Technical Report

SDL Profiles - Definition and Formal Extraction

R. Grammes R. Gotzhein

November 20, 2006

University of Kaiserslautern

Contents

Over a period of 30 years, ITU-T's Specification and Description Language (SDL) has matured to a sophisticated formal modelling language for distributed systems and communication protocols. The language definition of SDL-2000, the latest version of SDL, is complex and difficult to maintain. Full tool support for SDL is costly to implement. Therefore, only subsets of SDL are currently supported by tools. These SDL subsets - called *SDL profiles* - already cover a wide range of systems, and are often sufficient in practice. In this report, we present our approach for extracting the formal semantics for SDL profiles from the complete SDL semantics. We then formalise the approach, present our SDL-profile tool, and report on our experiences.

1 Introduction

Over a period of 30 years, ITU-T's Specification and Description Language (SDL) [6, 7, 8] has matured from a simple, informal graphical notation for describing a set of communicating finite state machines to a sophisticated formal modelling technique with graphical syntax, data types, structuring mechanisms, object-oriented features, formal semantics, support for reuse, companion notations, and commercial tool environments. This development has led to an expressive and sophisticated language for a wide range of domains. On the other hand, the language definition of SDL-2000, the latest version of SDL, is complex and difficult to maintain. Full tool support for SDL is costly to implement. Therefore, all commercial tool providers have decided to support subsets of SDL only. These SDL subsets - also called *SDL profiles* in this report - are targeted towards specific domains and companies, where, due to their reduced complexity, they are preferred by engineers.

While the use of SDL profiles is today's state-of-the-practice, their definition is not reflected in the SDL standard. One could argue that this is of no particular importance, since the full language definition covers all possible subsets. However, a drawback is that engineers working with a well-defined SDL profile only are still confronted with the entire language definition. Also, the task of tool builders to show conformance to the language definition is highly complex, in particular if the optimisation potential of a particular SDL profile is to be exploited.

To solve these problems, one could think of defining a separate standard for each SDL profile of interest. This, however, creates other problems arising from the extra work to define and maintain these standards, and of keeping them consistent. In this report, we address these problems and present a formalised, tool-based approach for extracting, for a given SDL profile, the formal semantics from the standardised SDL semantics. In Section 2, we survey the language definition of SDL, in particular, the formal semantics. Section 3 outlines our approach to the extraction of the formal semantics for SDL profiles. In Section 4, the complete formalisation of the approach is presented. Consistency of SDL profiles is visited in Section 5. We present our tool chain for the extraction approach in Section 6, survey related work in Section 7, and draw conclusions in Section 8.

2 Language Definition of SDL

2.1 Specification and Description Language (SDL)

The Specification and Description Language (SDL) [6] is a formal language standardised by the International Telecommunications Union (ITU). It is widely used both in industry and academia. SDL is based on the concept of asynchronously communicating 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 semantics was assessed as being too difficult to extend and maintain, a new formal semantics, based on Abstract State Machines, was defined from scratch [2, 9, 10].

2.2 Abstract State Machines

Abstract State Machines (ASMs) [4, 5] 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 many-sorted first-order structures, called *states*. A state consists of a signature containing domain names, function names and relation names, together with an interpretation of these names. 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.

ASM-Listing 1 shows the signature for the specification of a graph, consisting of domain names *Node* and *Edge* (defined as a pair of nodes), a 0-ary function name *start* and an unary function names *weight* and *visited*. A state of the ASM defines an interpretation of the names in the signature. *Node* and *Edge* are interpreted as sets of elements, *start* is interpretated as an element of the set *Node* (or the special element `undefined`), *weight* and *visited* are interpretated as functions from elements of set *Edge* to the set of natural numbers and the set of boolean values, respectively.

```
1 domain Node
2 domain Edge =_def Node × Node
3
4 start: → Node
5 weight: Edge → ℕ
```

```
6 visited : Edge → BOOLEAN
```

ASM-Listing 1: Vocabulary of a Graph Specification

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 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.

A detailed introduction to Abstract State Machines and the syntax used in this report is given in [2].

2.3 Formal Definition of SDL-2000

In November 2000, the formal semantics of SDL-2000 was officially approved to become part of the SDL language definition. It covers all static and dynamic language aspects, and consists of two major parts (for a detailed survey, see [2]):

- The *static semantics* of SDL defines well-formedness conditions on the concrete syntax of SDL. Transformations map extended features of SDL to core features of the language, reducing the complexity of the dynamic semantics.
- 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), providing a signal flow model and several types of agents. Agents go through an initialisation phase, creating the nested structure of an SDL system, and an execution phase, forwarding signals and executing state machines. Behaviour primitives form the instruction set of the SVM, defining basic actions such as sending signals, setting timers or calling procedures. A *compilation function* maps actions from transitions in an SDL specification to instructions of the SVM.

