SDL Profiles – Formal Semantics and Tool Support

R. Grammes and R. Gotzhein

Computer Science Department, University of Kaiserslautern

Abstract. 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 paper, we present a formalised approach for extracting the formal semantics for SDL profiles from the complete SDL semantics. Based on this formalisation, we then define a notion of profile consistency. Finally, we 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) [1] 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 are targeted towards specific domains and companies, where, due to their reduced complexity, they are preferred by engineers. Following the notion of UML profiles [2], which enable the specialisation of UML for specific domains, we call these subsets *SDL profiles*.

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 from keeping them consistent. In this paper, 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.

This paper is organised as follows: Section 2 gives a brief overview over 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, it is shown how the approach has been formalised. Consistency of SDL profiles is defined 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

In this section, we survey the definition of SDL, and briefly present ASMs, the formalism used to define the dynamic semantics of SDL.

2.1 Specification and Description Language (SDL)

The Specification and Description Language (SDL) [1] 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 (see Section 2.2), was defined from scratch [3]. 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 [1]. It covers all static and dynamic language aspects, and consists of two major parts (for a detailed survey, see [3]):

– 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), 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 extended 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. The dynamic semantics contains over 3700 lines of ASM specification.

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* (or vocabulary) containing domain names, function names and relation names, together with an interpretation of these names over a base set. Intuitively, it 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 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.

3 Outline of the Extraction Approach for SDL Profiles

SDL profiles are self-contained subsets of SDL, targeted towards specific domains and companies. In comparison to the full SDL language definition, SDL profiles have reduced complexity, a result that is useful for both engineers and tool builders. In [6], the SDL Task Force has identified an SDL profile called SDL+, which focuses on the state machine aspect of SDL and adds functionality for testing[1]. However, no formal semantics is provided for SDL+. In [7], we have identified a hierarchy of SDL profiles, some of which are supported by our configurable transpiler ConTraST [8].

An SDL profile, as a subset of the complete SDL specification, can be characterised by its reduced concrete and abstract syntax, by defining a reduced language grammar. For the reduced language grammar, well-formedness conditions and transformations have to be taken into account. Some well-formedness conditions and transformations become dispensable. A problem arises when, through the reduction of the grammar, a well-formedness condition cannot be met, and the resulting language is empty. This occurs when language constructs are removed, while language constructs that depend on them remain in the subset.

[1] For SDL profiles, we only regard the parts of SDL+ that are included in SDL.

Likewise, transformations are affected if a language construct in the subset is transformed to a language construct that has been removed. These problems are avoided by taking self-contained subsets of the language, like the SDL subsets implicitly defined by tool providers.

We define SDL profiles by *extracting* the profile definition from the complete SDL language definition. To obtain a particular SDL profile, we remove parts correspondig to language features not included in the profile from the formal syntax and semantics. While the syntax extraction for a given SDL profile is straightforward, the extraction of the formal semantics has turned out to be difficult. To solve this problem, 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 dead ASM 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 ASM rule recognition is practically feasible. Therefore, we have followed this approach, and will present safe criteria as well as some heuristics below.

4 Formalisation

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.

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 SDL Virtual Machine (SVM) data structure (ASM signature, see Figure 1). 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 (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.

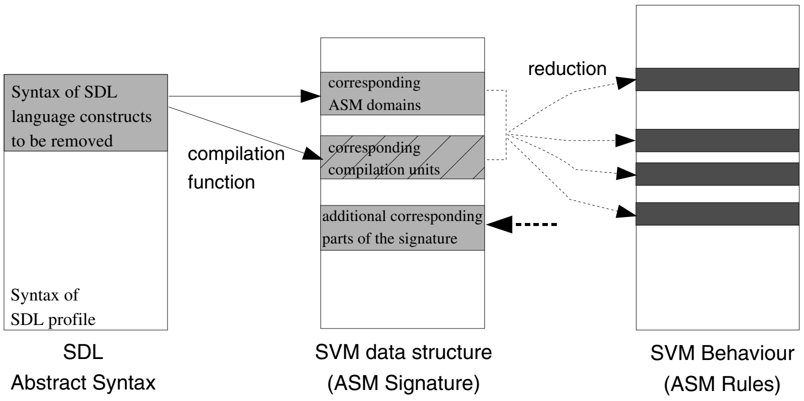
Syntax of SDL language constructs to be removed
Syntax of SDL profile
SDL Abstract Syntax
compilation function
corresponding ASM domains
corresponding compilation units
additional corresponding parts of the signature
SVM data structure (ASM Signature)
reduction
SVM Behaviour (ASM Rules)

