Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models

Suzana Andova Mark van den Brand Luc Engelen
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{S.Andova | M.G.J.v.d.Brand | L.J.P.Engelen}@tue.nl

A formal definition of the semantics of a domain-specific language (DSL) is a key prerequisite for the verification of the correctness of models specified using such a DSL and of transformations applied to these models. For this reason, we implemented a prototype of the semantics of a DSL for the specification of systems consisting of concurrent, communicating objects. Using this prototype, models specified in the DSL can be transformed to labeled transition systems (LTS). This approach of transforming models to LTSs allows us to apply existing tools for visualization and verification to models with little or no further effort. The prototype is implemented using the ASF+SDF Meta-Environment, an IDE for the algebraic specification language ASF+SDF, which offers efficient execution of the transformation as well as the ability to read models and produce LTSs without any additional pre or post processing.

1 Introduction

Domain-specific languages (DSL) and model transformations are the key concepts in model driven engineering [20]. A DSL is a language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [7]. A DSL enables domain experts to develop models using concepts in their own domain, rather than concepts provided by existing formalisms, which typically do not provide the required or correct abstractions. DSL models can be further transformed into models in other languages using model transformations. Model transformations can, for example, be used to transform DSL models into different implementations, each serving different purposes, such as validation, execution, testing, and visualization.

In our previous work, we have designed the Simple Language of Communicating Objects (SLCO) [1], which provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. The structure of a system is modeled using classes and their behavior is modeled by state machines. Simultaneously to the development of the language, we implemented a number of model transformations to other languages as well as within the SLCO language. The goal of the previous work was to generate, for a given SLCO model, (i) its implementation in NQC for execution on the Lego Mindstorms platform [3], (ii) its implementation in POOSL [21] for model simulation, and (iii) its implementation in Promela [14] for formal verification by means of the SPIN model checker [14]. All three of these languages have semantic properties different from the semantics of SLCO, and therefore, several semantic gaps needed to be bridged [2]. These semantic gaps are bridged using model transformations, that transform SLCO models into SLCO models with equivalent observable behavior.

The transformations from SLCO to NQC, POOSL, and Promela provide a partial transformational description of the semantics of SLCO, because each of these transformations only deals with a subset of SLCO. One of the goals of the work presented in this paper is to define the operational semantics of the entire language.

Francisco Durán and Vlad Rusu (Eds.):
Second International Workshop on Algebraic Methods
in Model-Based Software Engineering 2011 (AMMSE'11)
EPTCS 56, 2011, pp. 65–79, doi:10.4204/EPTCS.56.5

As for model transformations within SLCO, one way to reason about their correctness is to compare or relate the SLCO models before and after the transformation, by comparing or relating their implementations. If, in addition, the implementation language is supported by a toolset allowing for model reduction, rather larger SLCO models can be handled, either for analysis or for model comparison. As Promela and SPIN do not provide support for these features, we have been motivated to look for an alternative solution, which would impose no restrictions on the SLCO expressiveness, too.