ASM Listing 2 shows an excerpt from the SVM, which provides a formal definition for the setting of timers, encapsulated in the ASM rule macro SETTIMER. When evaluated, the macro defines an update of the input queue of an SDL agent, where a possibly existing timer instance is removed, and a new timer instance is appended.

```
1 SETTIMER(tm: TIMER, vSeq: VALUE*, t: [TIME]) ≡
2   let tmi = mk–TimerInst(Self.self, tm, vSeq ) in
3     if t = undefined then
4       Self.inport.schedule := insert(tmi, now + tm.duration, delete(tmi, Self.inport.schedule)
            )
5       tmi.arrival := now + tm.duration
6     else
7       Self.inport.schedule := insert(tmi, t, delete(tmi, Self.inport.schedule))
```

```
8           tmi.arrival := t
9       endif
10    endlet
```

ASM-Listing 2: Setting SDL Timers

The let-rule in line 2 defines a shortcut *tmi* for a new TimerInst tuple, consisting of the process identifier of the executing agent *Self.self*, the type of the timer *tm*, and a sequence of parameters *vSeq*. In case no time t is passed as a parameter to the rule macro, *tmi* is inserted into the schedule of the process (line 4) at the current time *now* plus the default duration of the timer type. In the same step, all timer tuples that are structurally equivalent to *tmi* (same type, parameters and agent) are deleted from the schedule. In case a time t is passed, the timer instance is inserted into the schedule at t (line 7).

3 Definition and Extraction of SDL Profiles

3.1 SDL Profiles

SDL has become a sophisticated and complex language with many language features. This results in a large and extensive language definition. In the formal semantics of SDL-2000, the operational nature of ASMs and the extensive use of modularisation lead to a readable formal semantics definition. However, due to the complexity of the language, the formal semantics is large and requires substantial effort to be understood completely: the dynamic semantics of SDL-2000 consist of more than 3000 lines of ASM specification.

The problem of the complexity of SDL-2000 has been identified, and the definition of simpler sublanguages of SDL has been proposed. One such language is defined by the SDL Task Force as the simplest useful enhanced subset of SDL (referred to as 'SDL+') [17]. This language is implemented by the SAFIRE tool, and here is called SAFIRE. SAFIRE focuses on the state machine aspect of SDL, and enhances it with functionality needed for testing. However, although a formal semantics exists for SDL, none is provided for SAFIRE.

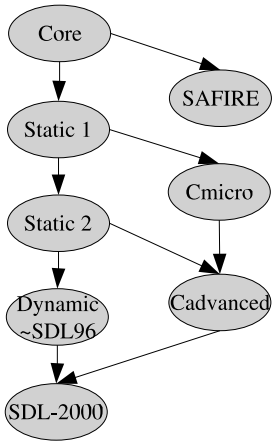

Figure 3.1: Superset Relationship between Language Profiles

SDL profiles are self-contained subsets of SDL, possibly enhanced by further language constructs, targeted towards specific domains and companies. In comparison to the full SDL language definition, SDL profiles have reduced complexity, which supports both engineers and tool builders. A sublanguage like SAFIRE is an SDL profile. Tools for an SDL profile can be developed faster, leading to less expensive tools and enabling code optimisations. Possible SDL profiles could also be derived from the supported features of the code generators Cbasic and Cadvanced in Telelogic Tau.

Apart from being subsets of the complete language, SDL profiles can be subsets of other SDL profiles, forming a hierarchy profiles. We have defined four SDL profiles. The smallest profile is *Core*, which contains a minimal set of features. $Static_1$, $Static_2$ and *Dynamic* extend *Core*, each profile adding additional features to the preceding one, *Dynamic* being roughly the equivalent of SDL-96. The subset relationships between different language profiles are shown in Figure 3.1. The SDL profiles *Core*, $Static_1$, $Static_2$ and *Dynamic* are supported syntactically by our configurable transpiler ConTraST [1].

3.2 Outline of the Extraction Approach for SDL Profiles

Being language definitions themselves, SDL profiles consist of a formal syntax and a formal semantics. To reduce the overhead of defining SDL profiles and to keep them consistent, a systematic, formalised approach is needed. Basically, there are two ways to define SDL profiles:

- **Composition (bottom-up).** Here, SDL profiles are defined by composing an SDL language core and selected SDL language modules. The *language core* can be understood as the smallest useful SDL subset, 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. Examples of SDL language modules are timers, exceptions, 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.

- **Extraction (top-down).** Here, SDL profiles are defined by extracting the profile definition from the complete SDL language definition. As in the composition approach, SDL language modules, each consisting of a set of SDL language constructs, can be identified. To obtain a particular SDL 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. A major problem to be solved is the reduction of the formal semantics.

From a methodological point of view, the composition approach seems more appealing. However, there is the difficulty of encapsulating the formal semantics of SDL language modules such that composition is supported, which we have not been able to overcome. Therefore, we have investigated the extraction approach, with quick first results. Meanwhile, we have further developed, consolidated, and formalised the extraction approach, and have built a sophisticated tool chain.

