Formal Operations for SDL Language Profiles

Rüdiger Grammes

Computer Science Department, University of Kaiserslautern
Postfach 3049, D-67653 Kaiserslautern, Germany
grammes@informatik.uni-kl.de

Abstract. Expressive system modelling languages lead to language definitions that are long and hard to understand. Tool support for these languages is hard to implement, and often only parts of the language are supported. In this paper we introduce the concept of language profiles as well-defined subsets of a language with formal syntax and semantics as the basis for tool support. We outline two approaches to generate language profiles for SDL from the complete formal semantics definition, and provide a formalisation for a reduction-based approach, on which a tool for this approach is based.

1 Introduction

In order to support a wide range of applications, system modelling languages are often complex and expressive. The complexity of the languages leads to language definitions that are long and hard to understand, and can limit their applicability in domains for which specialised, tailor-made languages are preferred. Another drawback is that tool support for complex languages usually covers only parts of the language. For example, there is no tool that supports the whole of SDL-96 [1,2], and only a few of the language constructs introduced in SDL-2000 [3,4,5] are supported.

Language profiles divide a language into a core language and a set of language modules that can be used as language building blocks. The language core represents a minimal subset of the language that a tool for the language should implement. This core is a profile that can be extended by language modules, yielding further language profiles that represent well-defined subsets of the language which a tool provider can implement. Thus, using language profiles it is possible to define sublanguages of a language that are of lesser complexity and are tailor-made for certain application areas.

Formal semantics gives a precise definition of the language and eliminate the ambiguities that come with an informal language definition. Operational mathematical formalisms like Abstract State Machines [6,7] can be executed and used to generate a compiler and runtime system [8], giving a reference for tool developers. Defining language profiles, we focus on the formal semantics of the language. Formal semantics allow us to formulate precise criteria for valid language extension and reduction. SDL-2000 [3] is a language with a complete formal semantics [9,10,11], defined using ASMs, which makes it well-suited for the definition of language profiles.

R. Gotzhein and R. Reed (Eds.): SAM 2006, LNCS 4320, pp. 49–63, 2006.

In this paper, we introduce language profiles of SDL (section 2). We define a process for the generation of language profiles for SDL from a formal semantics defined with Abstract State Machines. This process is based on the reduction of the semantics by formally defined operations (section 3), and formalised and implemented in an SDL-profile tool (section 4).

2 Language Profiles and Modules

2.1 Problem and Definition

SDL has become a sophisticated and complex language with many language features. SDL-2000, the most recent version, has added several new language constructs, for example composite states, exceptions, agents (a harmonisation of the concepts of systems, blocks and processes) and textual notation of algorithms. 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 subset of SDL [12]. 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.

A sublanguage like SAFIRE is a language profile. Tools for a language profile can be developed faster, leading to less expensive tools and enabling code optimisations. Possible language 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, language profiles can be subsets of other language profiles, forming a hierarchy profiles. For SDL, we have defined four language 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 1.

A *language module* encapsulates a language feature, defining its syntax, semantics and dependencies to other language modules. Some language modules of SDL are timers, exceptions, save, and inheritance. Figure 2 shows the (graphical) syntax elements of the timer feature, ASM-Listing 1 parts of the formal semantics of timers. ASM macro SETTIMER describes the setting of a timer by inserting a new timer instance into the schedule of the process. If a time t is