Fig. 1. Concept of the extraction process

Featuring 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 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. For example, *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.

In order to perform dead ASM rule recognition, we collect all parts of the ASM signature that correspond to language modules not included in the SDL profile in a *reduction profile*. 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 the compilation function, however, domain knowledge is still required. We specify a default value (`true` or `false`) for predicates, and assume the special element `undefined` and the empty set as default values for functions and domains. These elements are removed from the formal semantics definition according to a set of extraction rules formally defined in the following sections. The complete formalisation can be found in [9].

Save-signalset
SignalSaved = false

Fig. 2. Reduction profile for 'save' feature

Figure 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

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 [10]. The input of the reduction is the SDL formal semantics definition from [11] 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 $\mathit{undefined}_r$, 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.

$$\begin{aligned} \mathit{remove} &: G \times V \to G \\ \mathit{undefined}_r &: G \times V \to \mathsf{Boolean} \\ \mathit{true}_r &: G \times V \to \mathsf{Boolean} \\ \mathit{false}_r &: G \times V \to \mathsf{Boolean} \end{aligned}$$

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 $\mathit{undefined}_r$ 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 Definition of *true* and *false*

Identifying expressions as always true (false) is an important step in the reduction process. Predicates *true* and *false* hold for some first-order expressions that are true (false) in every state of the ASM. Generally, this is undecidable for first-order expressions.

Predicates of the ASM can be included in the reduction profile together with a default value `true` or `false`. Boolean-valued functions *true* and *false* introduced in Section 4.2 hold directly for these predicates. For general first-order expressions, we define the functions *true* and *false* recursively.

Predicate *undefined* holds for expressions that evaluate to the empty set or the special element `undefined` in every state, as defined by the reduction profile. Therefore, we can derive that equating an expression e with the empty set or the special ASM element `undefined` always yields `true` if *undefined* holds for expression e. Likewise, such an expression is never unequal to the empty set or `undefined`. With an expression e_2 being an empty set, we can determine the element-of operator to yield `false` in every state of the ASM. These considerations are reflected in the following definitions:

$$\begin{aligned} \mathit{true}(e = \mathtt{undefined}, \mathcal{V}) &\text{ iff } \mathit{undefined}(e, \mathcal{V}) \\ \mathit{false}(e \neq \mathtt{undefined}, \mathcal{V}) &\text{ iff } \mathit{undefined}(e, \mathcal{V}) \\ \mathit{true}(e = \emptyset, \mathcal{V}) &\text{ iff } \mathit{undefined}(e, \mathcal{V}) \\ \mathit{false}(e \neq \emptyset, \mathcal{V}) &\text{ iff } \mathit{undefined}(e, \mathcal{V}) \\ \mathit{false}(e_1 \in e_2, \mathcal{V}) &\text{ iff } \mathit{undefined}(e_2, \mathcal{V}) \end{aligned}$$

In the SVM, a state s is a tuple with several elements. An element is selected from the tuple using the **s-** operator together with the element type, for example

s.s-*Save-signalset* for the set of saved signals in this state. Given the reduction profile in Figure 2, *false* holds for the expression $sig \in s$.s-*Save-signalset* for all signals *sig* and all SDL states s.

The result of *true* and *false* for a boolean expression is derived from its subexpressions, and can be defined in a truth table. Table 1 defines if *true* (T), *false* (F), *undefined* (U) or none of these predicates (-) hold for the disjunction.

Table 1. Truth table for disjunction

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

Further boolean connectors and quantified expressions are defined in a similar fashion in [9].