To extract the formal semantics of an SDL profile from the complete SDL semantics, we have considered two approaches:

- **ASM rule coverage.** With each SDL profile, an ASM rule coverage comprising all ASM rules of the SVM that may be evaluated in some execution of some SDL specification written in that SDL profile can be associated. While this approach is semantically sound, it is practically infeasible. For a given SDL specification, the concurrent, non-deterministic nature of the SVM may lead to a very large number of possible executions. Furthermore, the number of SDL specifications that can be written in a given SDL profile is extremely large. 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 SDL specifications, we can develop safe criteria to recognise ASM rules that are never evaluated for a given SDL 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, i.e. ASM rules must only be removed if it can be formally proven that they are not evaluated for a given SDL 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.

4 Formalisation

4.1 Reduction Profile

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

- The abstract syntax yields part of the SVM data structure (ASM signature, see Figure 4.1). For each element of the abstract grammar, a domain of the same name is introduced in the ASM signature. E.g., 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 (e.g., assignments, setting timers), a compilation function maps parts of the abstract syntax to domains of the formal semantics definition that form the SVM. E.g., 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.

Following the extraction approach, we remove SDL language modules from the formal language definition. Language modules consist of sets of SDL 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 4.1). 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 SVM, and, since they are not modified by the SVM, will be empty in all reachable states, too. This observation is fundamental for recognising dead ASM rules of the SVM.

Apart from domains corresponding to elements of the abstract grammar of a language module, other domains, functions and predicates in the SVM signature correspond to specific language modules. E.g., *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 SVM 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 SDL profile. E.g., we can show that *SignalSaved* is always false if *save-signalset* is empty.

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 SDL profile in a

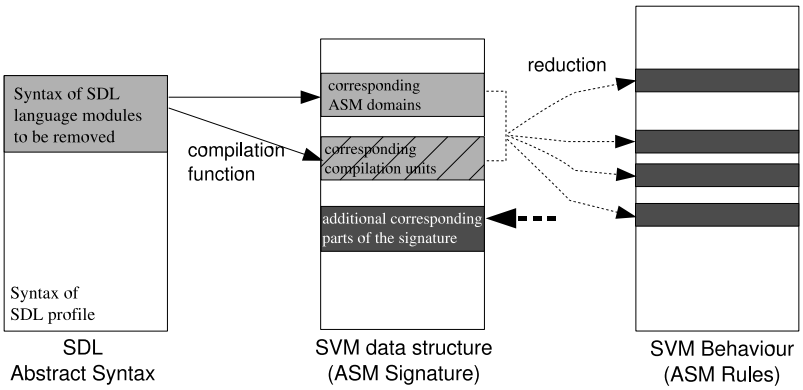

Figure 4.1: Concept of the Extraction Process

Save-signalset
SignalSaved = false

Figure 4.2: Reduction Profile for 'Save' Feature

reduction profile. The reduction profile is a list of domains, functions and predicates from the SVM signature to be removed in the extraction process. Additionally, we specify a default value (`true` or `false`) for predicates. These elements are removed from the formal language definition according to the rules formally defined in the following sections.

Figure 4.2 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.

4.2 Formalisation Signature

We now formalise our approach for extracting the formal semantics of SDL profiles from the complete SDL 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 SDL 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 [3]. The input of the reduction is the SDL formal semantics definition from [10] and a reduction profile r, as described in Section 4.1.

To formalise the extraction, we define a function *remove*, 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 $false_r$, that control the reduction. Predicates $undefined_r$, $true_r$ and $false_r$ form the interface to the reduction profile. Furthermore, $true_r$ and $false_r$ define heuristics to determine if an expression is `true` (`false`) in every reachable state.

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

The *remove* function is defined on all elements of the grammar G. Predicates $true_r$ and $false_r$ are defined on boolean and first-order logic expressions, and predicate $undefined_r$ on all expressions. In the following sections, we omit the index r from the predicates.

The function *remove* is defined recursively - a given term is mapped to a new term by applying the mapping defined by *remove* to the subterms. In case none of the predicates *undefined*, *true* and *false* holds, the current term is not reduced any further. This assures in particular that *remove* corresponds to the identical mapping if the reduction profile r is empty. In other cases, subterms can be replaced or omitted depending on which of the predicates hold.

4.3 Formal Reduction of ASMs

4.3.1 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 can be removed in a subsequent cleanup step. The following variables are used to define the 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.

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

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

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.

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

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

4.3.2 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.

$$
\begin{array}{lll}
\mathit{remove}(\text{ps }\text{','}\ x\ \text{':'}\ D, \mathcal{V}) = & & \\
\quad \mathit{remove}(\text{ps}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D) \\
\quad \mathit{remove}(\text{ps}, \mathcal{V})\ \text{','}\ x\ \text{':'}\ \mathit{remove}(D, \mathcal{V}) & & \text{else}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{remove}(x\ \text{':'}\ D, \mathcal{V}) = & & \\
\quad \text{''} & \text{iff} & \mathit{undefined}(D) \\
\quad x\ \text{':'}\ \mathit{remove}(D, \mathcal{V}) & & \text{else}
\end{array}
$$

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).

$$
\begin{array}{l}
\mathit{numfpar}(\text{fcs }\text{','}\ \text{exp}) = \mathit{numfpar}(\text{fcs}) + 1 \\
\mathit{numfpar}(\text{exp}) = 1 \\
\mathit{numfpar}(\text{ps }\text{','}\ x\ \text{':'}\ D) = \mathit{numfpar}(\text{ps}) + 1 \\
\mathit{numfpar}(x\ \text{':'}\ D) = 1
\end{array}
$$