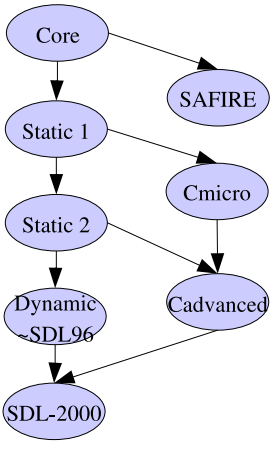

Fig. 1. Superset Relationship between Language Profiles

given, the arrival is set to this time, otherwise the arrival is computed from the current time and the standard duration defined for the timer. Signals in the schedule are sorted by time of arrival. They are invisible to the process until the current time is equal or greater than their time of arrival.

```
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 1. Setting SDL Timers

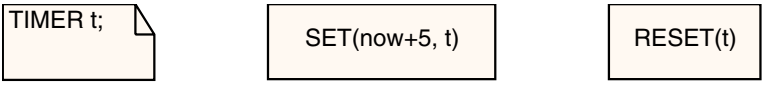

Fig. 2. Syntactical Elements of the Timer Module

2.2 Approach for the Generation of Language Profiles

SDL-2000 is a language with formal semantics, and this property should be retained for its sublanguages. However, it is not feasible to define a new formal

semantics from scratch for every sublanguage, since it requires substantial effort and can lead to inconsistencies between the language profiles. A sensible approach is to take the existing formal language definition, and to systematically modify it to match a subset of the language. In principle, there are two ways to achieve this goal:

- *bottom-up:* Given a modular structure of the formal language definition, i.e. consisting of a core language and a hierarchy of language modules that can be added to the core, the formal language definition for the language profile is obtained by constructing it from the core and the modules corresponding to the features contained in the language profile.
- *top-down:* Starting from the complete formal language definition, we remove all parts that correspond to features not contained in the subset of the language.

The bottom-up approach requires a modular language definition with a small core language, language features encapsulated in language modules, and a way to compose the language modules with the core and other modules, both syntactically and semantically. Feature interaction plays a crucial role with the bottom-up approach, as language features like exceptions may interact with other language features. This affects the order in which the language modules are composed. Another problem of the bottom-up approach is that it is very difficult to encapsulate the formal semantics of a language module in a way that it can be easily composed with a given language profile, while at the same time maintaining readability of the formal semantics. For these reasons, we are choosing the top-down approach.

2.3 Consistency of Language Profiles

The goal is for a specification defined with a language profile to behave in the same way with all supersets of the language profile. In order to accomplish this goal, we need to assure *consistency* between the language profiles. Deriving the profiles from a common language definition enables us to make statements about consistency, because, unlike profiles defined from scratch, the derived profiles share many common parts. With the bottom-up approach, we need to ensure that adding modules does not interfere with existing specifications. With the top-down approach, only parts of the language definition that do not apply to features contained in the subset may be removed (that is, parts of the ASM formalism that are not reached in the subset).

2.4 Derivation of Language Profiles with the Top-Down Approach

Reduction of the formal language definition consists of reduction of the formal syntax and reduction of the formal semantics. The formal syntax is reduced by deleting all syntax elements corresponding to features to be deleted from both the concrete and abstract syntax. In order to remove a feature from the formal semantics of SDL-2000, we start by identifying domains and functions from the signature of the formal semantics definition. The signature consists of names of domains, functions and relations of the ASM. We identify the parts of

the signature that correspond to the feature to be removed. Several domains in the formal semantics can be identified that correspond to a particular feature, for example the domains TIMER and TIMERINST are used to specify the timer feature of SDL. Furthermore, for each non-terminal in the abstract syntax, there is a domain in the formal semantics definition. As the abstract grammar is reduced, the respective domains can be removed, too.

```
1 TIMER =def Identifier
2 TIMERINST =def PID ×TIMER ×VALUE∗
3 SET =def TIMELABEL ×TIMER ×VALUELABEL ×CONTINUELABEL
4 RESET =def TIMER ×VALUELABEL ×CONTINUELABEL
```

ASM-Listing 2. Domains Corresponding to the TIMER Feature

Reduction of the signature of the formal semantics definition affects the ASM-rules of the definition, which have to be reduced accordingly. All occurrences of removed functions and domains must be removed from the definition. This leads to the removal of entire rule blocks, for example when the guard of an if-rule has to be removed. The rules should be reduced as much as possible, in order to get a concise formal semantics definition without any remaining parts of the removed features. On the other hand, care must be taken that the removal only affects language constructs that should be removed and no other language constructs are affected. In cases where this is not possible, there is very likely a feature interaction, which is either inherent to the language or was introduced in the formal semantics. For example, procedures and composite states share common parts in the formal semantics of SDL-2000, because their underlying concepts are very similar.

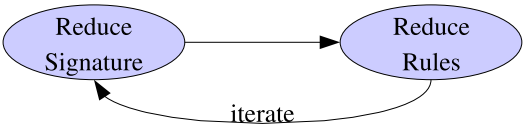

Fig. 3. Process of Feature Removal

A way to approach the removal of rules is to assign fixed default values to the functions and domains to be removed, and then to remove unreachable parts of the formal semantics accordingly. Possible default values for domains would be the empty set, for partial functions the special ASM element `undefined`, for boolean functions (predicates) it would be either `true` or `false`. For example, the default value for the predicate *Spontaneous* would be `false`, so that the triggering of a spontaneous transition during transition selection would never occur, disabling spontaneous transitions entirely. Listing 3 shows the rule fragment that defines how spontaneous transitions are triggered. Since the fixed default value `false` is assigned to *Spontaneous*, the entire **elseif**-block of the rule fragment can be removed. Consistency is guaranteed for specifications that do not use spontaneous transitions, since it can be proven that mode *selectSpontaneous* has

no effect in this case. That is, no updates are fired in this mode, except for updates that set the agent mode functions to the previous mode.