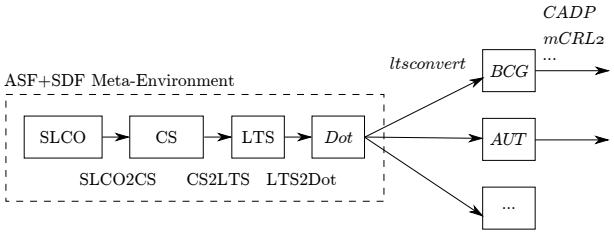


Figure 1: Overview of languages and tools

In this paper we link the SLCO language to a simple language for the representation of labeled transition systems, named simply LTS. Instead of selecting yet another target language supported by a toolset to which SLCO models will be transformed, we decided, first, to transform SLCO into LTS and, then, to link LTS to existing languages and toolsets (see Figure 1). The reason is that LTS, due to its simplicity, is much easier to link to other languages: languages that we have selected as appropriate for our purposes, such as those described in Section 4, but also other languages which may show useful in the future. In our case, we achieve this by a rather easy transformation from LTS to Dot, a language for the graphical representation of graphs that is also used by third-party tools to represent LTSs. Thus, our main effort is on translating SLCO to LTS. Once the translation into LTS has been achieved, different tools supporting LTSs are also within reach for manipulation, visualization, and verification of SLCO models. Our language transformations are defined and implemented in the ASF+SDF Meta-Environment [6]. We find its ability to generate command-line tools that can efficiently execute transformations as well as its ability to parse arbitrary context-free languages very beneficial and advantageous for our approach. As shown in Figure 1, our SLCO environment at the back-end can be connected to various already developed toolsets for various purposes.

Yet another motivation to take this approach is the following: the implementation of the transformations from SLCO models to LTSs gives a prototype of the semantics of SLCO. As shown in Figure 1, there is an intermediate representation language used, called CS, and therefore, two main transformation steps: SLCO2CS and CS2LTS. While objects in an SLCO model are defined separately and communicate via channels, in the CS representation of the same system the objects are merged into one (big) component, according to the communication as defined in the SLCO model. Thus, CS links, in a rather natural way, the two languages SLCO and LTS. The transformation from SLCO to CS essentially forms the core of the prototype semantics of SLCO, as the transformation from CS to LTS is rather straightforward. The implementation of this transformation has helped to gain a better understanding of the effect of various design decisions on the semantics of SLCO, in a larger part by the verboseness of the intermediate language CS and the visual representation of LTSs produced by external tools. Based on the conditional rewrite rules which form the core of this transformation, the formal operational semantics of the language can be defined, but this goes beyond the scope of this paper. While our environment is built around SLCO, we believe that the same methodology can be applied to other DSLs.

Structure of the paper In Section 2, SLCO is briefly described. In Section 3, the main ingredients of

the transformation environment, the languages and the tools, are introduced, and in Section 4, we link this environment to existing languages and tools. Section 5 addresses the related work, and Section 6 concludes the paper and discusses future work.

2 The Simple Language of Communicating Objects - SLCO

The *Simple Language of Communicating Objects* (SLCO) provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. Figure 2 shows the main metaclasses and relations of the SLCO metamodel. The remainder of the metamodel is shown in Figure 3, which shows the subclasses of the abstract metaclasses *Statement*, *Trigger*, and *Expression*, and the related metaclasses and relations.