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.

$$
\begin{array}{lll}
\mathit{remfpar}(\text{ps }\text{','}\ x\ \text{':'}\ D) = & & \\
\quad \{x\} \cup \mathit{remfpar}(\text{ps}) & \text{iff} & \mathit{undefined}(D) \\
\quad \mathit{remfpar}(\text{ps}) & & \text{else}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{remfpar}(x\ \text{':'}\ D) = & & \\
\quad \{x\} & \text{iff} & \mathit{undefined}(D) \\
\quad \{\} & & \text{else}
\end{array}
$$

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 $2^2 + 2^0 = 5$.

$$
\begin{array}{lll}
\mathit{code}(\text{ps }\text{','}\ x\ \text{':'}\ D, n) = & & \\
\quad \mathit{code}(\text{ps}, n-1) + 2^n & \text{iff} & \mathit{undefined}(D) \\
\quad \mathit{code}(\text{ps}, n-1) & & \text{else}
\end{array}
$$

$$
\begin{array}{l}
code(x\ ':'\ D, n) = \\
\quad 2^n \qquad\qquad\qquad\qquad\qquad\qquad \text{iff} \quad undefined(D) \\
\quad 0 \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \text{else}
\end{array}
$$

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

The function *removepar* removes arguments from a macro call corresponding to undefined formal parameters in the macro definition. The number n corresponds the 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.

$$
\begin{array}{l}
removepar(\text{fcs ',' exp}, n, p) = \\
\quad removepar(\text{fcs}, n-1, p-2^n) \qquad\qquad \text{iff} \quad p - 2^n > 0 \\
\quad removepar(\text{fcs}, n-1, p)\ \text{','}\ \text{exp} \qquad\qquad\quad\ \ \text{else}
\end{array}
$$

$$
\begin{array}{l}
removepar(\text{exp}, n, p) = \\
\quad \text{''} \qquad\qquad\qquad\qquad\qquad\qquad \text{iff} \quad p = 2^n \ \text{// n should be 0} \\
\quad \text{exp} \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \text{else}
\end{array}
$$

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

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

4.3.3 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. All together, 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.

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

The mapping of the **if**-rule (see below) 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]. 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.

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

In **let**-rules, we can additionally specify a type for the variable x in form of a domain D. In that case, we can remove the **let**-rule in the same manner as if the expression exp were undefined, if the domain D is undefined.

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.

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

$remove($**extend** dn **with** x R **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

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. 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 infinie set.

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

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

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 4.3.5), we can can conclude that these predicates will not hold for a constraint of this form.

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. 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	$false(\text{exp}, \mathcal{V}) \vee true(\text{exp}, \mathcal{V}) \vee$ $undefined(\text{exp}, \mathcal{V})$
do forall x ':' $remove(\text{exp}, \mathcal{V})$ $remove(R, \mathcal{V})$ **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 can be done in a subsequent optimisation step.

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

4.3.4 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.

A domain is removed if it is undefined. A union or product of several domains is removed if all of the domains are undefined, otherwise only the undefined subdomains are removed.

$remove(\text{dn}, \mathcal{V}) =$
  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
$remove((D)\text{–}\mathbf{set}, \mathcal{V}) =$
  nodomain iff $undefined(D, \mathcal{V})$
  $(remove(D, \mathcal{V}))\text{–}\mathbf{set}$ 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 $undefined(s_1 \times s_2, \mathcal{V})$
  $remove(s_1, \mathcal{V})$ iff $undefined(s_2, \mathcal{V})$
  $remove(s_2, \mathcal{V})$ iff $undefined(s_1, \mathcal{V})$
  $remove(s_1, \mathcal{V}) \times remove(s_2, \mathcal{V})$ else
$remove(t \times s, \mathcal{V}) =$
  ” iff $undefined(t \times s, \mathcal{V})$
  $remove(t, \mathcal{V})$ iff $undefined(s, \mathcal{V})$
  $remove(s, \mathcal{V})$ iff $undefined(t, \mathcal{V})$
  $remove(t, \mathcal{V}) \times remove(s, \mathcal{V})$ else
$remove(\text{'(' ')'}, \mathcal{V}) = \text{'(' ')'}$

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

$$
\begin{array}{lll}
\mathit{remove}(u \cup s, \mathcal{V}) = & & \\
\quad \text{''} & \text{iff} & \mathit{undefined}(u \cup s, \mathcal{V}) \\
\quad \mathit{remove}(u, \mathcal{V}) & \text{iff} & \mathit{undefined}(s, \mathcal{V}) \\
\quad \mathit{remove}(s, \mathcal{V}) & \text{iff} & \mathit{undefined}(u, \mathcal{V}) \\
\quad \mathit{remove}(u, \mathcal{V}) \cup \mathit{remove}(s, \mathcal{V}) & & \text{else}
\end{array}
$$