```
1 monitored Spontaneous: AGENT →BOOLEAN (default False)
2
3   if Self.stateNodeChecked = undefined then
4     NEXTSTATENODETOBECHECKED
5   elseif Self.Spontaneous then
6     Self.agentMode4 := selectSpontaneous
7   else
8     ...
9   endif
```

ASM-Listing 3. Triggering Spontaneous Transitions

Assigning the default value `false` to *Spontaneous* disables spontaneous transitions, however, unreachable parts of the formal semantics of spontaneous transitions still remain in the formal semantics definition. In order to remove them, a further reduction of the formal semantics definition is necessary. This reduction includes, for example, guarded rule fragments that check for the agent mode *selectSpontaneous*.

Table 1. Definition Size for Profiles

Profile	Features	Lines of Spec.
Core	System, Block, Process, Channel Simple Statemachines	1500 lines
$Static_1$	*Core* +Timer, +Actions, +Data, ...	1900 lines
$Static_2$	$Static_1$ +Services, +Inheritance, +Data +Priority Input, +Continuous Signal, ...	2240 lines
SAFIRE		2280 lines
Dynamic ∼ SDL 96	$Static_2$ +Procedures, +Dynamic Process Creation	2570 lines
SDL-2000		3130 lines

Table 1 shows the size of the reduced dynamic part of the formal semantics of SDL for several language profiles of SDL-2000. *Core*, $Static_1$, $Static_2$ and *Dynamic* build a hierarchy of language profiles, starting from *Core* with minimal features and going up to the dynamic subset, which roughly equals SDL'96. The formal semantics of SAFIRE is slightly larger than the second static subset, though $Static_2$ contains features not covered by SAFIRE. However, SAFIRE contains procedures, which are not part of $Static_2$.

3 Formalisation

In this section, we introduce a formalisation of the process for the derivation of language profiles with the top-down approach. The formalisation gives an exact

definition of the removal process, leading to deterministic results. It provides the foundation for tool support for the removal process. Finally, a formal definition is necessary in order to make precise statements about the consistency of language profiles. Since the formal syntax definition can be easily defined in a modular fashion, making reduction of the syntax straightforward, we focus on the reduction of the formal semantics definition.

The formal semantics definition consists of two parts, the static semantics and the dynamic semantics. The static semantics consists of well-formedness conditions and transformation rules. Where language modules are removed, corresponding well-formedness conditions and transformation rules have to be removed accordingly. However, in this paper we focus on the dynamic semantics of SDL.

For the formal definition of the removal process, we are looking for a mathematical formalism that is readable and easy to understand. Therefore, 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.

3.1 Formalisation Signature

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*, *true* and *false*, that control the reduction. The profile definition is given as a globally defined set of elements r from the signature of the formal semantics definition, annotated by default values `true` and `false` for predicates. This set represents the elements to be removed from the formal semantics definition, and is therefore called the *reduction profile*. For all elements in the reduction profile, *undefined* (*true* or *false* for predicates) holds.

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

The *remove* function is defined on all elements of the grammar G. It is defined recursively - a given term is mapped to a new term by applying the mapping defined by *remove* to the subterms. In case the predicates *undefined*, *true* and *false* do not hold, nothing more is done. This assures that *remove* corresponds to the identical mapping if the signature of the formal semantics definition is not reduced (that is, the reduction profile is empty). In other cases, subterms can be replaced or omitted depending on which of the predicates hold.

Predicates *true* and *false* are explicitly defined on boolean and first-order logic expressions. On all other elements of G, the predicates do not hold. Predicate $true(e, v)$ ($false(e, v)$) holds only if expression e always evaluates to `true` (`false`) in any state of the ASM with reduced signature. These predicates are determined using formal criteria and heuristics.

Predicate *undefined* is defined on all expressions and domains. It holds on any expression or domain that can not be reduced to a defined expression/domain. A defined expression or domain contains only elements that are not in the reduction profile r. For example, if *undefined* holds for expression e_1 and expression e_2, *undefined* also holds for expression $e_1 \vee e_2$.

3.2 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 representative rules.

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

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

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

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

[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.

3.3 Formal Reduction of ASM Expressions

Expressions are terms over the signature of the formal semantics definition. Additionally, ASMs include common mathematical structures like boolean algebra, or natural numbers. Our formal reduction covers all operations defined in [13]. Below is an excerpt of the formal reduction of ASM expressions, covering boolean and relational 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 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.

Table 2. Truth Table for Negation

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

Table 3. Truth Table for Disjunction

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

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

We define truth tables for all boolean operators from the concrete syntax of ASMs: negation ($\neg$, see Table 2), disjunction ($\vee$, see Table 3), conjunction ($\wedge$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). In order to ensure consistent results, we derive the definition of conjunction, implication and equivalence from the definitions of negation and disjunction.

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. It is important as it often appears in the guard of **if**-statements. 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.

In the same way as with the examples given above, the function *remove* is formally defined for all elements of the concrete grammar of ASMs, and

the predicates *true*, *false* and *undefined* are formally defined for the elements of the grammar for which they apply. This gives us a complete formalisation of the reduction process.

4 SDL-Profile Tool

Based on the formalisation provided in section 3, we have implemented an SDL-profile tool in order to validate 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. Figure 4 shows the sequence of steps performed during the removal, and the tools used for each step.

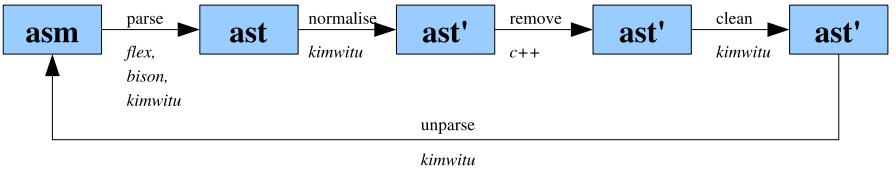

Fig. 4. Toolchain of the SDL-Profiling Tool

4.1 Toolchain

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 definitions of the lexis, grammar and abstract syntax of Abstract State Machines, as used in the formal semantics of SDL-2000. The definition of the abstract syntax is translated by kimwitu++ [14] 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 [8].

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. For example, during the normalisation step, **extend**-statements containing a list of variables to be bound to new elements in a domain are rewritten. The result is a set of nested **extend**-statements containing only one variable each.