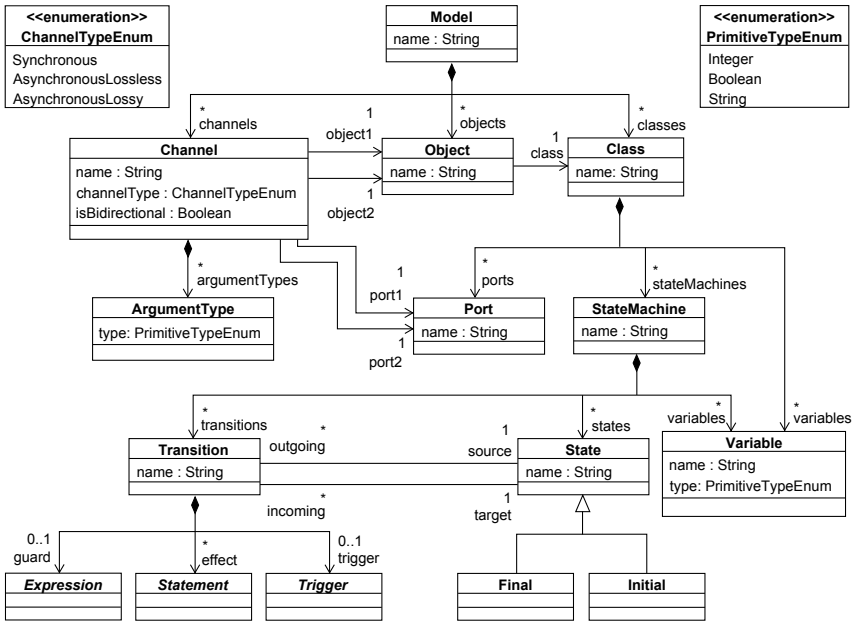
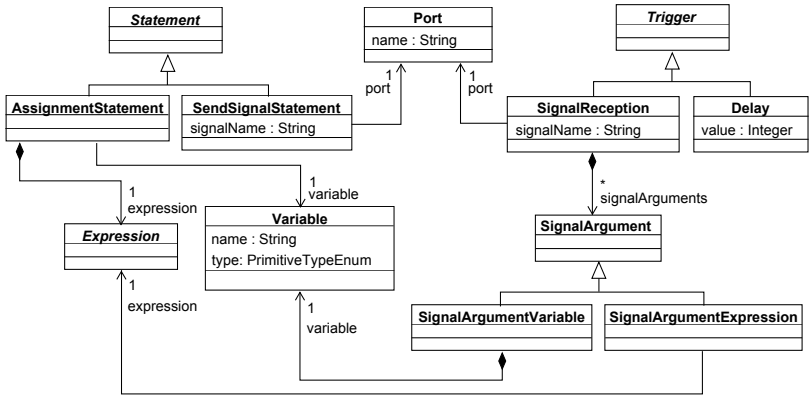


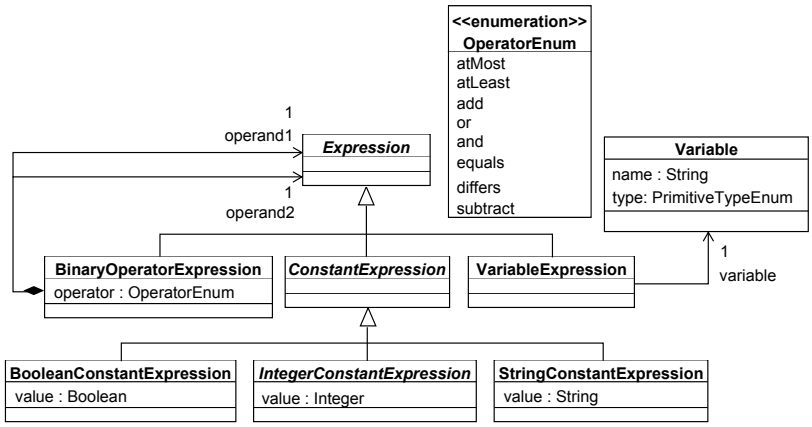
Figure 2: Overview of the SLCO metamodel

An SLCO model consists of a number of classes, objects, and channels. Objects are instances of classes and communicate with each other via channels, which are either bidirectional of unidirectional. SLCO offers three types of channels: synchronous channels, asynchronous lossy channels, and asynchronous lossless channels. An example of two objects connected by three channels is shown in Figure 4. The objects p and q, which are instances of classes P and Q, can communicate over channels $p1\_q1$, $q2\_p2$, and $p3\_q3$. The arrows at the ends of the channels denote the direction of communication. Synchronous channels are denoted by plain lines (e.g. $p1\_q1$), asynchronous lossless channels are denoted by dashed lines (e.g. $p3\_q3$), and asynchronous lossy channels are denoted by dotted lines (e.g. $q2\_p2$). A channel can only be used to send and receive signals with a certain signature, indicated by a number of argument types augmented to the channel. Channel $p1\_q1$, for instance, can only be used to send and receive signals with a boolean argument, and channel $q2\_p2$ only allows signals without any arguments.