$\mathit{remove}(\text{'\{' ics '\}'}, \mathcal{V}) = \textbf{if } \mathit{undefined}(\text{ics}, \mathcal{V}) \textbf{ then } \text{''} \textbf{ else } \text{'\{' } \mathit{remove}(\text{ics}, \mathcal{V}) \text{ '\}'} \textbf{ endif}$

The predicate undefined on domains specifies if a domain expression is undefined, given the basic domains that have been removed. A domain expression is undefined if the domain name it comtains is undefined. In case of union and product 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.

$$
\begin{array}{rll}
\mathit{undefined}(\text{dn}\underline{*}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\text{dn}\underline{**}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\text{dn}\underline{+}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\text{dn–}\textbf{set}, \mathcal{V}) & \text{iff} & \mathit{undefined}(dn) \\
\mathit{undefined}(\underline{(}D\underline{)}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{(}D\underline{)}\text{–}\textbf{set}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}\underline{*}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(\underline{[}D\underline{]}\underline{+}, \mathcal{V}) & \text{iff} & \mathit{undefined}(D, \mathcal{V}) \\
\mathit{undefined}(D_1 \rightarrow D_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(D_1, \mathcal{V}) \vee \mathit{undefined}(D_2, \mathcal{V})
\end{array}
$$

$$
\begin{array}{rll}
\mathit{undefined}(s_1 \times s_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(s_1, \mathcal{V}) \wedge \mathit{undefined}(s_2, \mathcal{V}) \\
\mathit{undefined}(t \times s, \mathcal{V}) & \text{iff} & \mathit{undefined}(t, \mathcal{V}) \wedge \mathit{undefined}(s, \mathcal{V}) \\
\mathit{undefined}(\text{`(` `)`}, \mathcal{V}) & \text{iff} & \text{false}
\end{array}
$$

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

$$
\begin{array}{rll}
\mathit{undefined}(\text{'\{'ics'\}'}, \mathcal{V}) & \text{iff} & \mathit{undefined}(\text{ics}, \mathcal{V}) \\
\mathit{undefined}(\text{ics}\text{','}\, x, \mathcal{V}) & \text{iff} & x \in \mathcal{V} \\
\mathit{undefined}(\text{ics}\text{','}\, \text{kw}, \mathcal{V}) & \text{iff} & \mathit{undefined}(\text{ics}, \mathcal{V}) \wedge \mathit{undefined}(\text{kw}) \\
\mathit{undefined}(\text{ics}\text{','}\, \text{kw1 kw2}, \mathcal{V}) & \text{iff} & \mathit{undefined}(\text{ics}, \mathcal{V}) \wedge \mathit{undefined}(\text{kw1}) \wedge
\end{array}
$$

$$\textit{undefined}(\text{kw2})$$
$$\textit{undefined}(\text{ics}', '\texttt{Literal}, \mathcal{V}) \quad \text{iff} \quad \text{false}$$

4.3.5 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 [3]. 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
-	$\neg\mathsf{T} \wedge \neg\mathsf{F} \wedge \neg\mathsf{U}$

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 to 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. 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	$\neg e_1$

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_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	$e_1 \vee e_2$

$e_1 \wedge e_2$ is defined as $\neg(\neg e_1 \vee \neg e_2)$, the truth table is derived accordingly.

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	$e_1 \wedge e_2$

$e_1 \rightarrow e_2$ is defined as $\neg e_1 \vee e_2$, the truth table is derived accordingly.

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	$\neg e_1$	$e_1 \rightarrow e_2$

$e_1 \leftrightarrow e_2$ is defined as $(e_1 \rightarrow e_2) \wedge (e_2 \rightarrow e_1)$, the truth table is derived accordingly.

$$
\begin{array}{rll}
\mathit{true}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) & \text{iff} & (\mathit{true}(e) \wedge \mathit{true}(e_1)) \vee \\
 & & (\mathit{false}(e) \wedge \mathit{true}(e_2)) \\
\mathit{false}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) & \text{iff} & (\mathit{true}(e) \wedge \mathit{false}(e_1)) \vee \\
 & & (\mathit{false}(e) \wedge \mathit{false}(e_2)) \\
\mathit{undefined}(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ endif}) & \text{iff} & (\mathit{true}(e) \wedge \mathit{undefined}(e_1)) \vee \mathit{undefined}(e) \\
 & & \vee(\mathit{false}(e) \wedge \mathit{undefined}(e_2)) \vee \\
 & & (\mathit{undefined}(e_1) \wedge \mathit{undefined}(e_2))
\end{array}
$$

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 interpretate 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.

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

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

`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

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	$e_1 \leftrightarrow e_2$

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.

	e_2	op $\in \{<, >, \leq, \geq\}$	
e_1	op	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

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_2		
e_1	$\in$	U	-
	U	F	F
	-	F	-

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

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 interpretate 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 $e = \texttt{undefined}$, it also holds for $\texttt{undefined} = e$.

