNode = undefined then False
16   elseif sn1.parentStateNode = undefined then False
17   else
18     sn2.parentStateNode ∈ sn1.parentStateNode.inheritedStateNodes ∧sn1.stateName = sn2.
           stateName
19   endif
20
21 DirectlyRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
22   sn2.parentStateNode = sn1
23
24 DirectlyInheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
25   DirectlyRefinedBy(sn1, sn2) ∨ DirectlyInheritsFrom(sn1, sn2)
26
27 InheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
28   DirectlyInheritsFromOrRefinedBy(sn1, sn2) ∨∃ sn3 ∈ { sn ∈ StateNode :
         DirectlyInheritsFromOrRefinedBy (sn1, sn) } : InheritsFromOrRefinedBy(sn3, sn2)
29
30 selectNextStateNode(snSet: StateNode−set): [StateNode] =def
31   let sn = take({sn1 ∈ snSet: (¬ ∃ sn2 ∈ snSet: InheritsFromOrRefinedBy(sn1, sn2))}) in
32     if sn = undefined then undefined
33     elseif ∃ sn1 ∈ snSet: DirectlyInheritsFrom(sn1, sn) ∨ sn = sn1.inheritedStateNode then
34       selectNextStateNode(snSet \ {sn})
35     else sn
36     endif
37   endlet
38
39 inheritedStateNodes(sn: StateNode): StateNode−set =def
40   if sn.inheritedStateNode = undefined then ∅
41   else {sn.inheritedStateNode} ∪ sn.inheritedStateNode.inheritedStateNodes
42   endif
43
44 mostSpecialisedStateNode(sn: StateNode): StateNode =def
45   let sn1 = take({sn2 ∈ StateNode: InheritsFrom(sn2, sn)}) in
46     if sn1 = undefined then sn else sn1.mostSpecialisedStateNode endif
47   endlet
48
49 selectInheritedStateNode(sn: StateNode, snSet: StateNode−set): [StateNode ]=def
50   take({sn1 ∈ snSet: DirectlyInheritsFrom(sn,sn1)})
```

Listing 6.16: Derived functions for inheritance and refinement [42]

Iteration Step 1

Remove: Function *inheritedStateNode* defaults to `undefined`, therefore the tool evaluates the guard of the following **if**-expression to true. The **if**-expression is replaced by the **then**-part, which is the empty set.

```
inheritedStateNodes(sn: StateNode): StateNode−set =def
  if sn.inheritedStateNode = undefined then ∅
     ──────────────────────────────────
                  True
  else {sn.inheritedStateNode} ∪ sn.inheritedStateNode.inheritedStateNodes
  endif
```

In *selectNextStateNode*, the SDL-profile tool determines that *sn* does not equal *sn1.inheritedStateNode*, which is `undefined`. Note that this only holds because if *sn* is `undefined`, the **then**-branch of the **if**-expression holds. The expression has no effect on the result of the disjunction and is removed by the SDL-profile tool.

selectNextStateNode(*snSet*: *StateNode−set*): [*StateNode*] $=_{\text{def}}$
  let *sn* = *take*({*sn1* ∈ *snSet*: (¬ ∃ *sn2* ∈ *snSet*: *InheritsFromOrRefinedBy*(*sn1*, *sn2*))}) **in**
    if *sn* = *undefined* **then** *undefined*
    elseif ∃ *sn1* ∈ *snSet*: *DirectlyInheritsFrom*(*sn1*, *sn*) ∨ $\underbrace{sn = sn1.inheritedStateNode}_{\text{False}}$ **then**
      selectNextStateNode(*snSet* \ {*sn*})
    else *sn*
    endif
  endlet

Iterate: For the next iteration, *inheritedStateNodes* is included in the reduction profile, defaulting to the empty set.

inheritedStateNodes(*sn*: *StateNode*): *StateNode−set* $=_{\text{def}}$
  ∅

Iteration Step 2

Remove: The second iteration of remove affects derived functions *DirectlyInheritsFrom* and *InheritsFrom*. Both contain guards with conjunctions where one subexpression is evaluated as false by the tool. The condition that a state node is contained in the set of inherited state nodes of another state node is false, since *inheritedStateNodes* returns the empty set. The corresponding parts of the SVM are therefore removed.

DirectlyInheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): Boolean $=_{\text{def}}$
  if *sn2.parentStateNode* = *undefined* **then** *False*
  elseif *sn1.parentStateNode* = *undefined* **then** *False*
  elseif $\underbrace{sn2.parentStateNode \in sn1.parentStateNode.inheritedStateNodes}_{\text{False}}$ ∧ *sn1.stateName* = *sn2.*
      stateName ∧(¬ ∃*sn3* ∈ *StateNode*:
      if *sn3.parentStateNode* = *undefined* **then**
        False
      else
        sn3.parentStateNode ∈ *sn1.parentStateNode.inheritedStateNodes* ∧
        sn2.parentStateNode ∈ *sn3.parentStateNode.inheritedStateNodes* ∧
        sn3.stateName = *sn2.stateName*) **then** *True*
  else *False*
  endif

InheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): Boolean $=_{\text{def}}$
  if *sn2.parentStateNode* = *undefined* **then** *False*
  elseif *sn1.parentStateNode* = *undefined* **then** *False*
  else
    $\underbrace{sn2.parentStateNode \in sn1.parentStateNode.inheritedStateNodes}_{\text{False}}$ ∧ *sn1.stateName* = *sn2.*
      stateName
  endif

Clean: The rewrite rules of the clean step match for *if*-expressions that return the same result for every alternative. The *if*-expression is replaced with the result common to all alternatives.

DirectlyInheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): BOOLEAN $=_{\text{def}}$
  if *sn2.parentStateNode* = *undefined* **then** *False*
  elseif *sn1.parentStateNode* = *undefined* **then** *False*
  else *False* **endif**

InheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): BOOLEAN $=_{\text{def}}$
  if *sn2.parentStateNode* = *undefined* **then** *False*
  elseif *sn1.parentStateNode* = *undefined* **then** *False*
  else *False* **endif**

Iterate: For the next iteration, *DirectlyInheritsFrom* and *InheritsFrom* are included in the reduction profile, defaulting to false.

DirectlyInheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): BOOLEAN $=_{\text{def}}$
  False

InheritsFrom(*sn1*: *StateNode*, *sn2*: *StateNode*): BOOLEAN $=_{\text{def}}$
  False

Iteration Step 3

Remove: Derived function *DirectlyInheritsFrom* is false, as specified in the reduction profile. It is removed by the SDL-profile tool since it has no effect on the result of the disjunction.

DirectlyInheritsFromOrRefinedBy(*sn1*: *StateNode*, *sn2*: *StateNode*): BOOLEAN $=_{\text{def}}$
  DirectlyRefinedBy(*sn1*, *sn2*) $\vee$ $\underbrace{\textit{DirectlyInheritsFrom}(sn1, sn2)}_{\text{False}}$

DirectlyInheritsFrom is always false, therefore no element in *snSet* can exist for which the derived function holds. The corresponding part of the **if**-expression is removed.

selectNextStateNode(*snSet*: *StateNode*−*set*): [*StateNode*] $=_{\text{def}}$
  let *sn* = *take*({*sn1* ∈ *snSet*: (¬ ∃ *sn2* ∈ *snSet*: *InheritsFromOrRefinedBy*(*sn1*, *sn2*))}) **in**
    if *sn* = *undefined* **then** *undefined*
    elseif $\underbrace{\exists sn1 \in snSet : \textit{DirectlyInheritsFrom}(sn1, sn)}_{\text{False}}$ **then**
      selectNextStateNode(*snSet* \ {*sn*})
    else *sn*
    endif
  endlet

In the derived functions *selectInheritedStateNode* and *mostSpecialisedStateNode* contain set constructors with a condition that always evaluates to false, according to the reduction profile in iteration step 3. The tool replaces the set constructor with the empty set.

selectInheritedStateNode(*sn*: *StateNode*, *snSet*: *StateNode*−*set*): [*StateNode*]$=_{\text{def}}$
  take($\underbrace{\{sn1 \in snSet : \textit{DirectlyInheritsFrom}(sn, sn1)\}}_{\emptyset}$)

```
mostSpecialisedStateNode(sn: StateNode): StateNode =def
  let sn1 = take(underbrace({sn2 ∈ StateNode : InheritsFrom(sn2, sn)}, ∅)) in
    if sn1 = undefined then sn else sn1.mostSpecialisedStateNode endif
  endlet
```

Clean: Two rewrite rules apply to *selectNextStateNode* during the clean step. First, the expression of the form '**if** $x = a$ **then** a **else** x **endif**' is replaced with 'x'. Second, the expression of the form '**let** $x = t$ **in** x **endlet**' is replaced by the expression 't'.

```
selectNextStateNode(snSet: StateNode−set): [StateNode] =def
  let sn = take({sn1 ∈ snSet: (¬ ∃ sn2 ∈ snSet: InheritsFromOrRefinedBy(sn1, sn2))}) in
    if sn = undefined then undefined
    else sn
    endif
  endlet
```

The *take* operator on the empty set evaluates to `undefined`, and is rewritten accordingly in the clean step. The expression of the form '**let** $x = t$ **in if** $x = t$ **then** e_1 **else** e_2 **endif endlet**' is rewritten to e_1.

```
selectInheritedStateNode(sn: StateNode, snSet: StateNode−set): [StateNode]=def
  underbrace(take(∅), Undef)

mostSpecialisedStateNode(sn: StateNode): StateNode =def
  let sn1 = take(∅) in
    if sn1 = undefined then sn else sn1.mostSpecialisedStateNode endif
  endlet
```

Iteration: After cleanup, *selectInheritedStateNode* is defined as `undefined` and is included in the reduction profile for the next iteration, which we do not show here since it does not affect parts of the semantics definition included in Listing 6.16.

Listing 6.17 is the final result of the extraction for the derived functions for inheritance and refinement, starting from a reduction profile containing only the function *inheritedStateNode*. The 50 lines of definition from Listing 6.16 have been reduced to 14 lines of definition by the SDL-profile tool. Further reduction can be achieved by replacing predicate *DirectlyInheritsFromOrRefinedBy* and *mostSpecialisedStateNode* by their definitions.