A class describes the structure and behavior of its instances. A class has ports and variables that define the structure of its instances, and state machines that describe their behavior. It is possible to specify the initial values of variables. If no initial value is specified, integer variables are initialized to 0,



(a) Statements and triggers



(b) Expressions

Figure 3: Statements, triggers, and expressions in SLCO

boolean variables are initialized to *true*, and string variables are initialized to the empty string. Ports are used to connect channels to objects. Figure 4 shows that object p has ports *P1*, *P2*, and *P3*, connecting it to channels $p1\_q1$, $q2\_p2$, and $p3\_q3$, and that object q has ports *Q1*, *Q2*, and *Q3*, connecting it to the same channels.

A state machine consists of variables, states, and transitions. SLCO allows for two special types of states: a state can be an initial state or a final state. Each state machine has exactly one initial state, and can contain any number of ordinary and final states. Figure 5 shows an example of an SLCO model consisting of two state machines, whose initial states, `Initial`, are indicated by a black dot-and-arrow, and whose final states are denoted as circled black dots. As explained below, the left state machine specifies the behavior of object p and the right state machine specifies the behavior of object q, both already introduced in Figure 4.

A transition has a source and a target state, and possibly a guard, a trigger, a number of statements that form its effect, or a combination of these. A guard is a boolean expression that must hold to enable the transition from the source to the target state to be taken. For instance, `[n >= 2]` is the guard of the transition with the source `State` and the final state as the target state in the state machine of p. There are two types of triggers: a signal reception and a delay. A transition with a delay trigger is enabled after a specified amount of time has passed since entering its source state. Note that our running example

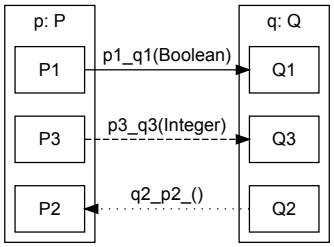


Figure 4: Objects, ports and channels in SLCO

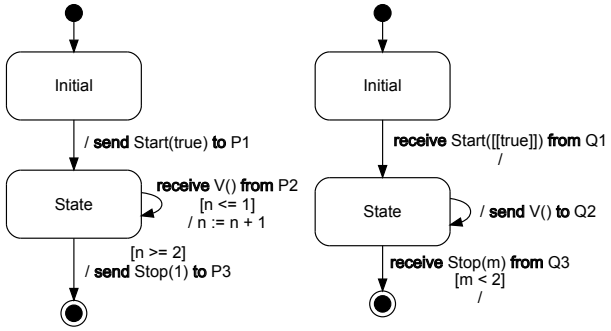


Figure 5: Two SLCO state machines

does not have this type of triggers. A transition with a signal reception trigger is enabled if a signal is received via the port indicated by the trigger. When a signal reception trigger is combined with a guard, naturally, the guard must hold for the transition to be enabled. It is allowed for the guard to refer to arguments of the signal just being received, which yields a form of conditional message reception. Take for instance the transition in q from `State` to the final state, with trigger `receive` `Stop`(m) `from` $Q3$. It is only taken if the value of the argument sent with the signal `Stop` is smaller than 2, specified as $[m<2]$. Additionally, another form of conditional signal reception is offered. Expressions given as arguments of a signal reception specify that only signals whose argument values are equal to the corresponding expressions are accepted. Thus, q in state `Initial` accepts only signals whose argument equals *true*.

When a transition is made from one state to another state, the statements that constitute the effect of the transition are executed. SLCO offers statements for assigning values to variables and for sending signals over channels. The state machines in Figure 4 specify the following communication between p and q. The two objects first communicate synchronously over channel $p1\_q1$, after which q repeatedly sends signals to p over the lossy channel $q2\_p2$. As soon as p receives 2 of the signals sent by q, it sends a signal over channel $p3\_q3$ and terminates. After receiving this signal, q terminates as well.

In addition to the graphical concrete syntax shown above, SLCO has a textual concrete syntax, which is used by the tools described in Section 3 to perform transformations of SLCO models. Listing 1 shows a part of the textual equivalent of the model described above.