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

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

Arithmetic Operators

Arithmetic operators are removed if one of their subexpressions is undefined. Removal is less strict for plus and minus, which are only removed if both expressions are undefined.

$$
\begin{array}{lll}
\mathit{remove}(-e, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\quad -\mathit{remove}(e, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 + e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) + \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 - e_2, \mathcal{V}) = & & \\
\quad \texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_2, \mathcal{V}) \\
\quad \mathit{remove}(e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \\
\quad \mathit{remove}(e_1, \mathcal{V}) - \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 * e_2, \mathcal{V}) = & &
\end{array}
$$

$$
\begin{array}{lll}
\texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{remove}(e_1, \mathcal{V}) * \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \,/\, e_2, \mathcal{V}) = & & \\
\texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{remove}(e_1, \mathcal{V}) \,/\, \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \text{ MOD } e_2, \mathcal{V}) = & & \\
\texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{remove}(e_1, \mathcal{V}) \text{ MOD } \mathit{remove}(e_2, \mathcal{V}) & & \text{else} \\
\mathit{remove}(e_1 \text{ DIV } e_2, \mathcal{V}) = & & \\
\texttt{undefined} & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{remove}(e_1, \mathcal{V}) \text{ DIV } \mathit{remove}(e_2, \mathcal{V}) & & \text{else}
\end{array}
$$

$$
\begin{array}{rll}
\mathit{undefined}(-e, \mathcal{V}) & \text{iff} & \mathit{undefined}(e, \mathcal{V}) \\
\mathit{undefined}(e_1 + e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 - e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \wedge \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 * e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \,/\, e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \textbf{ mod } e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V}) \\
\mathit{undefined}(e_1 \textbf{ rem } e_2, \mathcal{V}) & \text{iff} & \mathit{undefined}(e_1, \mathcal{V}) \vee \mathit{undefined}(e_2, \mathcal{V})
\end{array}
$$

Sets and Sequences

Sequence (set) composition constructs a sequence (set) by applying expression e_1 to elements from e_2 for which the boolean expression e_3 holds. If expression e_1 or e_3 are undefined, so is the sequence/set. In that case, *false* will hold when testing if an element is contained in the structure. 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.

$$< e_1 \mid x \in e_2 : e_3 >, \quad \{e_1 \mid x \in e_2 : e_3\}$$

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

$\{remove(e_1, \mathcal{V}) \mid x \in remove(e_2, \mathcal{V})$ else
$: remove(e_3, \mathcal{V})\}$

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

Operations on sets and sequences need at least one argument for which *undefined* doesn't hold, otherwise *undefined* holds for the operation. Removal is straightforward, the definition is equivalent to the definition for the arithmetic operators plus and minus.

		op $\in \{\mid\mid, \bigcup\}$	
e_1	op	U	-
		U	-

	e_2	op $\in \{^\cap, \cup, \cap\}$	
e_1	op	U	-
U		U	-
-		-	-

For the operations map ($\mapsto$) and range (..), *undefined* holds if *undefined* holds for any subexpression. The definition of removal is equivalent to the arithmetic operator *.

	e_2	op $\in \{\mapsto, ..\}$	
e_1	op	U	-
U		U	U
-		U	-

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 4.3.2. *count*is a natural number that holds the information which formal parameters were removed.

$remove(\texttt{MkName}(), \mathcal{V}) =$
- `undefined` iff $undefined(\texttt{MkName})$
- `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

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

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.

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

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

Names and keywords in expressions are removed if they are marked as undefined, otherwise they are left untouched. Predicate *undefined* never holds for literals (for example, natural numbers) and the special ASM element `undefined`.

$\mathit{remove}(\texttt{DomainName}, \mathcal{V}) =$ **if** $\mathit{undefined}(\texttt{DomainName})$ **then** `undefined` **else** `DomainName` **endif**
$\mathit{remove}(\texttt{SynName}, \mathcal{V}) =$ **if** $\mathit{undefined}(\texttt{SynName})$ **then** `undefined` **else** `SynName` **endif**
$\mathit{remove}(\texttt{ProgramName}, \mathcal{V}) =$ **if** $\mathit{undefined}(\texttt{ProgramName})$ **then** `undefined` **else** `ProgramName` **endif**
$\mathit{remove}(\texttt{Keyword}, \mathcal{V}) =$ **if** $\mathit{undefined}(\texttt{Keyword})$ **then** `undefined` **else** `Keyword` **endif**
$\mathit{remove}(\texttt{Literal}, \mathcal{V}) = \texttt{Literal}$
$\mathit{remove}(\texttt{undefined}, \mathcal{V}) = \texttt{undefined}$
$\mathit{remove}(\underline{\leq}\text{pcs}\underline{\geq}, \mathcal{V}) = \underline{\leq}\mathit{remove}(\text{pcs}, \mathcal{V})\underline{\geq}$

$$\begin{aligned} \mathit{undefined}(\texttt{DomainName}, \mathcal{V}) &\quad \text{iff} \quad \mathit{undefined}(\texttt{DomainName}) \\ \mathit{undefined}(\texttt{SynName}, \mathcal{V}) &\quad \text{iff} \quad \mathit{undefined}(\texttt{SynName}) \\ \mathit{undefined}(\texttt{ProgramName}, \mathcal{V}) &\quad \text{iff} \quad \mathit{undefined}(\texttt{ProgramName}) \\ \mathit{undefined}(\texttt{Keyword}, \mathcal{V}) &\quad \text{iff} \quad \mathit{undefined}(\texttt{Keyword}) \end{aligned}$$

$$
\begin{array}{rcl}
\mathit{undefined}(\texttt{Literal}, \mathcal{V}) & \text{iff} & \texttt{false} \\
\mathit{undefined}(\texttt{undefined}, \mathcal{V}) & \text{iff} & \texttt{false} \\
\mathit{undefined}(\leq \text{pcs} \geq, \mathcal{V}) & \text{iff} & \mathit{undefined}(\text{pcs}, \mathcal{V})
\end{array}
$$