```
Extend(dom, ConsnameList(nhead,nrest), rul)
-> < normal: ExtendSingle(dom, nhead, Extend(dom, nrest, rul)) >;
```

Remove. The *remove* step is the implementation of the removal formalised in section 3. For each type of node (called *phyla* in kimwitu) 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 `Assign`, `Choose`, `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.

The remove functions for phyla follow a pattern. Formal arguments of the function are a phylum and a set of casestrings (the locally undefined ASM names). The return type is the same as the phylum used as formal argument, ensuring the resulting term has the correct type in the context in which it occurs. The outermost statement is a switch over all terms of the phylum, using the kimwitu control structure *with*. For each term, the actions for removal are defined separately.

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 removal 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. Removal starts at the root of the abstract syntax tree and works towards the leaves, without any backtracking. Therefore, removal on a subtree does not take the context of the subtree into account. However, the removal can affect the context and make it obsolete. If the entire rule body of an **extend**-statement is reduced to skip, the **extend**-statement itself could be removed. 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.

Cleanup performs the following modifications to the formal semantics:

- The rule skip is removed from parallel rule blocks.
- Rules with subrules are replaced with skip if all subrules of the rule are skip-rules.

- **if**-rules with identical subrules in the **then**- and **else**-part are replaced by the subrule in the **then**-part.
- All local definitions of a rule macro with a skip-rule as rule-body are removed. These definitions are not visible outside of the rule macro, and are not referenced by the rule-body.

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 formal semantics definition. In case the reduction profile is incomplete, the SDL-profile tool can identify further names in the signature that can be removed, and iterate the removal process. For example, a function 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 SDL-profile tool can be used as the input for a subsequent run of the tool. It is also possible to output the result as a latex document, for better readability. A partial compilation of ASM rules to C++ exists as a third output format. This compilation is still in an early development phase.

4.2 Results

Given a formal semantics definition in ASM 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 compared the original semantics definition with the reduced version. For this, we have used graphical diff-based tools (for example, tkdiff) to highlight the differences between the versions. Using the SDL-profile tool, we have created reduction profiles for several language features, such as timers, exceptions, save, composite states and inheritance. We have also created reduction profiles for language profiles like SAFIRE, resulting in a formal semantics definition that, with small modifications, matches that language profile.

Listings 4 and 5 show the results of applying the SDL-profile tool on the formal semantics definition for the macro SELECTTRANSITIONSTARTPAHSE, using a reduction profile for 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**-statement 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 4. Macro SelectTransitionStartPhase before Removal

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

ASM-Listing 5. Macro SelectTransitionStartPhase after Removal

5 Related Work

A modular language definition as described in this paper can be found in the language definition of UML [15]. 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. Each meta-model class/language element has a description of its semantics in an informal way.

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 don't conflict with existing semantics and constraints. For example, the profile mechanism is used to define a UML profile for SDL, enabling the use of UML 2.0 as a front-end for SDL-2000.

In [16], 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.