3 Prototyping Semantics

Figure 1 shows that the SLCO transformation environment involves other languages and transformation tools. Designing the environment and defining its main ingredients required, among others, thorough understanding of the SLCO semantics, and its specification in terms of basic activities, each one either being executed by a particular object or being the result of objects' interaction. The idea behind the intermediate language CS is to specify explicitly these low-level activities, which are implicit in SLCO, and to serve as an underlying language to express the semantics of SLCO. As a result, CS together with the SLCO2CS transformation captures the semantics of SLCO. All languages and transformations described in this section are available for download [16].

3.1 Languages

The process of transforming an SLCO model into an LTS is split into two steps. First, the SLCO model is translated into a list of configurations and steps, represented in the CS language, where CS stands for

```
model M {
  classes
    P {
      variables Integer n
      ports P1 P2 P3
      state machines
        P {
          initial Initial state State final Final
          transitions
            Receive from State to State {
              trigger receive V() from P2
              guard n <= 1
              effect n := n + 1 }
             ...
        }
    }
  ...
  objects p:P q:Q
  channels p1_q1(Boolean) sync from p.P1 to q.Q1
           ...
}
```

Listing 1: Part of a textual SLCO model

*C*onfigurations and *S*teps. In this language, we describe the behavior of the entire system, resulting from the communication of the constituting objects, which are originally modeled as a set of separate state machines in SLCO. Then, this CS representation is transformed into a list of states and transitions, which form the LTS representation of the input model.

```
<
 <p, P, Initial> <q, Q, Initial>,
 [<<p, n>,0>, <<q, m>,0>],
 [<<p3_q3, p, P3, q, Q3>,>, <<q2_p2, q, Q2, p, P2>,>],
 initial
>
```

Listing 2: The initial configuration of the running example

CS The main ingredients in a CS description are *configurations* and *steps*. A configuration is a representation of a possible state of the system described by the SLCO model. A configuration can make a step, after which the system reaches another configuration. It is important to note that a step in CS does not correspond to a transition in SLCO, but, in general, several steps (not necessarily executed in sequence) accomplish the behavior specified with such an SLCO transition. Therefore, the CS representation of an SLCO model refines its behavior by splitting the execution in more basic activities.

Each configuration consists of three mandatory parts and an optional status. The configuration given in Listing 2 is the initial configuration of the model consisting of objects p and q, shown in Figures 4 and 5. The first part of a configuration specifies the current states of all state machines of all objects in the SLCO model. This part of the configuration is referred to as the *active states* of a configuration. If a state machine *sm* of an object *o* is currently in state *st*, this is specified as `<o, sm, st>` in the active states part of the configuration. This type of active state is referred to as *plain active state*. Recall now that the effect of an SLCO transition may consist of several statements. Thus, in general, the state machine can start executing the transition by receiving a signal, for instance, after which it starts executing the statements forming the effect of this transition. All these parts of the transition's execution are represented as

separate steps in a CS model. In view of the whole system, this means that the state machine to which this transition belongs to cannot proceed with any other transition as long as all statements are not executed; however, any other state machine is allowed to proceed with its own execution. In order to specify these intermediate configurations in the CS model, we introduce a second type of, so-called, *partial active states*. A partial active state <o, sm, st, k, m> represents that the kth statement in the list of statements is to be executed next as a part of the execution of the mth outgoing transition of state *st*. Consequently, the active states part of a configuration consists of a number of plain and partial active states, as many as the state machines in the model.