4.4 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. Below, we show the formalisation of the reduction for two rules.

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[2]. 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.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{remove}(\textbf{if } \text{exp } \textbf{then } R_1 \textbf{ else } R_2 \textbf{ endif}, \mathcal{V}) =} \\
\quad \mathit{remove}(R_1, \mathcal{V}) & \text{iff} & \mathit{true}(\text{exp}, \mathcal{V}) \\
\quad \mathit{remove}(R_2, \mathcal{V}) & \text{iff} & \mathit{false}(\text{exp}, \mathcal{V}) \\
\quad \text{skip} & \text{iff} & \mathit{undefined}(\text{exp}, \mathcal{V}) \\
\quad \textbf{if } \mathit{remove}(\text{exp}, \mathcal{V}) \textbf{ then } \mathit{remove}(R_1, \mathcal{V}) & & \text{otherwise} \\
\qquad \textbf{else } \mathit{remove}(R_2, \mathcal{V}) \textbf{ endif} & &
\end{array}
$$

Figure 3 shows an **if**-rule with a guard expression for which *false* holds, given the reduction profile in Figure 2. The **if**-rule is removed except for the else-branch, which is empty.

[2] 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.

```
if $sig \in s.\mathrm{s}{-}Save{-}signalset$ then
    ...
endif
```

Fig. 3. **if**-Rule with unsatisfiable guard condition

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.If *true* or *undefined* hold for the constraint, the **choose**-rule is invalid since it ranges over a potentially infinite set.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{remove}(\textbf{choose}\ x : \mathrm{exp}\ R\ \textbf{endchoose}, \mathcal{V}) =} \\
\quad \mathrm{skip} & \text{iff} & \mathit{false}(\mathrm{exp}, \mathcal{V}) \lor \mathit{true}(\mathrm{exp}, \mathcal{V}) \lor \\
 & & \mathit{undefined}(\mathrm{exp}, \mathcal{V}) \\
\quad \textbf{choose}\ x : \mathit{remove}(\mathrm{exp})\ \mathit{remove}(R, \mathcal{V}) & & \text{otherwise} \\
\quad \textbf{endchoose} & &
\end{array}
$$

5 Consistency of SDL Profiles

We call a set of SDL profiles *consistent*, if any specification that can be stated in all of 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 which 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.

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[3]. 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.

[3] 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.

Figure 4 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 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.

```
if $Self.signalChecked.signalType \in$
      $sn.stateAS1.\mathrm{s}{-}Save{-}signalset.\mathrm{s}{-}Signal{-}identifier{-}\mathbf{set}$ then
   $Self.SignalSaved := True$
endif
```

Fig. 4. Removed part of formal 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 can not be reached.

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.

$$true(e_1 \wedge e_2) \text{ iff } true(e_1) \textbf{ and } true(e_2) \tag{1}$$

$$false(e_1 \wedge e_2) \text{ iff } false(e_2) \textbf{ or } false(e_2) \tag{2}$$

$$true(e_1 \vee e_2) \text{ iff } true(e_1) \textbf{ or } true(e_2) \tag{3}$$

$$false(e_1 \vee e_2) \text{ iff } false(e_1) \textbf{ and } false(e_2) \tag{4}$$

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 section. In order to prove consistency, we show that all generated proof obligations hold. Currently, this verification is done manually.

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. Figure 5 shows the sequence of steps performed during the removal, and the tools used for each step.

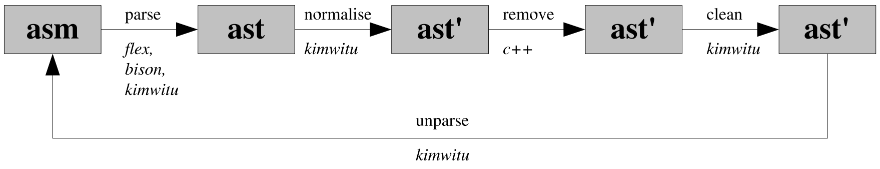

Fig. 5. Tool chain of the SDL-profile tool

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 [10]. 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 [13].