ConTraST [17] 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 language profiles syntactically by allowing the deactivation of language features. In particular, the language profiles *Core*, $Static_1$, $Static_2$ and *Dynamic* - as described in section 2 - are supported. In order to support language profiles

semantically, we can use the results of the formally defined derivation of language profiles from the complete formal semantics definition. Using the SDL-profile tool, the translation from the reduced formal semantics definition into a C++ runtime environment can be performed semi-automatic. The resulting runtime environment is smaller, leading to a more efficient execution.

6 Conclusions and Outlook

In this paper, we have introduced the concept of language profiles as well-defined subsets of a language, leading to smaller, more understandable language definitions. Tool support can be based on these language 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 language profiles from a common formal semantics definition. This allows us to compare the formal semantics of different language profiles, and to make assertions about the consistency of language profiles.

To achieve deterministic results, we have formalised the process of deriving formal semantics for language profiles from a complete formal semantics definition, based on Abstract State Machines and applied to the formal semantics of SDL-2000. This process is based on reducing the signature of the ASM, subsequently leading to the reduction of parts of ASM-rules that become unreachable. We have implemented this formally defined process in an SDL-profile tool, making it possible to validate the results of the reduction. This tool was used to create several language profiles for SDL-2000, by removing language features from the formal semantics definition, such as exceptions, timers, save and composite states.

Based on the formally defined process for the derivation of SDL language profiles, we can define precise criteria for the consistency of language profiles. However, currently the consistency has to be verified manually. Our future work will focus on modifying the derivation process, so that as many automatic guarantees as possible can given for the consistency of the derived profiles.

References

1. ITU Recommendation Z.100 (03/93): Specification and Description Language (SDL). Geneva (1993)
2. ITU Recommendation Z.100 Addendum 1 (10/96): Specification and Description Language (SDL). Geneva (1996)
3. ITU Recommendation Z.100 (08/02): Specification and Description Language (SDL). Geneva (2002)
4. ITU Recommendation Z.100 (2002) Corrigendum 1 (08/04): Specification and Description Language (SDL). Geneva (2004)
5. ITU Recommendation Z.100 (2002) Amendment 1 (10/03): Specification and Description Language (SDL). Geneva (2003)

6. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
7. Gurevich, Y.: May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan (1997)
8. 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
9. ITU Study Group 10: Draft Z.100 Annex F1 (11/00) (2000)
10. ITU Study Group 10: Draft Z.100 Annex F2 (11/00) (2000)
11. ITU Study Group 10: Draft Z.100 Annex F3 (11/00) (2000)
12. SDL Task Force: SDL+ - The Simplest, Useful 'Enhanced SDL-Subset' for the Implementation and Testing of State Machines (2004) www.sdltaskforce.org/sdl-tf_draftresult_4.pdf, www.sdltaskforce.org/sdl-plus_syntax.html, www.sdltaskforce.org/sdl-plus_codec.html.
13. Glässer, U., Gotzhein, R., Prinz, A.: An Introduction To Abstract State Machines. Technical Report 326/03, Department of Computer Science, University of Kaiserslautern (2003)
14. 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)
15. OMG Unified Modelling Language Specification: Version 2.0 (2003) www.uml.org.
16. Nowack, A.: Slicing Abstract State Machines. In Zimmermann, W., Thalheim, B., eds.: Abstract State Machines 2004 - Advances in Theory and Practice. Volume 3052 of LNCS., Wittenberg, Germany, Springer (2004) pp.186–201
17. Weber, C.: Entwurf und Implementierung eines konfigurierbaren SDL Transpilers für eine C++ Laufzeitumgebung. Master's thesis, University of Kaiserslautern, Germany (2005)