```
1  DirectlyRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
2    sn2.parentStateNode = sn1
3
4  DirectlyInheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
5    DirectlyRefinedBy(sn1, sn2)
6
7  InheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def
8    DirectlyInheritsFromOrRefinedBy(sn1, sn2) ∨∃ sn3 ∈ { sn ∈ StateNode :
         DirectlyInheritsFromOrRefinedBy (sn1, sn) } : InheritsFromOrRefinedBy(sn3, sn2)
9
10 selectNextStateNode(snSet: StateNode−set): [StateNode] =def
11   take({sn1 ∈ snSet: (¬ ∃ sn2 ∈ snSet: InheritsFromOrRefinedBy(sn1, sn2))})
12
```

```
13 mostSpecialisedStateNode(sn: StateNode): StateNode =def
14   sn
```

Listing 6.17: Extracted derived functions for inheritance and refinement

In other parts of the SVM, 36 lines of definition are removed, 29 for transition selection and 7 for entering state nodes. Furthermore, several expressions are replaced with less complex subexpressions. Adding the function *stateNodesToBeSpecialised* to the reduction profile in Figure 6.4 leads to the removal of 33 further lines of definition for the creation of state nodes. Overall, 105 lines of definition are removed, about 4% of the dynamic semantics.

6.5.3 Size of Extracted SDL Profiles

We have used the SDL-profile tool to extract formal semantics definitions for several SDL profiles. Apart from the profiles described in Section 4.3, we have extracted a number of profiles where only one feature was removed from the formal semantics definition. The number of removed lines of specification is a measure for the complexity of the removed feature. Table 6.4 shows the size, in lines of definition (LoDef), of the extracted semantics definition for ten language features. The profiles were extracted from the dynamic semantics of SDL, including the compilation function and excluding the data semantics, with a size of 2731 lines of definition.

	Feature	**LoDef**	**Δ**	**ΔΔ**
	Complete dynamic semantics	*2731*		
1	Procedure	2480	251	
2	Exception	2510	221	
3	Continuous signal	2610	121	
4	Inheritance	2626	105	
5	Exit composite states	2627	104	
6	Priority input	2641	90	
7	Timer	2647	84	
8	Spontaneous transition	2675	56	
9	Entry-/Exit-procedure	2702	29	
10	Save	2715	16	
	2, 4, 7	2325	406	4
	2, 4–5, 7–10	2123	608	7
	1, 9	2480	251	29
	1–10	1744	987	88

Table 6.4: Size of extracted SDL profiles

The most complex language feature are procedures, which consume almost 10% of the formal semantics definition. This includes the creation and initialisation of procedure graphs, procedure calls and returns, storing and restoring procedure control blocks and parts of transition selection.

The reduction profiles for these features can be combined, for example by repeated application of the SDL-profile tool, to yield further SDL profiles. The number of removed lines of definition is equal or smaller to the sum of removed lines for the individual features, since parts of the dynamic semantics are removed in more than one profile. The difference between the sum of removed lines for the individual features, and the number of removed lines of the combined features, is a measure for feature interaction. For example, Table 6.4 shows that the feature interaction between exception (2), inheritance (4) and timer (7) is small. For procedure (1) and entry-/exit-procedure (9), feature interaction is maximal, since entry- and exit-procedures of composite states rely on procedures. The parts of definition removed for entry-/exit-procedure are a subset of the parts of definition removed for procedure.

Feature	**LoDef**	**Δ**
Complete dynamic semantics	*2731*	
Dynamic	2441	290
SDL Profile for UML	2306	425
$Static_2$	2053	678
SDL+	1669	1062
$Static_1$	1665	1066
Core	996	1735

Table 6.5: Size of extracted SDL profiles

Table 6.5 shows the size of the extracted formal semantics definition for several SDL profiles introduced in Section 4.3. The extracted dynamic semantics of the smallest profile (*Core*) contains about a third of the lines of definition of the complete dynamic semantics. 24 lines of definition of *Core* belong to features not contained in the profile, but could not be removed by the SDL-profile tool.

For the SDL profiles defined by ConTraST, the extracted semantics of $Static_2$ is significantly reduced (678 lines of definition) by excluding exception, and dynamic aspects of the language (procedures, creating and stopping processes). The extracted semantics of $Static_1$ has almost 400 additional removed lines of definition compared to $Static_2$, by excluding inheritance and several transition kinds.

6.6 Summary and Conclusions

In this chapter, we have described the implementation of the SDL-profile tool, and its application to the dynamic semantics of SDL, to extract a number of SDL profiles. We have applied the tool to extract SDL profiles with single features removed, and identified features with a large impact on the dynamic semantics, such as procedures, exceptions, continuous signals, and inheritance. We have extracted the dynamic semantics for several SDL profiles described in Section 4.3, including *SDL+*, the *SDL Profile for UML*, and all SDL profiles introduced by ConTraST.

The reduction of the dynamic semantics achieved for smaller profiles such as *Core*,

Static$_1$ and *SDL+* is significant. However, the extracted semantics definitions are still large, since the general structure, introduced in order to cope with the complexity of SDL-2000, remains intact. For example, the extracted semantics of *Core* still has about 1000 lines of definition.

The format of the reduction profile and the extraction process have been kept simple, with only a small set of default values. However, these default values have proved sufficient to extract a formal semantics for all profiles mentioned in this chapter. The number of dead ASM rules (rules not executed for any SDL specification in a given SDL profile, see Section 5.3.2) that could not be removed by the SDL-profile tool is small compared to the number of lines removed. For example, *Core* contains 24 lines of dead ASM rules not removed by the SDL-profile tool, compared to 1735 lines removed.

6.7 Future Work

Applying the Extraction to Further Modelling Languages

We have defined the extraction for Abstract State Machines, and applied it to SDL, attaining good results. The extraction can be applied to every formal semantics that is defined using ASMs[5]. An open question is whether good results can be attained for these semantics definitions, given the small set of default values sufficient for the dynamic semantics of SDL, or with an extended set of default values.

The most common constructs in ASM specifications are parallel update rules, usually guarded by one or more **if**-rules. The effectiveness of the extraction relies on the identification of domains, functions and predicates that are static for all possible runs of an ASM for a given language profile, and consequently, the evaluation of guards of **if**-rules as true or false. Usually, such domains (functions, predicates) are related to the concrete/abstract syntax of language features excluded in a profile.

```
1  state(entry, exit, do(A), defer)
2
3  deferrable(e) = true ⇔ enabled(e) = ∅ ∧ e ∈ defer(deepest)
4  releasable(e) = true ⇔ e ∈ deferQueue ∧ enabled(e) ≠ ∅
5
6  Rule Transition Selection
7    choose e : dispatched(e) ∨ releasable(e)
8      choose trans ∈ FirableTrans(e)
9        stateMachineExecution(trans)
10     if deferrable(e) then insert(e, deferQueue)
11     if releasable(e) then delete(e, deferQueue)
```

Listing 6.18: Excerpt from an ASM semantics for UML statemachines [5]

UML does not have a complete, standardised formal semantics definition. Non-standardised formal semantics definitions using ASMs have been defined for some aspects of UML, such as statemachines [5], activity diagrams [4], and classes/objects,

[5]Because no standardised syntax exists for ASMs, usually some syntactical adjustments are necessary.

relationships and actions [49]. Listing 6.18 shows an excerpt from an ASM semantics definition for UML statecharts [5]. Line 1 shows the abstract syntax of plain states, containing a set of deferred events *defer* that is empty for profiles that exclude deferred events. The extraction subsequently determines predicate *deferrable* (line 3) as false for all events ($e \in defer(deepest)$ is evaluated as false, and therefore the conjunction and equivalence), and subsequently removes lines 3 and 10. An empty domain *defer* leads to an empty *deferQueue*, and a subsequent removal of lines 4 ($e \in deferQueue$ evaluated as false) and 11 by the extraction.

7 Conclusions

7.1 State of the Art

7.1.1 Combined Use of Modelling Languages

The harmonisation of SDL and UML, and possibly further modelling languages, has been an ongoing work in research and industry. For several years, these languages have influenced each other's development, incorporating features from the other language into new language standards. In [57], a proposal for a true harmonisation of UML, SDL and other languages was put forward. This proposal aimed at defining a common semantic core for UML, SDL and VHDL/SystemC.

In order to define a common semantic core, a rigorous comparison of the languages involved is necessary. We have performed this comparison for SDL and a corresponding subset of UML on a syntactic and semantic level. This comparison resulted in a mapping between the abstract syntax representations of SDL (abstract grammars) and UML (meta-models), a common abstract syntax, and a comparison of the semantics for the specification of behaviour in both languages. From this comparison we concluded that, while SDL and UML are syntactically similar only for the specification of structural aspects, both languages have similar semantics for behavioural elements (state machines and process graphs), with small semantic differences for some language features.

Given the similarities between SDL and UML on the semantic level, the approach of the ITU - the standardisation body for SDL - was to use SDL as the semantic core for UML. This approach mapped a subset of UML to the dynamic semantics of SDL, thus reusing its formal semantics definition, and resolving the semantic variation points of UML. With the mature UML profile mechanism introduced in the new UML 2.0 standard [52], this mapping was integrated into UML as a UML profile, defining stereotypes for existing UML meta-model classes. In order to carry over the formality of SDL to the subset of UML, we surveyed a formalisation approach for the UML Profile for SDL. This formalisation produced feedback for the profile definition, leading to the correction of several errors and imprecise statements. From the formalisation, the meta-model constraints specified with OCL are planned to be included in the final standard.

With the standardisation of the UML Profile for SDL [33], SDL and UML can be used together. A developer can specify a system design in UML, with the UML Profile for SDL applied, automatically checking the constraints defined on the model. The model can then be translated[1] to SDL for the detailed design phase. To combine

[1] This translation can be automatically derived from a complete formalisation of the profile.

UML and SDL in one model, for example, structural elements of UML with behaviour specification of SDL, we can extract a meta-model for SDL, using our mapping between abstract syntax representations. This SDL meta-model can be integrated with the UML meta-model, using inheritance, and ensuring that no constraints on the UML meta-model are violated[2]. Note that this approach goes beyond the UML profile approach, since UML profiles are not allowed to extend the referenced meta-model.

7.1.2 Language Profiles and Modular Language Definition

Modelling languages like UML and SDL are complex and expressive, leading to language definitions that are long and take substantial effort to be understood completely, and that have a limited applicability in domains where tailor-made, domain-specific languages are preferred. Tool support usually covers only a subset of these languages, omitting features that are hard to implement, or unsuitable for the target domain of the tool. Our aim was to develop methods for defining modular language definitions (horizontal modularity), particularly formal language definitions, which enable a tool provider to extract a tailor-made formal language definition for a subset of the language - called *language profile*[3].

We have examined two approaches for generating formal language definitions for language profiles. The *composition approach* splits the language into a *language core* and a set of *language modules*. Language modules encapsulate a *language feature*, and can be composed with the language core to yield a tailor-made language. For imperative languages like C, C♯ and Java, such modular language definitions exist, defining an interpreter with incremental extensions, using Abstract State Machines [29, 63, 6]. For the formal semantics for a language such as SDL, which is used to define distributed, asynchronous systems, a virtual machine approach is more suitable (see Section 2.4). For these kind of formal semantics definitions, it proved difficult to encapsulate language features into language modules, without sacrificing readability.

Therefore, we have followed the *extraction approach*, where, starting from a complete language definition, we extract the formal semantics of a language profile by removing parts of the formal language definition that are not included in the profile. We have formalised this approach for Abstract State Machines [27, 28], and applied it to the formal semantics definition of SDL [42], extracting a number of SDL profiles using a tool that implements the formal definition of the extraction. We have used the extraction approach to extract a language core for SDL, containing only basic features of SDL. While the reduction achieved was significant (two thirds of the original semantics definition), the resulting semantics definition was still large, since the core still contains the infrastructure for advanced SDL features.

[2] For example, an SDL meta-model class State-machine-definition can be integrated into the UML meta-model as a subclass of abstract metaclass *Behavior*.

[3] Named after UML profiles, which are used to specialise UML.

7.2 Future Work

In this thesis, we have focused on the horizontal modularisation of modelling languages. Future work is the provision of sound approaches for the other forms of modularity introduced in Section 1.2: vertical, optional, temporal and hybrid modularity.

To some extent, the extraction approach defined in Chapter 5 can be applied to vertical modularity. Given two languages of a language family, with one language providing an abstract view of the model defined by the other language, the extraction approach can be used to reduce the semantics definition of the more detailed language to aspects relevant to the abstract view. ASM slicing [47, 48] can be applied, identifying locations of interest to the abstract view, and reducing the formal semantics definition to rules that affect these locations directly and indirectly.

For optional modularity, UML 2.0 [52] introduces a mature UML profile mechanism. Using UML profiles, semantic variation points can be resolved, leading to a specialisation of the language. The UML profile concept can be applied to other languages. However, it is only feasible for languages with many semantic variation points.

For temporal modularity, the composition approach to modular language definition, outlined in Section 5.11, would enable the conservative extension of languages by adding new language modules to the language core. For extensions that affect existing modules, criteria for valid modifications of modules must be defined.

A UML Kernel Abstract Grammar

```
/* 1.2 Kernel - Root */

Kernel_Element ::=(=) Kernel_Relationship
        | Kernel_Comment
        | Kernel_NamedElement
        | Kernel_MultiplicityElement
        | Kernel_Slot

Kernel_Relationship ::=(=) Kernel_DirectedRelationship
        | Kernel_Association