5 Consistency of SDL Profiles

A set of SDL profiles is called *consistent*, if any specification that can be stated in these profiles behaves exactly the same way in each profile. Deriving the profiles from a common language definition enables us to make statements about consistency, because, unlike profiles defined independently, the derived profiles share many common parts.

A run of an ASM is a sequence of states, where each subsequent state is the result of firing all rules whose conditions are true 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. Two SDL profiles are considered consistent, if they yield the same set of runs of their respective ASMs for all specifications contained in both profiles.

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 heuristics in some parts. However, based on these operations, it is possible to derive proof obligations that have to be verified in order to prove consistency.

E.g., 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[1]. 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 cannot be reached at all.

ASM-Listing 1 shows a part of the formal language definition that was removed as part of the save feature of SDL. 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 any reachable state of the ASM. An element cannot 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.

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

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

```
4 endif
```

ASM-Listing 1: Removed Part of Semantics Definition

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 cannot 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 cannot 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, e.g. 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. E.g., 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.

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

Relational Operators Relational Operators like $<, >, \leq, \geq$ are undefined if one of their subexpressions is undefined. To prove consistency, in this case we have to prove that the expression cannot 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.

6 SDL-Profile Tool

Based on the formalisation provided in Section 4, we have implemented a tool called SDL-profile tool in order to automate the reduction process, providing visible results. The tool reads the formal semantics definition, 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), possibly defining default values, e.g. for predicates. Figure 6.1 shows the sequence of steps performed during the removal, and the tools used for each step.

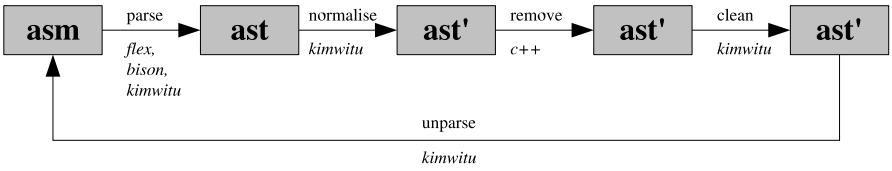

Figure 6.1: Tool Chain of the SDL-Profile Tool

6.1 Tool Chain

Parser

The *parser* takes an ASM specification as input and creates an abstract syntax tree representation of the specification as output. It is generated out of specifications of the lexis, grammar and abstract syntax of Abstract State Machines, as used in the formal semantics of SDL-2000 [3]. The specification of the abstract syntax is translated by kimwitu++ [12] to a data structure for the abstract syntax tree, using C++ classes. Scanner and parser are generated by flex and bison, respectively. Apart from minor differences, the parser is identical to the parser used in [16].

Normalisation

The *normalisation* 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 make the definition of the remove function easier.

Remove

The *remove* step is the implementation of the removal formalised in Section 4. For each type of node (*phylum*) in the abstract syntax definition, a remove function is introduced. The remove function performs removal for each term of the respective phylum, e.g. the terms `IfThenElse`, `Choose`, and `Extend` for the *rule* phylum. It returns a term of the respective phylum as result – e.g. the remove function for rules always returns a term of type rule.

For a term of a phylum, removal starts by checking conditions consisting of the predicates *true*, *false* and *undefined*, as defined in the formalisation of the reduction process. If a condition evaluates to `true`, a modified term is returned, calling remove recursively on the subterms of the term if necessary. E.g., for the rule term `IfThenElse`, if the predicate *true* holds for expression *exp*, removal continues with the **then**-part, if the predicate *false* holds for expression *exp*, removal continues with the **else**-part. If *undefined* holds for the expression *exp*, the rule term `Skip` is returned.

```
IfThenElse(exp, r1, r2):  {
   if (eval_true(exp,V)) { return remove(r1,V); };
   if (eval_false(exp,V)) { return remove(r2,V); };
   if (eval_undef(exp,V)) { return Skip(); };
   return IfThenElse(remove(exp,V), remove(r1,V), remove(r2,V));
}
```

Cleanup

The *cleanup* step transforms superfluous rules resulting from the removal 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 cleanup step only removes trivial parts of the ASM specification. The resulting specification is semantically equivalent to the specification before the cleanup step.

Iteration

Given a completely defined reduction profile, only one run of the SDL-profile tool is needed to generate a reduced ASM 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 removal process. E.g., a function in the ASM signature with a target domain that has been removed during the previous removal step, is included in the reduction profile of a subsequent iteration.