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 (called *phylum*) in the abstract syntax definition, a remove function is introduced. The remove function performs removal for each term of the respective phylum, for example the terms `IfThenElse`, `Choose`, and `Extend` for the *rule* phylum. It returns a term of the respective phylum as result – for example 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. For example, 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.

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, for example 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 (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 SDL+ and Core, resulting in formal semantics definitions that, with small modifications, match these SDL profiles.

```
SELECTTRANSITIONSTARTPHASE ≡
  if ($Self.currentExceptionInst \neq undefined$) then
    $Self.agentMode3 := selectException$
    $Self.agentMode4 := startPhase$
  elseif ($Self.currentStartNodes \neq \emptyset$) then
    ...
  else
    ...
  endif

SELECTTRANSITIONSTARTPHASE ≡
  if ($Self.currentStartNodes \neq \emptyset$) then
    ...
  else
    ...
  endif
```

Fig. 6. Macro SELECTTRANSITIONSTARTPHASE before and after reduction

Figure 6 shows 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.

7 Related Work

A modular language definition can be found in the language specification of UML [2]. 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 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 to define a UML profile for SDL, enabling the use of UML 2.0 as a front-end for SDL-2000 [14].

ConTraST [8] 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 SDL-profile tool.

In [15], 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. ASM slicing does not cover indeterminism, which usually occurs in language definitions.

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. Deriving the formal semantics of SDL profiles from a common formal semantics definition 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 exceptions, timers, save and composite states. The reduction achieved is significant. The formal semantics definition for SDL+ 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.

References

1. ITU Recommendation Z.100: Specification and Description Language. Geneva (1999)
2. OMG Unified Modelling Language Specification: Version 2.0 (2003) www.omg.org.
3. Glässer, U., Gotzhein, R., Prinz, A.: The Formal Semantics of SDL-2000 - Status and Perspectives. Computer Networks, Elsevier Sciences **42** (2003) pp. 343–358
4. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36

5. Gurevich, Y.: May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan (1997)
6. SDL Task Force: SDL+ - The Simplest, Useful 'Enhanced SDL-Subset' for the Implementation and Testing of State Machines (2005) www.sdltaskforce.org.
7. Grammes, R.: Formal Operations for SDL Language Profiles. In Gotzhein, R., Reed, R., eds.: SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany. Volume 4320 of LNCS., Springer (2006) pp.51–65
8. Fliege, I., Grammes, R., Weber, C.: ConTraST - A Configurable SDL Transpiler And Runtime Environment. In Gotzhein, R., Reed, R., eds.: SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany. Volume 4320 of LNCS., Springer (2006) pp.222–234
9. Grammes, R., Gotzhein, R.: SDL Profiles - Definition and Formal Extraction. Technical Report 350/06, Department of Computer Science, University of Kaiserslautern (2006)
10. Glässer, U., Gotzhein, R., Prinz, A.: An Introduction To Abstract State Machines. Technical Report 326/03, Department of Computer Science, University of Kaiserslautern (2003)
11. ITU Study Group 10: Draft Z.100 Annex F3 (11/00) (2000)
12. von Löwis, M., Piefel, M.: The Term Processor Kimwitu++. In Callaos, N., Harnandez-Encinas, L., Yetim, F., eds.: SCI 2002: The 6th World Multiconference on Systemics, Cybernetics and Informatics, Orlando, USA (2002)
13. Prinz, A., von Löwis, M.: Generating a Compiler for SDL from the Formal Language Definition. In Reed, R., Reed, J., eds.: SDL 2003: System Design. Volume 2708 of LNCS., Springer (2003) pp. 150–165
14. ITU Study Group 17: UML Profile for SDL. Draft Recommendation Z.109 (2005)
15. Nowack, A.: Slicing Abstract State Machines. In Zimmermann, W., Thalheim, B., eds.: Abstract State Machines 2004. Advances in Theory and Practice, Lutherstadt Wittenberg, Germany. Volume 3052 of LNCS., Springer (2004) 186–201