Kernel_DirectedRelationship ::=(=) Kernel_ElementImport
        | Kernel_PackageImport
        | Kernel_Generalization
        | Kernel_PackageMerge

Kernel_Comment ::=(::)
        String                                  /* body */
        Kernel_Element_Idenfifier-set           /* annotatedElement */

/* 1.3 Kernel - Namespaces */

Kernel_VisibilityKind ::=(=) PUBLIC | PRIVATE | PROTECTED | PACKAGE

Kernel_NamedElement ::=(=) Kernel_PackageableElement
        | Kernel_Namespace
        | Kernel_TypedElement
        | Kernel_RedefinableElement

Kernel_PackageableElement ::=(=) Kernel_Type
        | Kernel_Constraint
        | Kernel_InstanceSpecification
        | Kernel_Package

Kernel_Namespace ::=(=) Kernel_Classifier
        | Kernel_BehavioralFeature
        | Kernel_Package

Kernel_ElementImport ::=(::)
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* alias */
        Kernel_PackageableElement_Identifier    /* importedElement */


Kernel_PackageImport ::=(::)
        Kernel_VisibilityKind                   /* visibility */
        Kernel_Package_Identifier               /* importedPackage */
```

```
/* 1.4 Kernel - Multiplicities */

Kernel_MultiplicityElement ::=(=) Kernel_StructuralFeature
        | Kernel_Parameter

Kernel_TypedElement ::=(=) Kernel_ValueSpecification
        | Kernel_StructuralFeature
        | Kernel_Parameter

Kernel_Type ::=(=) Kernel_Classifier

/* 1.5 Kernel - Expressions */

Kernel_ValueSpecification ::=(=) Kernel_Expression
                                | Kernel_OpaqueExpression
                                | Kernel_LiteralSpecification
                                | Kernel_InstanceValue

Kernel_Expression ::=(::)
        String                                  /* symbol */
        Kernel_ValueSpecificatioN-set           /* operand */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_OpaqueExpression ::=(::)
        String                                  /* body */
        [String]                                /* language */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_LiteralSpecification ::=(=) Kernel_LiteralBoolean
        | Kernel_LiteralInteger
        | Kernel_LiteralString
        | Kernel_LiteralUnlimitedNatural
        | Kernel_LiteralNull

Kernel_InstanceValue ::=(::)
        Kernel_InstanceSpecification_Identifier /* instance */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_LiteralBoolean ::=(::)
        Boolean                                 /* value */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_LiteralInteger ::=(::)
        Integer                                 /* value */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */
```

```
Kernel_LiteralString ::=(::)
        String                                  /* value */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_LiteralUnlimitedNatural ::=(::)
        UnlimitedNatural                        /* value */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

Kernel_LiteralNull ::=(::)
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */

/* 1.6 Kernel - Constraints */

Kernel_Constraint ::=(::)
        Kernel_Element_Identifier-set           /* constrainedElement */
        Kernel_ValueSpecification               /* specification */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */

/* 1.7 Kernel - Instances */

Kernel_InstanceSpecification ::=(=) Kernel_InstanceSpecification_Concrete
        | Kernel_EnumerationLiteral

Kernel_InstanceSpecification_Concrete ::=(::)
        Kernel_Slot-set                         /* slot */
        [Kernel_ValueSpecification]             /* specification */
        Kernel_Classifier_Identifier-set        /* classifier */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */

Kernel_Slot ::=(::)
        Kernel_ValueSpecification-set           /* value */
        Kernel_StructuralFeature_Identifier     /* definingFeature */

/* 1.8 Kernel - Classifiers */

Kernel_Classifier ::=(=) Kernel_Class
        | Kernel_Association
        | Kernel_DataType

Kernel_RedefinableElement ::=(=) Kernel_Classifier
        | Kernel_Feature

Kernel_Generalization ::=(::)
        Boolean                                 /* isSuitable */
        Kernel_Classifier_Identifier            /* general */

/* 1.9 Kernel - Features */
```

```
Kernel_ParamerDirectionKind ::=(=) IN | INOUT | OUT | RETURN

Kernel_Feature ::=(=) Kernel_StructuralFeature
        | Kernel_BehavioralFeature

Kernel_StructuralFeature ::=(=) Kernel_Property

Kernel_BehavioralFeature ::=(=) Kernel_Operation

Kernel_Parameter ::=(::)
        Kernel_ParamerDirectionKind             /* direction */
        [String]                                /* default */
        [Kernel_ValueSpecification]             /* defaultValue */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        Boolean                                 /* isOrdered */
        Boolean                                 /* isUnique */
        [Kernel_Type_Identifier]                /* type */
        [Kernel_ValueSpecification]             /* upperValue */
        [Kernel_ValueSpecification]             /* lowerValue */

/* 1.10 Kernel - Operations */

Kernel_Operation ::=(::)
        Boolean                                 /* isQuery */
        Kernel_Parameter-set                    /* formalParameter */
        Kernel_Constraint-set                   /* precondition */
        Kernel_Constraint-set                   /* postcondition */
        [Kernel_Constraint]                     /* bodyCondition */
        Kernel_Type_Identifier-set              /* raisedException */
        Kernel_Operation_Identifier-set         /* redefinedOperation */
        Boolean                                 /* is Static */
        Boolean                                 /* isLeaf */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        Kernel_Parameter-set                    /* returnResult */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

/* 1.11 Kernel - Classes */

Kernel_AggregationKind ::=(=) NONE | SHARED | COMPOSITE

Kernel_Class ::=(::)
        Kernel_Property-set                     /* ownedAttribute */
        Kernel_Classifier-set                   /* nestedClassifier */
        Kernel_Operation-set                    /* ownedOperation */
        Boolean                                 /* isAbstract */
        Boolean                                 /* isLeaf */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_Classifier_Identifier-set        /* redefinedClassifier */
        Kernel_Generalization-set               /* generalization */
        Kernel_ElementImport-set                /* elementImport */
```

```
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

Kernel_Property ::=(::)
        Boolean                                 /* isDerived */
        Boolean                                 /* isReadOnly */
        Boolean                                 /* isDerivedUnion */
        Kernel_AggregationKind                  /* aggregation */
        Kernel_Property_Identifier-set          /* subsettetProperty */
        Kernel_Property_Identifier-set          /* refinedProperty */
        [Kernel_ValueSpecification]             /* defaultValue */
        [Kernel_Association_Identifier]         /* associastion */
        Boolean                                 /* isStatic */
        Boolean                                 /* isLeaf */
        Boolean                                 /* isOrdered */
        Boolean                                 /* isUnique */
        Boolean                                 /* isReadOnly */
        [Kernel_VisibilityKind]                 /* visibility */
        [String]                                /* name */
        [Kernel_Type_Identifier]                /* type */
        [Kernel_ValueSpecification]             /* upperValue */
        [Kernel_ValueSpecification]             /* lowerValue */

Kernel_Association ::=(::)
        Boolean                                 /* isDerived */
        Kernel_Property-set                     /* ownedEnd */
        Kernel_Property-set                     /* memberEnd */
        Boolean                                 /* isAbstract */
        Boolean                                 /* isLeaf */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_Classifier_Identifier-set        /* redefinedClassifier */
        Kernel_Generalization-set               /* generalization */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

/* 1.12 Kernel - Datatype */

Kernel_DataType ::=(=) Kernel_Datatype_Concrete
        | Kernel_PrimitiveType
        | Kernel_Enumeration

Kernel_DataType_Concrete ::=(::)
        Kernel_Property-set                     /* ownedAttribute */
        Kernel_Operation-set                    /* ownedOperation */
        Boolean                                 /* isAbstract */
        Boolean                                 /* isLeaf */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_Classifier_Identifier-set        /* redefinedClassifier */
        Kernel_Generalization-set               /* generalization */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */
```

```
Kernel_PrimitiveType ::=(::)
        Boolean                                 /* isAbstract */
        Boolean                                 /* isLeaf */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_Property-set                     /* ownedAttribute */
        Kernel_Operation-set                    /* ownedOperation */
        Kernel_Classifier_Identifier-set        /* redefinedClassifier */
        Kernel_Generalization-set               /* generalization */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

Kernel_Enumeration ::=(::)
        Kernel_EnumerationLiteral-set           /* literal */
        Boolean                                 /* isAbstract */
        Boolean                                 /* isLeaf */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_Property-set                     /* ownedAttribute */
        Kernel_Operation-set                    /* ownedOperation */
        Kernel_Classifier_Identifier-set        /* redefinedClassifier */
        Kernel_Generalization-set               /* generalization */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

Kernel_EnumerationLiteral ::=(::)
        Kernel_VisibilityKind                   /* visibility*/
        [String]                                /* name */
        Kernel_Slot-set                         /* slot */
        [Kernel_ValueSpecification]             /* specification */
        Kernel_Classifier-set                   /* classifier */

/* 1.13 Kernel - Package */

Kernel_Package ::=(::)
        Kernel_PackageableElement-set           /* ownedMember */
        Kernel_PackageMerge-set                 /* packageExtension */
        Kernel_Package-set                      /* nestedPackage */
        Kernel_VisibilityKind                   /* visibility */
        [String]                                /* name */
        Kernel_ElementImport-set                /* elementImport */
        Kernel_PackageImport-set                /* packageImport */
        Kernel_Constraint-set                   /* ownedRule */