Unparsing

Unparsing traverses the abstract syntax tree and outputs a string representation of every node. The result is a textual representation of the formal semantics 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, e.g. a Latex document of the formal semantics, or a compilation to C++.

6.2 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 removal process, we compare the original semantics definition with the reduced version. For this, we have used graphical diff-based tools (e.g., tkdiff) to highlight the differences between the versions. Using the SDL-profile tool, we have created reduction profiles for several language modules, such as timers, exceptions, save, composite states and inheritance. We have also created reduction profiles for language profiles like SAFIRE and *Core*, resulting in formal semantics definitions that, with small modifications, match these SDL profiles.

ASM Listings 1 and 2 show an excerpt of the formal semantics definition before and after applying the SDL-profile tool, using a reduction profile for SDL exceptions. 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
```

ASM-Listing 1: Macro SELECTTRANSITIONSTARTPHASE before Reduction

```
1 SELECTTRANSITIONSTARTPHASE ≡
2   if (Self.currentStartNodes ≠ ∅) then
3     ...
4   else
5     ...
6   endif
```

ASM-Listing 2: Macro SELECTTRANSITIONSTARTPHASE after Reduction

7 Related Work

A modular language definition as described in this paper can be found in the language specification of UML [15, 14]. 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. However, each meta-model class/language element has only an informal description of its semantics.

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. E.g., the profile mechanism has been used to define a UML profile for SDL, enabling the use of UML 2.0 as a front-end for SDL-2000 [11].

ConTraST [1] 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.

In [13], 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. While the complexity of the algorithm is acceptable in the average case, the worst case complexity is exponential.

8 Conclusions and Outlook

In this paper, we have introduced the concept of SDL profiles as well-defined subsets of SDL, leading to smaller, more understandable language definitions. Tool support can be based on these profiles, leading to faster tool development and less expensive tools. Based on the smaller language definitions, code optimisations can be performed when generating code from a specification.

We have argued for the importance of formal semantics for language definitions, and the importance of deriving the formal semantics of SDL profiles from a common formal semantics definition. This allows us to compare the formal semantics of different SDL profiles, and to make assertions about their consistency.

To achieve deterministic results, we have formalised the extraction of the formal semantics for SDL profiles from the complete formal semantics of SDL-2000. 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. The extraction has been automated by the SDL-profile tool, providing visible results. This tool has been used to create several language profiles for SDL-2000, by removing SDL language modules from the formal semantics definition, such as e.g. exceptions, timers, save and composite states. The reduction achieved is significant. E.g., the formal semantics definition for Safire has been reduced to less than 2300 lines of specification, and less than 1100 lines for a small core of SDL, from about 3700 lines of the complete formal semantics.

Based on the formally defined process for the derivation of formal language definitions for SDL profiles, we can define precise criteria for the consistency of SDL profiles. Currently, some consistency criteria have to be verified manually. Our future work will focus on improving the extraction process, so that further criteria can be checked automatically.

Bibliography

[1] Fliege, Ingmar ; Grammes, Rüdiger ; Weber, Christian: ConTraST - A Configurable SDL Transpiler And Runtime Environment. In: Gotzhein, Reinhard (Hrsg.): *SAM'06 - Fifth Workshop on System Analysis and Modelling, Kaiserslautern, Germany*, 2006. – to be published

[2] 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

[3] Glässer, Uwe ; Gotzhein, Reinhard ; Prinz, Andreas: An Introduction To Abstract State Machines / Department of Computer Science, University of Kaiserslautern. 2003 (326/03). – Forschungsbericht

[4] Gurevich, Yuri: Evolving Algebras 1993: Lipari Guide. In: Börger, Egon (Hrsg.): *Specification and Validation Methods*. Oxford University Press, 1995, S. 9–36

[5] Gurevich, Yuri: May 1997 Draft of the ASM Guide / EECS Department, University of Michigan. 1997 (CSE-TR-336-97). – Forschungsbericht

[6] ITU: *Recommendation Z.100 (08/02): Specification and Description Language (SDL)*. Geneva, 2002

[7] ITU: *Recommendation Z.100 (2002) Amendment 1 (10/03): Specification and Description Language (SDL)*. Geneva, 2003

[8] ITU: *Recommendation Z.100 (2002) Corrigendum 1 (08/04): Specification and Description Language (SDL)*. Geneva, 2004

[9] ITU Study Group 10: *Draft Z.100 Annex F2 (11/00)*. 2000

[10] ITU Study Group 10: *Draft Z.100 Annex F3 (11/00)*. 2000

[11] ITU Study Group 17: *UML Profile for SDL*. 2005. – Draft Recommendation Z.109

[12] 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

[13] 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

[14] Object Management Group: *Unified Modeling Language Specification, Version 1.3.* 2000. – www.uml.org

[15] Object Management Group: *Unified Modeling Language: Superstructure, Version 2.0.* 2005. – www.uml.org

[16] 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

[17] SDL Task Force: *SDL+ - The Simplest, Useful 'Enhanced SDL-Subset' for the Implementation and Testing of State Machines.* 2005. – www.sdltaskforce.org