Kernel_PackageMerge ::=(::)
        Kernel_Package_Identifier               /* mergedPackage */
```

B Agent Moves for Ping-Pong System

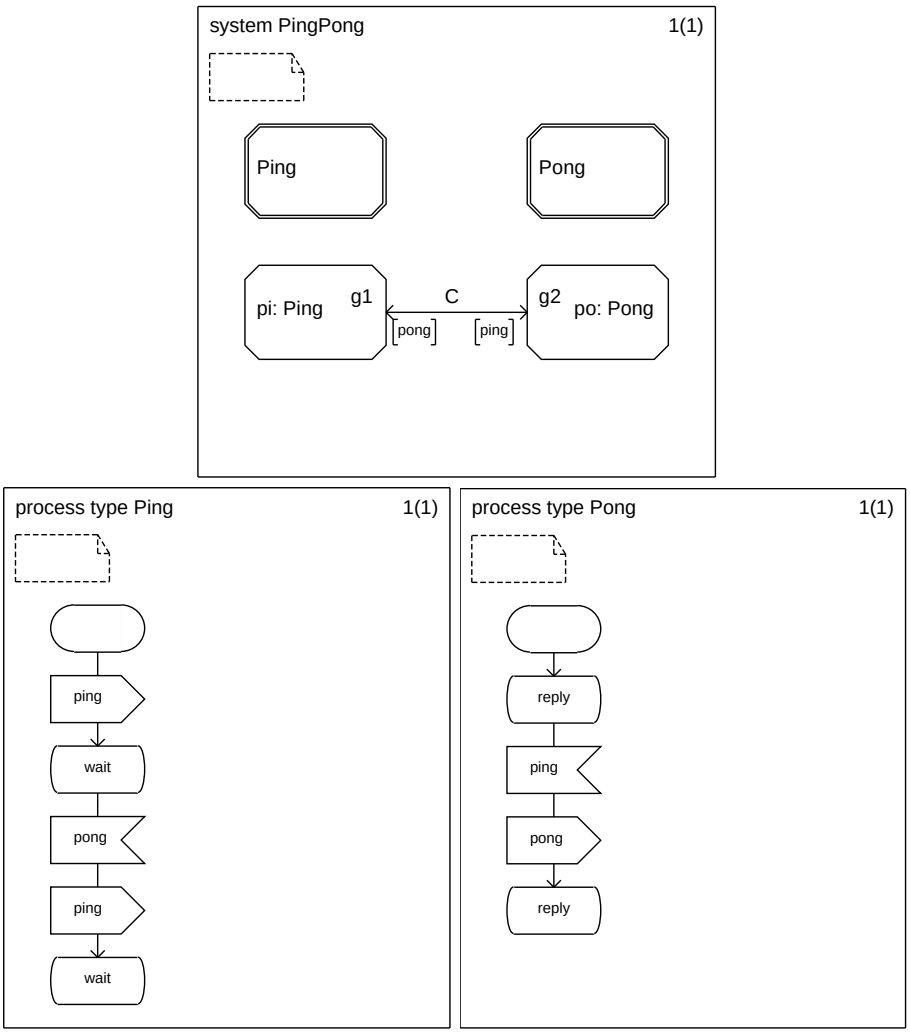


Figure B.1: SDL specification of ping-pong system

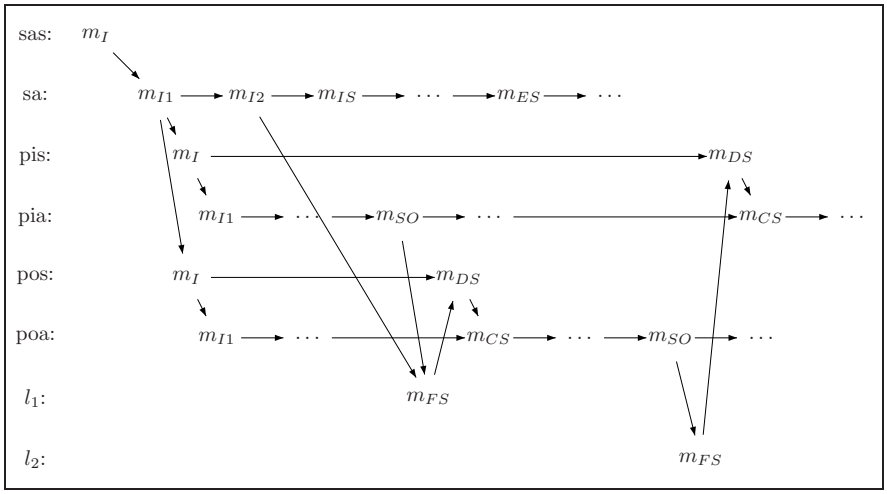


Figure B.2: Partially-ordered run for ping-pong system

Move	**Agent mode (level)**	**Action**
m_I	*initialisation* (1)	initialise agent set
m_{I1}	*initialising1* (2)	initialise agent
m_{I2}	*initialising2* (2)	create agent channels
m_{IS}	*initialisingStateMachine* (2)	initialise state machine
m_{ES}	*enteringStateNode* (3)	enter statenode
m_{DS}	*execution* (1)	deliver signal to inport
m_{SO}	*firingAction* (3)	signal output
m_{CS}	*selectionPhase* (4)	consume input signal
m_{FS}	- (-)	forward signal via link

Table B.1: Moves performed by ping-pong ASM agents

SDL	**SVM**
System PingPong	AgentSet sas, Agent sa
Agent Ping	AgentSet pis, Agent pia
Agent Pong	AgentSet pos, Agent poa
Channel C	Link l_1, Link l_2

Table B.2: SDL entities and corresponding ASM agents

C Complete Formal Definition of Extraction

C.1 Definitions

$D, D_1, D_2 \in$ domain, dn $\in$ `DomainName`, exp $\in$ formula, $R, R_1, R_2 \in$ rule, ps $\in$ paramSeq

$remove(\text{mode } \textbf{domain } \text{dn}, \mathcal{V}) = \text{mode } \textbf{domain } \text{dn}$
$remove(\textbf{domain } \text{dn}, \mathcal{V}) = \textbf{domain } \text{dn}$
$remove(\text{dn} =_{def} D, \mathcal{V}) = \text{dn} =_{def} remove(D, \mathcal{V})$

$remove(\text{mode } f \text{ ':' } D_1 \to D_2, \mathcal{V}) = \text{mode } f \text{ ':' } remove(D_1, \mathcal{V}) \to remove(D_2, \mathcal{V})$
$remove(\text{mode } f \text{ ':' } \to D_2, \mathcal{V}) = \text{mode } f \text{ ':' } \to remove(D_2, \mathcal{V})$
$remove(f \text{ ':' } D_1 \to D_2, \mathcal{V}) = f \text{ ':' } remove(D_1, \mathcal{V}) \to remove(D_2, \mathcal{V})$
$remove(f \text{ ':' } \to D_2, \mathcal{V}) = f \text{ ':' } \to remove(D_2, \mathcal{V})$

$remove(f \text{ ':' } D =_{\text{def}} \text{exp}, \mathcal{V}) = f \text{ ':' } remove(D, \mathcal{V}) =_{\text{def}} remove(\text{exp}, \mathcal{V})$
$remove(f(\text{ps}) \text{ ':' } D =_{\text{def}} \text{exp}, \mathcal{V}) =$
$\quad f(remove(\text{ps}, \mathcal{V})) \text{ ':' } remove(D, \mathcal{V}) =_{\text{def}} remove(\text{exp}, \mathcal{V} \cup remfpar(\text{ps}))$

$remove(\texttt{RuleName} \equiv R, \mathcal{V}) = \texttt{RuleName} \equiv remove(R, \mathcal{V})$
$remove(\texttt{RuleName}(\text{ps}) \equiv R, \mathcal{V}) =$
$\quad \texttt{RuleName}(remove(\text{ps}, \mathcal{V})) \equiv remove(R, \mathcal{V} \cup remfpar(\text{ps}))$

$remove(\textbf{constraint } \text{exp}, \mathcal{V}) = \textbf{constraint } remove(\text{exp}, \mathcal{V})$
$remove(\textbf{initially } \text{exp}, \mathcal{V}) = \textbf{initially } remove(\text{exp}, \mathcal{V})$

$remove(\texttt{ProgramName} \text{ ':' } R, \mathcal{V}) = \texttt{ProgramName} \text{ ':' } remove(R, \mathcal{V})$
$remove(\texttt{ProgramName} \text{ ':'}, \mathcal{V}) = \texttt{ProgramName} \text{ ':'}$

C.2 Macros, Functions and Parameters

fcs $\in$ `formulaCommaSeq`, $n, p \in \mathbb{N}$

$remove(\text{ps} \text{ ',' } x \text{ ':' } D, \mathcal{V}) =$
$\quad remove(\text{ps}, \mathcal{V})$ iff $undefined(D)$

$remove(\text{ps}, \mathcal{V})$ ',' x ':' $remove(D, \mathcal{V})$ else

$remove(x$ ':' $D, \mathcal{V}) =$
- " iff $undefined(D)$
- x ':' $remove(D, \mathcal{V})$ else

$numfpar(\text{fcs}$ ',' $\text{exp}) = numfpar(\text{fcs}) + 1$
$numfpar(\text{exp}) = 1$
$numfpar(\text{ps}$ ',' x ':' $D) = numfpar(\text{ps}) + 1$
$numfpar(x$ ':' $D) = 1$

$remfpar(\text{ps}$ ',' x ':' $D) =$
- $\{x\} \cup remfpar(\text{ps})$ iff $undefined(D)$
- $remfpar(\text{ps})$ else

$remfpar(x$ ':' $D) =$
- $\{x\}$ iff $undefined(D)$
- $\{\}$ else

$code(\text{ps}$ ',' x ':' $D, n) =$
- $code(\text{ps}, n-1) + 2^n$ iff $undefined(D)$
- $code(\text{ps}, n-1)$ else

$code(x$ ':' $D, n) =$
- 2^n iff $undefined(D)$
- 0 else

$count(\texttt{MacroName}) = code(\text{ps}, numfpar(\text{ps}) - 1)$

$removepar(\text{fcs}$ ',' $\text{exp}, n, p) =$
- $removepar(\text{fcs}, n-1, p-2^n)$ iff $p - 2^n > 0$
- $removepar(\text{fcs}, n-1, p)$ ',' exp else

$removepar(\text{exp}, n, p) =$
- " iff $p = 2^n$ // n should be 0
- exp else

$$undefined(\text{fcs}, \text{exp}, \mathcal{V}) \quad \text{iff} \quad undefined(\text{fcs}, \mathcal{V}) \vee undefined(\text{exp}, \mathcal{V})$$

C.3 Rules

$remove(f(\text{fcs}) := \text{exp}, \mathcal{V}) =$

skip iff $undefined(f, \mathcal{V}) \vee undefined(\text{fcs}, \mathcal{V}) \vee undefined(\text{exp}, \mathcal{V})$
$f(remove(\text{fcs}, \mathcal{V})) := remove(\text{exp}, \mathcal{V})$ else

$remove(\textbf{if } \text{exp } \textbf{then } R_1 \textbf{ else } R_2 \textbf{ endif}, \mathcal{V}) =$
$remove(R_1, \mathcal{V})$ iff $true(\text{exp}, \mathcal{V})$
$remove(R_2, \mathcal{V})$ iff $false(\text{exp}, \mathcal{V})$
skip iff $undefined(\text{exp}, \mathcal{V})$
if $remove(\text{exp}, \mathcal{V})$ **then** $remove(R_1, \mathcal{V})$ else
else $remove(R_2, \mathcal{V})$ **endif**

$remove(\textbf{let } x = \text{exp } \textbf{in } R \textbf{ endlet}, \mathcal{V}) =$
$remove(R, \mathcal{V} \cup \{x\})$ iff $undefined(\text{exp}, \mathcal{V})$
let $x = remove(\text{exp}, \mathcal{V})$ **in** $remove(R, \mathcal{V})$ else
endlet

$remove(\textbf{let } x : D = \text{exp } \textbf{in } R \textbf{ endlet}, \mathcal{V}) =$
$remove(R, \mathcal{V} \cup \{x\})$ iff $undefined(\text{exp}, \mathcal{V}) \vee undefined(D)$
let $x : remove(D, \mathcal{V}) = remove(exp, \mathcal{V})$ **in** else
$remove(R, \mathcal{V})$ **endlet**

$remove(\textbf{extend } \text{dn } \textbf{with } x\ R \textbf{ endextend}, \mathcal{V}) =$
$remove(R, \mathcal{V} \cup \{x\})$ iff $undefined(\text{dn}, \mathcal{V})$
extend dn **with** x $remove(R, \mathcal{V})$ **endextend** else

$remove(\textbf{choose } x : \text{exp } R \textbf{ endchoose}, \mathcal{V}) =$
skip iff $false(\text{exp}, \mathcal{V}) \vee true(\text{exp}, \mathcal{V}) \vee undefined(\text{exp}, \mathcal{V})$
choose x : $remove(\text{exp}, \mathcal{V})$ $remove(R, \mathcal{V})$ else
endchoose

$remove(\textbf{do forall } x \text{ ':' } \text{exp } R \textbf{ enddo}, \mathcal{V}) =$
skip iff $false(\text{exp}, \mathcal{V}) \vee true(\text{exp}, \mathcal{V}) \vee undefined(\text{exp}, \mathcal{V})$
do forall x ':' $remove(\text{exp}, \mathcal{V})$ $remove(R, \mathcal{V})$ else
enddo

$remove(R_1\ R, \mathcal{V}) = remove(R_1, \mathcal{V})\ remove(R, \mathcal{V})$

C.4 Domains

$s, s_1, s_2 \in$ simpledomain, $t \in$ tupledomain, $u \in$ uniondomain, ics $\in$ itemCommaSeq.

$remove(\text{dn}, \mathcal{V}) =$
nodomain iff $undefined(dn, \mathcal{V})$

dn else

$remove(dn\ mod, \mathcal{V}) =$

nodomain iff $undefined(dn)$

$dn\ mod$ else

$remove((D), \mathcal{V}) =$

nodomain iff $undefined(D, \mathcal{V})$

$(remove(D, \mathcal{V}))$ else

$remove((D)\textbf{-set}, \mathcal{V}) =$

nodomain iff $undefined(D, \mathcal{V})$

$(remove(D, \mathcal{V}))\textbf{-set}$ else

$remove([D], \mathcal{V}) =$

nodomain iff $undefined(D, \mathcal{V})$

$[remove(D, \mathcal{V})]$ else

$remove([D]\ mod, \mathcal{V}) =$

nodomain iff $undefined(D, \mathcal{V})$

$[remove(D, \mathcal{V})]\ mod$ else

$remove(D_1 \to D_2, \mathcal{V}) =$

nodomain iff $undefined(D_1, \mathcal{V}) \vee undefined(D_2, \mathcal{V})$

$remove(D_1, \mathcal{V}) \to remove(D_2, \mathcal{V})$ else

$remove(s_1 \times s_2, \mathcal{V}) =$

" iff $undefined(s_1 \times s_2, \mathcal{V})$

$remove(s_1, \mathcal{V}) \times remove(s_2, \mathcal{V})$ else

$remove(t \times s, \mathcal{V}) =$

" iff $undefined(t \times s, \mathcal{V})$

$remove(t, \mathcal{V}) \times remove(s, \mathcal{V})$ else

$remove('(' ')', \mathcal{V}) = '(' ')'$

$remove(s_1 \cup s_2, \mathcal{V}) =$

" iff $undefined(s_1 \cup s_2, \mathcal{V})$

$remove(s_1, \mathcal{V})$ iff $undefined(s_2, \mathcal{V})$

$remove(s_2, \mathcal{V})$ iff $undefined(s_1, \mathcal{V})$

$remove(s_1, \mathcal{V}) \cup remove(s_2, \mathcal{V})$ else

$remove(u \cup s, \mathcal{V}) =$

" iff $undefined(u \cup s, \mathcal{V})$

$remove(u, \mathcal{V})$ iff $undefined(s, \mathcal{V})$

$remove(s, \mathcal{V})$ iff $undefined(u, \mathcal{V})$

$remove(u, \mathcal{V}) \cup remove(s, \mathcal{V})$ else

$remove('\{' \text{ ics } '\}', \mathcal{V}) =$ **if** $undefined(\text{ics}, \mathcal{V})$ **then** " **else** '{' $remove(\text{ics}, \mathcal{V})$ '}' **endif**

$$
\begin{array}{rcl}
\mathit{undefined}(\mathrm{dn}\underline{*}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\mathrm{dn}\underline{**}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\mathrm{dn}\underline{+}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\mathrm{dn}\texttt{-set}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\underline{(}D\underline{)}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{(}D\underline{)}\texttt{-set}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}\underline{*}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}\underline{+}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(D_1 \rightarrow D_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(D_1, \mathcal{V}) \vee \mathit{undefined}(D_2, \mathcal{V})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{undefined}(s_1 \times s_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(s_1, \mathcal{V}) \vee \mathit{undefined}(s_2, \mathcal{V}) \\
\mathit{undefined}(t \times s, \mathcal{V}) & \text{iff} & \mathit{undefined}(t, \mathcal{V}) \vee \mathit{undefined}(s, \mathcal{V}) \\
\mathit{undefined}(\text{`(` `)`}, \mathcal{V}) & \text{iff} & \text{false} \\
 & & \\
\mathit{undefined}(s_1 \cup s_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(s_1, \mathcal{V}) \wedge \mathit{undefined}(s_2, \mathcal{V}) \\
\mathit{undefined}(u \cup s, \mathcal{V}) & \text{iff} & \mathit{undefined}(u, \mathcal{V}) \wedge \mathit{undefined}(s, \mathcal{V}) \\
 & & \\
\mathit{undefined}('\{'\mathrm{ics}'\}', \mathcal{V}) & \text{iff} & \mathit{undefined}(\mathrm{ics}, \mathcal{V}) \\
\mathit{undefined}(\mathrm{ics}',' x, \mathcal{V}) & \text{iff} & x \in \mathcal{V} \\
\mathit{undefined}(\mathrm{ics}',' \mathrm{kw}, \mathcal{V}) & \text{iff} & \mathit{undefined}(\mathrm{ics}, \mathcal{V}) \wedge \mathit{undefined}(\mathrm{kw}) \\
\mathit{undefined}(\mathrm{ics}',' \mathrm{kw1\ kw2}, \mathcal{V}) & \text{iff} & \mathit{undefined}(\mathrm{ics}, \mathcal{V}) \wedge \mathit{undefined}(\mathrm{kw1}) \wedge \\
 & & \mathit{undefined}(\mathrm{kw2}) \\
\mathit{undefined}(\mathrm{ics}',' \texttt{Literal}, \mathcal{V}) & \text{iff} & \text{false}
\end{array}
$$

C.5 Expressions

Boolean Operators

e_1	¬	T	F	U	-
		F	T	U	$\neg e_1$

e_1	e_2 $\vee$	T	F	U	-
	T	T	T	T	T
	F	T	F	F	e_1
	U	T	F	U	e_1
	-	T	e_2	e_2	$e_1 \vee e_2$

e_1	e_2 $\wedge$	T	F	U	-
	T	T	F	T	e_1
	F	F	F	F	F
	U	T	F	U	e_1
	-	e_2	F	e_2	$e_1 \wedge e_2$

e_1	e_2 $\rightarrow$	T	F	U	-
	T	T	T	T	T
	F	F	T	F	$\neg e_1$
	U	F	T	U	$\neg e_1$
	-	e_2	T	$\neg e_1$	$e_1 \rightarrow e_2$

e_1	e_2 $\leftrightarrow$	T	F	U	-
	T	T	F	F	e_1
	F	F	T	F	$\neg e_1$
	U	F	F	U	F
	-	e_2	$\neg e_2$	F	$e_1 \leftrightarrow e_2$

$$
\begin{aligned}
\mathit{true}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) \quad \text{iff} \quad & (\mathit{true}(e) \wedge \mathit{true}(e_1)) \vee \\
& (\mathit{false}(e) \wedge \mathit{true}(e_2)) \\
\mathit{false}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) \quad \text{iff} \quad & (\mathit{true}(e) \wedge \mathit{false}(e_1)) \vee \\
& (\mathit{false}(e) \wedge \mathit{false}(e_2)) \\
\mathit{undefined}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) \quad \text{iff} \quad & (\mathit{true}(e) \wedge \mathit{undefined}(e_1)) \vee \mathit{undefined}(e) \\
& \vee(\mathit{false}(e) \wedge \mathit{undefined}(e_2)) \vee \\
& (\mathit{undefined}(e_1) \wedge \mathit{undefined}(e_2))
\end{aligned}
$$

Quantification

$$Qx \in e_1 : e_2,\ Q \in \{\forall, \exists, \exists_1\}$$

e_2	e_1 $\forall$	T	F	U	-
	U	T	T	T	T
	-	T	-	U	-

$\mathit{remove}(\forall \text{nseq} \in e_1\text{`} : \text{`}e_2, \mathcal{V}) =$

`true` iff $undefined(e_1, \mathcal{V}) \vee true(e_2, \mathcal{V})$
`undefined` iff $\neg undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V})$
$\forall \text{nseq} \in remove(e_1, \mathcal{V})\text{`} : \text{`}remove(e_2, \mathcal{V})$ else

e_2	e_1 $\exists$	T	F	U	-
	U	F	F	F	F
	-	-	F	U	-

$remove(\exists \text{nseq} \in e_1\text{`} : \text{`}e_2, \mathcal{V}) =$
`false` iff $undefined(e_1, \mathcal{V}) \vee false(e_2, \mathcal{V})$
`undefined` iff $\neg undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V})$
$\forall \text{nseq} \in remove(e_1, \mathcal{V})\text{`} : \text{`}remove(e_2, \mathcal{V})$ else

e_2	e_1 $\exists_1$	T	F	U	-
	U	F	F	F	F
	-	-	F	U	-

$remove(\exists_1 \text{nseq} \in e_1\text{`} : \text{`}e_2, \mathcal{V}) =$
`false` iff $undefined(e_1, \mathcal{V}) \vee false(e_2, \mathcal{V})$
`undefined` iff $\neg undefined(e_1, \mathcal{V}) \wedge undefined(e_2, \mathcal{V})$
$\forall \text{nseq} \in remove(e_1, \mathcal{V})\text{`} : \text{`}remove(e_2, \mathcal{V})$ else

Relational Operators

e_1	e_2 op	op $\in \{<, >, \leq, \geq\}$ U	-
	U	U	U
	-	U	-

$remove(e_1 > e_2, \mathcal{V}) =$
`undefined` iff $undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V}) > remove(e_2, \mathcal{V})$ else
$remove(e_1 < e_2, \mathcal{V}) =$
`undefined` iff $undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V}) < remove(e_2, \mathcal{V})$ else
$remove(e_1 \geq e_2, \mathcal{V}) =$
`undefined` iff $undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V}) \geq remove(e_2, \mathcal{V})$ else
$remove(e_1 \leq e_2, \mathcal{V}) =$
`undefined` iff $undefined(e_1, \mathcal{V}) \vee undefined(e_2, \mathcal{V})$
$remove(e_1, \mathcal{V}) \leq remove(e_2, \mathcal{V})$ else

e_1	e_2 $\in$	U	-
	U	F	F
	-	F	-

e_1	e_2 $\notin$	U	-
	U	T	T
	-	T	-

$$
\begin{array}{rll}
\mathit{true}(e = \texttt{undefined}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \texttt{undefined}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{true}(e = \emptyset, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \emptyset, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{true}(e = \mathit{empty}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e \neq \mathit{empty}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{false}(e_1 = e_2, \mathcal{V}) & \text{iff} & \neg \mathit{true}(e_1 = e_2, \mathcal{V}) \wedge \\
 & & (\mathit{undefined}(e_1, \mathcal{V}) \wedge \neg \mathit{undefined}(e_2, \mathcal{V}) \vee \\
 & & \neg \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V})) \\
\mathit{true}(e_1 \neq e_2, \mathcal{V}) & \text{iff} & \neg \mathit{false}(e_1 \neq e_2, \mathcal{V}) \wedge \\
 & & (\mathit{undefined}(e_1, \mathcal{V}) \wedge \neg \mathit{undefined}(e_2, \mathcal{V}) \vee \\
 & & \neg \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}))
\end{array}
$$

Arithmetic Operators

$$
\begin{array}{lll}
\mathit{remove}(-e, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\quad -\mathit{remove}(e, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 + e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) + \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 - e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) - \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 * e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) * \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \,/\, e_2, \mathcal{V}) = & &
\end{array}
$$

$$
\begin{array}{lll}
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) \mathbin{/} \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \text{ MOD } e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) \text{ MOD } \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \text{ DIV } e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) \text{ DIV } \mathit{remove}(e_2, \mathcal{V}) & & \text{else}
\end{array}
$$

$$
\begin{array}{rll}
\mathit{undefined}(-e, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{undefined}(e_1 + e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 - e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 * e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \mathbin{/} e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \textbf{ mod } e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \textbf{ rem } e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V})
\end{array}
$$

Sets and Sequences

$$< e_1 \mid x \in e_2 : e_3 >, \quad \{e_1 \mid x \in e_2 : e_3\}$$

$$
\begin{array}{lll}
\mathit{remove}(< e_1 \mid x \in e_2 : e_3 >, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_3, \mathcal{V}) \\
\quad \mathit{empty} & \text{iff} & (\mathit{false}(e_3, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V})) \\
 & & \wedge \neg \mathit{undefined}(e_1, \mathcal{V}) \\
\quad < \mathit{remove}(e_1, \mathcal{V}) \mid x \in \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\quad : \mathit{remove}(e_3, \mathcal{V}) > & & \\
\mathit{remove}(\{e_1 \mid x \in e_2 : e_3\}, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_3, \mathcal{V}) \\
\quad \emptyset & \text{iff} & (\mathit{false}(e_3, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V})) \\
 & & \wedge \neg \mathit{undefined}(e_1, \mathcal{V}) \\
\quad \{\mathit{remove}(e_1, \mathcal{V}) \mid x \in \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\quad : \mathit{remove}(e_3, \mathcal{V})\} & &
\end{array}
$$

$$
\begin{array}{rll}
\mathit{undefined}(< e_1 \mid x \in e_2 : e_3 >, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_3, \mathcal{V}) \\
\mathit{undefined}(\{e_1 \mid x \in e_2 : e_3\}, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_3, \mathcal{V})
\end{array}
$$

		op ∈ {| |, ⋃}
e_1	op	⋃ -
		⋃ -

	e_2	op ∈ {∩, ∪, ∩}
e_1	op	⋃ -
	⋃	⋃ -
	-	- -

	e_2	op ∈ {↦, ..}
e_1	op	⋃ -
	⋃	⋃ ⋃
	-	⋃ -

Function and Macro Calls

$remove(\texttt{MkName}(), \mathcal{V}) =$

`undefined`	iff	$undefined(\texttt{MkName})$
$\texttt{MkName}()$		else

$remove(\texttt{MkName}(\text{fcs}), \mathcal{V}) =$

skip	iff	$undefined(\texttt{MkName}) \vee undefined(\text{fcs}, \mathcal{V})$
$\texttt{MkName}(removepar(\text{fcs}, numfpar(fcs) - 1, count(\texttt{MkName})))$		else

$$undefined(\texttt{MkName}(), \mathcal{V}) \quad \text{iff} \quad undefined(\texttt{MkName})$$
$$undefined(\texttt{MkName}(\text{fcs}), \mathcal{V}) \quad \text{iff} \quad undefined(\texttt{MkName}) \vee undefined(\text{fcs}, \mathcal{V})$$

$remove(f, \mathcal{V}) =$

`undefined`	iff	$undefined(f)$
f		else

$remove(exp.f, \mathcal{V}) =$

`undefined`	iff	$undefined(f) \vee undefined(exp, \mathcal{V})$
$remove(exp, \mathcal{V}).f$		else

$remove(f(\text{fcs}), \mathcal{V}) =$

`undefined`	iff	$undefined(f) \vee undefined(\text{fcs}, \mathcal{V})$
$f(removepar(\text{fcs}, numfpar(fcs) - 1, count(f)))$		else

$$undefined(f(\text{fcs}), \mathcal{V}) \quad \text{iff} \quad undefined(f) \vee undefined(\text{fcs}, \mathcal{V})$$
$$undefined(f, \mathcal{V}) \quad \text{iff} \quad undefined(f)$$
$$undefined(exp.f, \mathcal{V}) \quad \text{iff} \quad undefined(f) \vee undefined(exp, \mathcal{V})$$

$remove(\texttt{DomainName}, \mathcal{V}) = \textbf{if } undefined(\texttt{DomainName}) \textbf{ then } \texttt{undefined} \textbf{ else } \texttt{DomainName} \textbf{ endif}$
$remove(\texttt{SynName}, \mathcal{V}) = \textbf{if } undefined(\texttt{SynName}) \textbf{ then } \texttt{undefined} \textbf{ else } \texttt{SynName} \textbf{ endif}$
$remove(\texttt{ProgramName}, \mathcal{V}) = \textbf{if } undefined(\texttt{ProgramName}) \textbf{ then } \texttt{undefined} \textbf{ else } \texttt{ProgramName} \textbf{ endif}$
$remove(\texttt{Keyword}, \mathcal{V}) = \textbf{if } undefined(\texttt{Keyword}) \textbf{ then } \texttt{undefined} \textbf{ else } \texttt{Keyword} \textbf{ endif}$
$remove(\texttt{Literal}, \mathcal{V}) = \texttt{Literal}$
$remove(\texttt{undefined}, \mathcal{V}) = \texttt{undefined}$
$remove(\underline{\leq}\text{pcs}\underline{\geq}, \mathcal{V}) = \underline{\leq} remove(\text{pcs}, \mathcal{V}) \underline{\geq}$

$$
\begin{aligned}
undefined(\texttt{DomainName}, \mathcal{V}) &\text{ iff } undefined(\texttt{DomainName}) \\
undefined(\texttt{SynName}, \mathcal{V}) &\text{ iff } undefined(\texttt{SynName}) \\
undefined(\texttt{ProgramName}, \mathcal{V}) &\text{ iff } undefined(\texttt{ProgramName}) \\
undefined(\texttt{Keyword}, \mathcal{V}) &\text{ iff } undefined(\texttt{Keyword}) \\
undefined(\texttt{Literal}, \mathcal{V}) &\text{ iff } \texttt{false} \\
undefined(\texttt{undefined}, \mathcal{V}) &\text{ iff } \texttt{false} \\
undefined(\underline{\leq}\text{pcs}\underline{\geq}, \mathcal{V}) &\text{ iff } undefined(\text{pcs}, \mathcal{V})
\end{aligned}
$$

D Reduction Profiles for SDL Features

Reduction Profile for Procedure

d−Procedure−definition
d−Procedure−graph
d−Procedure−start−node
d−Value−return−node
d−Value−returning−call−node
d−Entry−procedure−definition
d−Exit−procedure−definition
d−Call
d−Call−node
f−procedureAS1
f−callingProcedureNode
a−initialisingProcedure
a−procedureNode

r−CreateProcedureVariables
r−EvalExitProcedure

True:
f−functional

Reduction Profile for Exception

d−Exception
d−ExceptionInst
d−ExceptionScope
d−ExceptionHandlerName
d−ExceptionHandlerNode
d−SetHandler
d−Raise

d−Exception−name
d−Exception−handler−name
d−Exception−identifier
d−Exception−definition
d−Exception−handler−node
d−On−exception
d−Handle−node
d−Raise−node

a−selectException

Reduction Profile for Continuous Signal

a−selectContinuous/*a−startSelection*

Reduction Profile for Inheritance

f−inheritedStateNode
f−stateNodesToBeSpecialised

Reduction Profile for Exit Transition

d−StateExitPoint
f−currentExitStateNodes
f−stateNodeToBeExited
a−selectExitTransition
a−exitingCompositeState

Reduction Profile for Priority Input

f−priorityInputTransitions
a−selectPriorityInput/*a−selectInput*

Reduction Profile for Timer

d−Set−node
d−Reset−node
d−Timer−active−expression

d−Timer
d−TimerInst
d−TimerActive
d−TimeLabel
d−Set
d−Reset

Reduction Profile for Spontaneous Transition

a−selectSpontaneous

False:
f−Spontaneous

Reduction Profile for Entry-/Exit-Procedure

d−Entry−procedure−definition
d−Exit−procedure−definition

Reduction Profile for Save

d−Save−signalset

False:
f−SignalSaved

E Syntax of Abstract State Machines

Concrete Syntax of ASMs

The concrete syntax of ASMs, as defined by the SDLC compiler [58].

```
/*************** general structure ***************/
spec:       asmSpec
    ;

asmSpec: /* empty */ %prec DO_SHIFT
    | asmSpec asmItem
    ;

asmItem: domainDef
    | functionDecl
    | functionDef
    | ruleDef
    | programDef
    | constraint
    | initialCond
    ;

/*************** definitions ***************/
domainDef:
      mode DOMAIN DOMAINNAME
    | DOMAIN DOMAINNAME
    | DEF_START DOMAINNAME DEFINES domain
    ;

functionDecl:
      DEF_START mode FUNCTIONNAME ':' domain ARROW domain
    | DEF_START mode FUNCTIONNAME ':' ARROW domain
    | DEF_START FUNCTIONNAME ':' domain ARROW domain
    | DEF_START FUNCTIONNAME ':' ARROW domain
    ;

functionDef:
      DEF_START FUNCTIONNAME ':' domain DEFINES formula_with_let
    | DEF_START FUNCTIONNAME ':' domain DEFINES DEF_START formula
    | DEF_START FUNCTIONNAME formalParams ':' domain DEFINES formula_with_let
    | DEF_START FUNCTIONNAME formalParams ':' domain DEFINES DEF_START formula
    ;

ruleDef:
      DEF_START RULENAME EQUIV rules wherePart
    | DEF_START RULENAME formalParams EQUIV rules wherePart
```

```
    ;

constraint: CONSTRAINT formula
    ;

initialCond: INITIALLY formula
    ;

programDef:
      PROGRAMNAME ':' rules wherePart
    | PROGRAMNAME ':'
    ;

/**************** auxiliar for definitions ****************/
mode:    STATIC
    | SHARED
    | MONITORED
    | CONTROLLED
    | DERIVED
    ;

formalParams:
      '(' ')'
    | '(' paramSeq ')'
    ;

paramSeq:
      ASMNAME ':' domain
    | paramSeq ',' ASMNAME ':' domain
    ;

wherePart:
      /* empty */
    | WHERE abbreviations ENDWHERE
    ;

abbreviations:
      functionDef
    | ruleDef
    | abbreviations functionDef
    | abbreviations ruleDef
    ;

/**************** expressions ****************/
formula_with_let: formula
    | letPart formula ENDLET
    ;

formula:
      primary
    | formula '=' formula
    | formula NEQ formula
    | formula AND formula
    | formula OR formula
    | formula IMPLIES formula
    | formula IFF formula
```

```
    | NOT formula %prec UMINUS
    | IF formula THEN formula elsepart
    | IF formula THEN letPart formula elsepart
    | IF formula THEN letPart formula ENDLET elsepart
    | formula '>' formula
    | formula '<' formula
    | formula GEQ formula
    | formula LEQ formula
    | '-' formula %prec UMINUS
    | formula '+' formula
    | formula '-' formula
    | formula '*' formula
    | formula '/' formula
    | formula MOD formula
    | formula REM formula
    | '<' primary '|' ASMNAME IN primary ':' formula '>'
    | '<' primary '|' ASMNAME IN primary '>'
    | '<' ASMNAME IN primary ':' formula '>'
    | ASMNAME IN ASMNAME
    | formula CONCAT formula
    | formula UNION formula
    | formula INTERSECT formula
    | formula SETMINUS formula
    | formula ELEMENTOF formula
    | formula NOTIN formula
    | formula SUBSETEQ formula
    | formula SUBSET formula
    | '|' formula '|'
    | BIGUNION formula %prec UMINUS
    | EMPTYSET
    | '{' primaryCommaSeq '}'
    | '{' ASMNAME ELEMENTOF formula ':' formula '}'
    | '{' formula '|' ASMNAME ELEMENTOF formula '}'
    | '{' formula '|' ASMNAME ELEMENTOF formula ':' formula '}'
    | '{' formula ARROW formula '}'
    | formula DOTDOT formula
    | quantified
    ;

elsepart : ELSE formula
    | ELSE formula ENDIF
    | ELSE letPart formula ENDIF
    | ELSE letPart formula ENDLET ENDIF
    | ELSEIF formula THEN formula_with_let elsepart
    | ENDIF
    ;

primary: functionCall
    | MKNAME '(' formulaCommaSeq ')'
    | MKNAME '(' ')'
    | '(' formula_with_let ')'
    | DOMAINNAME
    | SYNNAME
    | PROGRAMNAME
    | KEYWORD
    | KEYWORD KEYWORD
```

```
    | LITERAL
    | UNDEFINED
    | '<' primaryCommaSeq '>'
    ;

primaryCommaSeq:
      primary
    | primaryCommaSeq ',' primary
    ;

formulaCommaSeq:
      formula
    | formulaCommaSeq ',' formula
    ;

quantified:
      FORALL nameCommaSeq ELEMENTOF formula ':' formula
    | EXISTS nameCommaSeq ELEMENTOF formula ':' formula
    | EXISTS '!' nameCommaSeq ELEMENTOF formula ':' formula
    | FORALL nameCommaSeq ELEMENTOF formula ':' letPart formula ENDLET
    | EXISTS nameCommaSeq ELEMENTOF formula ':' letPart formula ENDLET
    | EXISTS '!' nameCommaSeq ELEMENTOF formula ':' letPart formula ENDLET
    ;

functionCall:
      FUNCTIONNAME
    | FUNCTIONNAME '(' formulaCommaSeq ')'
    | SYNNAME '(' formulaCommaSeq ')'
    | ASMNAME
    | ASMNAME '(' formula ')'
    | SNAME '(' formula ')'
    | S2NAME '(' formula ')'
    | S3NAME '(' formula ')'
    | functionCall '.' FUNCTIONNAME
    | functionCall '.' SNAME
    | functionCall '.' S2NAME
    | functionCall '.' S3NAME
    | functionCall '[' formula ']'
    ;

/*************** rules ***************/
rule:     FUNCTIONNAME ASSIGN formula
    | ASMNAME ASSIGN formula
    | FUNCTIONNAME '(' formulaCommaSeq ')' ASSIGN formula
    | functionCall '.' FUNCTIONNAME ASSIGN formula
    | functionCall '.' DOMAINNAME ASSIGN formula
    | RULENAME
    | RULENAME '(' formulaCommaSeq ')'
    | IF formula THEN rules elserules
    | IF formula THEN elserules
    | DOFORALL ASMNAME ':' formula rules %prec DOFORALL
    | DOFORALL ASMNAME ':' formula rules ENDDO
    | CHOOSE ASMNAME ':' formula rules %prec CHOOSE
    | CHOOSE ASMNAME ':' formula rules ENDCHOOSE
    | EXTEND DOMAINNAME WITH nameCommaSeq rules %prec EXTEND
    | EXTEND DOMAINNAME WITH nameCommaSeq rules ENDEXTEND
```

```
    | DOINPARALLEL rules ENDDO
    | oneLet rules %prec LET
    | oneLet rules ENDLET
    | SKIP
    ;

elserules : /* empty */ %prec THEN
    | ENDIF
    | ELSEIF formula THEN rules elserules
    | ELSE rules
    | ELSE rules ENDIF
    | ELSE ENDIF
    ;

rules:      rule
    | rules  rule
    ;

nameCommaSeq: ASMNAME
    | nameCommaSeq ',' ASMNAME
    ;

/*************** domains ***************/
domain:   simpledomain
    | tupledomain
    | uniondomain
    | '{' itemCommaSeq '}'
    | domain ARROW domain
    ;

simpledomain:   DOMAINNAME
    | DOMAINNAME '*'
    | DOMAINNAME '*' '*'
    | DOMAINNAME '+'
    | DOMAINNAME SET
    | SYNNAME
    | SYNNAME '*'
    | SYNNAME '+'
    | SYNNAME SET
    | KEYWORD
    | '(' domain ')'
    | '(' domain ')' SET
    | '[' domain ']'
    | '[' domain ']' '*'
    | '[' domain ']' '+'
    ;

tupledomain: simpledomain TIMES simpledomain
    | tupledomain TIMES simpledomain
    | '(' ')'
    ;

uniondomain: simpledomain UNION simpledomain
    | uniondomain UNION simpledomain
    ;
```

```
itemCommaSeq: ASMNAME
    | KEYWORD
    | KEYWORD KEYWORD
    | LITERAL
    | itemCommaSeq ',' ASMNAME
    | itemCommaSeq ',' KEYWORD
    | itemCommaSeq ',' KEYWORD KEYWORD
    | itemCommaSeq ',' LITERAL
    ;

/*************** transformations ***************/

letPart :  oneLet
    | letPart  oneLet
    ;

oneLet:   LET ASMNAME '=' formula IN
    | LET ASMNAME ':' domain '=' formula IN
    ;
```

Abstract Syntax of ASMs

The abstract syntax of ASMs, defined by the SDLC compiler [58] and modified for the SDL-profile tool.

```
asmSpec: Defs(defList);

defList :  list  definition ;

definition :
    DomainDecl(mode casestring)
  | DomainDef(casestring domain)
  | FunctionDecl(mode casestring domain domain)
  | FunctionDef(casestring fargList  domain expr)
  | RuleDef(casestring fargList  defList  rule)
  | ProgramDef(casestring defList rule)
  | Constraint(expr)
  | InitialCond(expr)
  { int refcounter = 0; }
  ;

letStatements:  list  letStatement;

letStatement: LetStatement(casestring domain expr);

pattern:  NamedPattern(casestring pattern)
  | ConstructorPattern(casestring parameters)
  | MatchAll()
  | KeywordP(casestring)
  | TwoKeywordP(casestring casestring)
  | LiteralP(casestring)
  | UndefP()
  | UnionP(nameList)
  ;
```

```
parameters: list pattern;

mode:   Static()
  | Controlled()
  | Shared()
  | Monitored()
  | Derived()
  ;

domain:   PlainDomain(casestring)
  | KWDomain(casestring)
  | SetDomain(domain)
  | SeqDomain(domain)
  | SeqPlusDomain(domain)
  | OptDomain(domain)
  | MapDomain(domain domain)
  | TupleDomain(tupleDomainList)
  | UnionDomain(unionDomainList)
  | ItemDomain(argumentList)
  | NoDomain()
  ;

tupleDomainList: list domain;
unionDomainList: list domain;

fargList: list farg;

farg: FArg(casestring domain);

rule:    Assign(casestring argumentList expr)
  | IfThenElse(expr rule rule)
  | Empty()
  | Skip()
  | Parallel(ruleList)
  | ForAll(casestring expr rule)
  | Choose(casestring expr rule)
  | Extend(domain nameList rule)
  | Let(letStatements rule)
  | RuleCall(casestring argumentList)
  | ProofObligation(casestring expr)
  { definition def; }
  ;

ruleList: list rule;

argumentList: list expr;

nameList: list casestring;

expr:    Variable(casestring)
  | Literal(casestring)
  | Program(casestring)
  | Domain(casestring)
  | IfExpr(expr expr expr)
  | FunCall(casestring argumentList)
  | Select(casestring casestring expr)
```

```
  | Index(expr expr)
  | MKCall(casestring argumentList)
  | BinOp(casestring expr expr)
  | UnOp(casestring expr)
  | Quant(qkind nameList expr expr)
  | SetComp(expr casestring expr expr)
  | SeqComp(expr casestring expr expr)
  | Range(expr expr)
  | MapElement(expr expr)
  | Keyword(casestring)
  | TwoKeyword(casestring casestring)
  | EmptySet()
  | Undef()
  | NoExpr()
  | LetExpr(letStatements expr)
  { definition def; }
  ;

qkind:    Exi()
  | ExiOne()
  | Gen()
  ;
```

Bibliography

[1] *The precise UML group.* – `http://www.cs.york.ac.uk/puml/`

[2] Bjørner, Dines (Hrsg.) ; Jones, Cliff B. (Hrsg.): *The Vienna Development Method: The Meta-Language.* Bd. *61.* Springer, 1978 (LNCS)

[3] Börger, Egon: The ASM Refinement Method. In: *Formal Aspects of Computing* 15 (2003), Nr. 2-3, S. 237–257

[4] Börger, Egon ; Cavarra, Alessandra ; Riccobene, Elvinia: An ASM Semantics for UML Activity Diagrams. In: Rus, Teodor (Hrsg.): *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology* Bd. 1816, Springer, 2000 (LNCS), S. 293–308

[5] Börger, Egon ; Cavarra, Alessandra ; Riccobene, Elvinia: Modeling the Dynamics of UML State Machines. In: Gurevich, Yuri (Hrsg.) ; Kutter, Peter W. (Hrsg.) ; Odersky, Martin (Hrsg.) ; Thiele, Lothar (Hrsg.): *Abstract State Machines. Theory and Applications* Bd. 1912, Springer, 2000 (LNCS), S. 223–241

[6] Börger, Egon ; Fruja, Nicu G. ; Gervasi, Vincenzo ; Stärk, Robert F.: A high-level modular definition of the semantics of C♯. In: *Theoretical Computer Science* 336 (2005), Mai, S. 235–284

[7] Börger, Egon ; Schulte, Wolfram: A Programmer Friendly Modular Definition of the Semantics of Java. In: Alves-Foss, Jim (Hrsg.): *Formal Syntax and Semantics of Java* Bd. 1523, Springer, 1999 (LNCS), S. 353–404

[8] Börger, Egon ; Stärk, Robert: *Abstract State Machines.* Springer, 2003

[9] Clark, Tony ; Evans, Andy ; Kent, Stuart: The Meta-modeling Language Calculus: Foundation Semantics for UML. In: Hussmann, Heinrich (Hrsg.): *Fundamental Approaches to Software Engineering : 4th International Conference, FASE 2001 : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001, Proceedings* Bd. 2029, Springer, 2001 (LNCS), S. 17–31

[10] Clark, Tony ; Evans, Andy ; Kent, Stuart: Engineering Modelling Languages: A Precise Meta-Modelling Approach. In: Kutsche, Ralf-Detlef (Hrsg.) ; Weber, Herbert (Hrsg.): *Fundamental Approaches to Software Engineering : 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002. Proceedings* Bd. 2306, Springer, 2002 (LNCS), S. 159–173

[11] Damm, Werner ; Josko, Bernhard ; Pnueli, Amir ; Votintseva, Angelika: Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In: de Boer, Frank S. (Hrsg.) ; Bonsangue, Marcello M. (Hrsg.) ; Graf, Susanne (Hrsg.) ; de Roever, Willem-Paul (Hrsg.): *First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures* Bd. 2852, Springer, 2003 (LNCS), S. 71–98

[12] Eijk, Peter van ; Belinfante, Axel ; Eertink, Henk ; Alblas, Henk: The Term Processor Generator Kimwitu / Centre for Telematics and Information Technology. University of Twente, Enschede, 1996 (CTIT TR 96-49). – Forschungsbericht

[13] Ellsberger, Jan ; Hogrefe, Dieter ; Sarma, Amardeo: *SDL - Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1997

[14] Eschbach, Robert: A Termination Detection Algorithm: Specification and Verification. In: Wing, Jeannette (Hrsg.) ; Jim Davies, Jim W. (Hrsg.): *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France.* Bd. 1709, Springer, 1999 (LNCS)

[15] Eschbach, Robert: *Formal Specification and Verification*, University of Kaiserslautern, Germany, Diss., 2005

[16] Eschbach, Robert ; Glässer, Uwe ; Gotzhein, Reinhard ; Löwis, Martin von ; Prinz, Andreas: Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. In: *Journal of Universal Computer Science, Special Issue on Abstract State Machines* 7 (2001), Nr. 11, S. 1024–1049

[17] Fischer, Joachim ; Piefel, Michael ; Scheidgen, Markus: A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In: Amyot, Daniel (Hrsg.) ; Williams, Alan W. (Hrsg.): *System Analysis and Modeling: 4th International SDL and MCS Workshop, SAM 2004, Ottawa, Canada* Bd. 3319, Springer, Januar 2005 (LNCS), S. 208–223

[18] Fliege, Ingmar ; Grammes, Rüdiger ; Weber, Christian: ConTraST - A Configurable SDL Transpiler And Runtime Environment. In: Gotzhein, Reinhard (Hrsg.) ; Reed, Rick (Hrsg.): *SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany* Bd. 4320, Springer, 2006 (LNCS), S. 222–234

[19] Glässer, Uwe ; Gotzhein, Reinhard ; Prinz, Andreas: The Formal Semantics of SDL-2000 - Status and Perspectives. In: *Computer Networks* 42 (2003), Nr. 3, S. 343–358

[20] Glässer, Uwe ; Gotzhein, Reinhard ; Prinz, Andreas: An Introduction To Abstract State Machines / Department of Computer Science, University of Kaiserslautern. 2003 (326/03). – Forschungsbericht

[21] Grammes, Rüdiger: Formal Operations for SDL Language Profiles. In: Gotzhein, Reinhard (Hrsg.) ; Reed, Rick (Hrsg.): *SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany* Bd. 4320, Springer, 2006 (LNCS), S. 51–65

[22] Grammes, Rüdiger: Formalisation of the UML Profile for SDL - A Case Study / Department of Computer Science, University of Kaiserslautern. 2006 (352/06). – Forschungsbericht

[23] Grammes, Rüdiger ; Gotzhein, Reinhard: Towards the Harmonisation of UML and SDL - Syntactic and Semantic Alignment - / Department of Computer Science, University of Kaiserslautern. 2003 (327/03). – Forschungsbericht

[24] Grammes, Rüdiger ; Gotzhein, Reinhard: Towards the Harmonisation of UML and SDL. In: Frutos-Escrig, David de (Hrsg.) ; Núñez, Manuel (Hrsg.): *Formal Techniques for Networked and Distributed Systems - FORTE 2004, Madrid, Spain* Bd. 3235, Springer, Januar 2004 (LNCS), S. 61–78

[25] Grammes, Rüdiger ; Gotzhein, Reinhard: SDL Profiles - Definition and Formal Extraction / Department of Computer Science, University of Kaiserslautern. 2006 (350/06). – Forschungsbericht

[26] Grammes, Rüdiger ; Gotzhein, Reinhard: SDL Profiles - Formal Semantics and Tool Support. In: Lopes, Antónia (Hrsg.) ; Dwyer, Matt (Hrsg.): *Fundamental Approaches to Software Engineering. 10th International Conference, FASE 2007 : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 2007, Proceedings* Bd. 4422, Springer, 2007 (LNCS), S. 200–214

[27] Gurevich, Yuri: Evolving Algebras 1993: Lipari Guide. In: Börger, Egon (Hrsg.): *Specification and Validation Methods.* Oxford University Press, 1995, S. 9–36

[28] Gurevich, Yuri: May 1997 Draft of the ASM Guide / EECS Department, University of Michigan. 1997 (CSE-TR-336-97). – Forschungsbericht

[29] Gurevich, Yuri ; Huggins, James K.: The Semantics of the C Programming Language. In: Börger, Egon (Hrsg.) ; Martini, Simone (Hrsg.) ; Jäger, Gerhard (Hrsg.) ; Büning, Hans K. (Hrsg.) ; Richter, Michael M. (Hrsg.): *Computer Science Logic. 6th Workshop, CSL '92 San Miniato, Italy, September 28 - October 2, 1992 Selected Papers* Bd. 702, Springer, 1993 (LNCS), S. 274–308

[30] Haugen, Øystein: Comparing UML 2.0 Interactions and MSC-2000. In: Amyot, Daniel (Hrsg.) ; Williams, Alan W. (Hrsg.): *System Analysis and Modeling. 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004, Revised Selected Papers.* Bd. 3319, Springer, 2005 (LNCS), S. 65–79

[31] ITU: *Recommendation Z.100 (03/93): Specification and Description Language (SDL).* Geneva, 1993

[32] ITU: *Recommendation Z.100 Addendum 1 (10/96): Specification and Description Language (SDL).* Geneva, 1996

[33] ITU: *Recommendation Z.109: SDL combined with UML.* Geneva, 2000

[34] ITU: *Recommendation Z.120: Message Sequence Charts.* Geneva, 2000

[35] ITU: *Recommendation Z.100 (08/02): Specification and Description Language (SDL).* Geneva, 2002

[36] ITU: *Recommendation Z.100 (2002) Amendment 1 (10/03): Specification and Description Language (SDL).* Geneva, 2003

[37] ITU: *Recommendation Z.100 (2002) Corrigendum 1 (08/04): Specification and Description Language (SDL).* Geneva, 2004

[38] ITU: *Recommendation Z.119: Guidelines for UML profile design.* Geneva, 2005

[39] ITU: *Recommendation Z.109: The UML Profile for SDL.* Geneva, 2006

[40] ITU Study Group 10: *Draft Z.100 Annex F1 (11/00).* 2000

[41] ITU Study Group 10: *Draft Z.100 Annex F2 (11/00).* 2000

[42] ITU Study Group 10: *Draft Z.100 Annex F3 (11/00).* 2000

[43] ITU Study Group 17: *UML Profile for SDL.* 2005. – Draft Recommendation Z.109

[44] Jones, Cliff B.: The META-Language: A Reference Manual. In: Bjørner, Dines (Hrsg.) ; Jones, Cliff B. (Hrsg.): *The Vienna Development Method: The Meta-Language* Bd. 61, Springer, 1978 (LNCS), S. 218–277

[45] Löwis, Martin von ; Piefel, Michael: The Term Processor Kimwitu++. In: Callaos, Nagib (Hrsg.) ; Hernández-Encinas, Luis (Hrsg.) ; Yetim, Fahri (Hrsg.): *SCI 2002: The 6th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, USA*, 2002, S. 182–186

[46] Neumann, Toby ; Piefel, Michael: *Kimwitu++: A Term Processor. Users guide 1.0.* Berlin, Germany: Humboldt-University Berlin, 2002

[47] Nowack, Antje: Slicing Abstract State Machines. In: Zimmermann, Wolf (Hrsg.) ; Thalheim, Bernhard (Hrsg.): *Abstract State Machines 2004. Advances in Theory and Practice, Lutherstadt Wittenberg, Germany* Bd. 3052, Springer, Januar 2004 (LNCS), S. 186–201

[48] Nowack, Antje: *A Polynomial-Time Slicing Algorithm.* Proceedings of the 12th International Workshop on Abstract State Machines (ASM 2005), March 8-11, 2005, Paris, France, 2005

[49] Ober, Ileana: An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. In: Börger, Egon (Hrsg.) ; Gargantini, Angelo (Hrsg.) ; Riccobene, Elvinia (Hrsg.): *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, Taormina, Italy* Bd. 2589, Springer, 2003 (LNCS), S. 356–371

[50] Object Management Group: *Unified Modeling Language Specification, Version 1.3*. 2000. – www.uml.org

[51] Object Management Group: *Meta Object Facility (MOF) Specification, Version 1.4*. 2002. – www.omg.org

[52] Object Management Group: *Unified Modeling Language: Superstructure, Version 2.0*. 2005. – www.uml.org

[53] Object Management Group: *Meta Object Facility (MOF) Core Specification, Version 2.0*. 2006. – www.omg.org

[54] Object Management Group: *Object Constraint Language (OCL), Version 2.0*. 2006. – www.omg.org

[55] Pfahler, Peter ; Kastens, Uwe: Language Design and Implementation by Selection. In: *Proceedings of the 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages, DSL '97, Paris, France*, University of Illinois at Urbana-Champaign, 1997, S. 97–108

[56] Pfahler, Peter ; Kastens, Uwe: Configuring Component-based Specifications for Domain-Specific Languages. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Januar 2001

[57] Prinz, Andreas: *SMILE – Semantic Middleware for System Modelling Languages*. 2003. – Specific Targetet Research Project Proposal in the 2nd call of the IST priority

[58] Prinz, Andreas ; Löwis, Martin von: Generating a Compiler for SDL from the Formal Language Definition. In: Reed, Rick (Hrsg.) ; Reed, Jeanne (Hrsg.): *SDL 2003: System Design* Bd. 2708, Springer, 2003 (LNCS), S. 150–165

[59] Schellhorn, Gerhard: Verification of ASM Refinements Using Generalized Forward Simulation. In: *Journal of Universal Computer Science* 7 (2001), Nr. 11, S. 952–979

[60] Schellhorn, Gerhard: ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. In: *Theoretical Computer Science* 336 (2005), Nr. 2-3, S. 403–436

[61] SDL Task Force: *SDL+ - The Simplest, Useful 'Enhanced SDL-Subset' for the Implementation and Testing of State Machines*. 2005. – www.sdltaskforce.org

[62] Selic, Bran ; Rumbaugh, Jim: *Mapping SDL to UML.* Rational Software Whitepaper, 1999

[63] Stärk, Robert ; Schmid, Joachim ; Börger, Egon: *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer, 2001

[64] Stärk, Robert F. ; Nanchen, Stanislas: A Logic for Abstract State Machines. In: *Journal of Universal Computer Science* 7 (2001), Nr. 11, S. 980–1005

[65] Weber, Christian: *Entwurf und Implementierung eines konfigurierbaren SDL Transpilers für eine C++ Laufzeitumgebung*, University of Kaiserslautern, Germany, Diplomarbeit, Dezember 2005

[66] Werner, Constantin ; Kraatz, Sebastian ; Hogrefe, Dieter: A UML Profile for Communicating Systems. In: Gotzhein, Reinhard (Hrsg.) ; Reed, Rick (Hrsg.): *SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany* Bd. 4320, Springer, 2006 (LNCS), S. 1–18

Index

abstract grammar, 22
Abstract State Machine, 9
    Distributed ASM, 14
    Real-time ASM, 16
abstract transition system, 65
action, 11
Active, 70
activity diagram, 8
agent, 14
agent control block, 19
agent mode, 19
ASM slicing, 107

behaviour diagram, 8
behaviour primitive, 18

channel, 5
class diagram, 8
coherence condition, 14, 66
compilation function, 19
composite state, 6
composite state graph, 6
composite structure diagram, 8
composition, 77
concatenation, 22
consistency, 63
consistency relation, 64
ConTraST, 106

data equivalence, 64
dead rule recognition, 78
default value, 83
dynamic semantics, 18

environment, 15
Exec, 70
extraction, 77

false, 85
foreach-statement, 115

gate, 5

idToNode, 56
initial segment, 14
inport, 5
interaction diagram, 8

kimwitu++, 113

language core, 62
language module, 62
language profile, 1, 62
location, 11

Mapping, 55
$Mapping_{trg}$, 59
meta-model, 8, 23
meta-model hierarchy, 7
metameta-model, 7
model, 8
MOF, 7

nupd, 99
NUpdate, 101

OCL, 52

package diagram, 8
partially-ordered run, 14
phylum, 113
program, 14
proof obligation, 97

reduction profile, 83
refinement, 64

remove, 84
reserve, 11
rewrite rule, 114
rule coverage, 78

SAM agent, 18
SDL, 4
SDL abstract machine, 18
SDL agent, 4
SDL agent set, 4
SDL profile, 71
    Cadvanced, 72
    Cmicro, 72
    Core, 72
    Dynamic, 72
    SAFIRE, 73
    $Static_1$, 72
    $Static_2$, 72
    UML profile for SDL, 73
SDL virtual machine, 18
SDL-profile tool, 111
sequence diagram, 9
sequential run, 14
signal, 5
signal flow, 18
state, 9, 10
state aggregation, 6
state partition, 6
statemachine, 6
statemachine diagram, 9
static semantics, 16
stereotype, 48
structure diagram, 8
synonym, 22

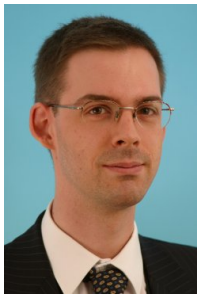
tag definition, 48, 49
term, 22
toId, 53
true, 85

UML, 7
UML profile, 47
UML Profile for SDL, 49
undefined, 85
unparse rule, 114
upd, 99
Update, 99
update, 12
update set, 12

Vienna Development Method, 22
vocabulary, 10

with-statement, 115

Lebenslauf



Persönliche Daten

Name:	Rüdiger Grammes
Geburtsdatum:	03.08.1978
Geburtsort:	Mainz
Familienstand:	ledig

Schulausbildung

1985-1989	Grundschule, Langenlonsheim
1989-1998	Stefan-George Gymnasium, Bingen
07/1998	Abitur

Hochschulausbildung

10/1998 - 03/2003	Angewandte Informatik, Universität Kaiserslautern Schwerpunkt Eingebettete Systeme
09/2002 - 02/2003	Diplomarbeit “Evaluierung und Anwendung des SDL-Pattern-Ansatzes” (Firma Siemens, Preis des Freundeskreises der Universität Kaiserslautern)
03/2003	Diplom-Informatiker
03/2003 - 06/2007	Promotionsprogramm Informatik