In the configuration shown in Listing 2, both active states, viz. <p, P, Initial> and <q, Q, Initial>, are plain. Listing 3 gives an example of a configuration in which <q, Q, State> is a plain active state and <p, P, State, 0, 1> is a partial active state, the latter indicating that the state machine of *p* (Figure 5) has to execute the 0th statement, n:=n+1, in the list of statements of the outgoing transition of state State whose identifier equals 1.

```
<
 <p, P, State, 0, 1> <q, Q, State>,
 [<<p, n>,0>, <<q, m>,0>],
 [<<p3_q3, p, P3, q, Q3>,>,<<q2_p2, q, Q2, p, P2>,>]
>
```

Listing 3: A configuration containing a plain active state and a partial active state

The second part of the configuration, the *valuation* part, maps variables to values. In the example configuration in Listing 2, it is specified as [<<p,n>,0>, <<q,m>,0>], expressing that variable *n* of object *p* and variable *m* of object *q* both have value 0 in this configuration.

The third part of the configuration represents a set of buffers and is simply referred to as the *buffers* of the configuration. For each asynchronous channel in the model, one or two buffers are introduced: in case of a bidirectional channel two buffers are introduced and in case of a unidirectional channel one buffer is introduced. The configuration in Listing 2 contains two buffers, one for each (unidirectional) channel, and they are both empty. The first buffer corresponds to channel $p3\_q3$ and the second buffer corresponds to channel $q2\_p2$. Because channel $p1\_q1$ is synchronous, it has no corresponding buffer. The optional status of this configuration is set to *initial*. If all state machines in a configuration are in a final state, the status of this configuration is set to *final*.

The dynamics of the system modeled by a set of communicating objects is represented by *steps*. Each step has a source and a target configuration, and an optional label. A step is a representation of a (basic) activity a given state machine has to perform as a part of the set of activities assigned to a transition in the SLCO representation of this state machine. Listing 4 shows two steps from the CS model of our SLCO running example model. The first step has a label that represents the reception of the asynchronous signal *V* by the state machine of *p*. The second step represents the execution of the assignment statement n := n + 1 of this state machine and does not have a label.

LTS The language LTS is a simple language for representing labeled transition systems as a list of states and a list of transitions. A state can be typed as initial or final state. Each transition is described as a couple of states, the source and the target states, and an optional label. Listing 5 shows the LTS description of a tiny LTS with four states and three transitions. State 0 is declared as an initial state, and states 1 and 3 are final states. There are three transitions, one of which has label *a*. There are different languages that describe LTSs; the ones used in our environment are described in Section 4. The

```
<
 <
  <p, P, State> <q, Q, State>, [<<p, n>,0>, <<q, m>,0>],
  [<<p3_q3, p, P3, q, Q3>,>, <<q2_p2, q, Q2, p, P2>,<V, >>]
 >,
 "receiving V()",
 <
  <p, P, State, 0, 1> <q, Q, State>, [<<p, n>,0>, <<q, m>,0>],
  [<<p3_q3, p, P3, q, Q3>,>, <<q2_p2, q, Q2, p, P2>,>]
 >
>
<
 <
  <p, P, State, 0, 1> <q, Q, State>, [<<p, n>,0>, <<q, m>,0>],
  [<<p3_q3, p, P3, q, Q3>,>,<<q2_p2, q, Q2, p, P2>,>]
 >,
 <
  <p, P, State> <q, Q, State>, [<<p, n>,1>, <<q, m>,0>],
  [<<p3_q3, p, P3, q, Q3>,>,<<q2_p2, q, Q2, p, P2>,>]
 >
>
```

Listing 4: Two steps depicting the reception of a signal and the execution of an assignment statement

most notable feature of LTS in comparison to these other languages is that it has a notion of final state, to distinguish between successful termination and deadlock. A state without any outgoing transitions represents a deadlock. It is considered undesirable for a system to reach these deadlock states. Reaching a final state and thus terminating successfully, however, is considered desirable behavior.

```
states
  initial 0
  final 1
  2
  final 3
transitions
  0 1
  0 "a" 2
  2 3
```

Listing 5: An example of an LTS

3.2 Tools

Now that the languages used to prototype SLCO are in place, we describe the two main transformations of our environment in this subsection. The first transformation produces a CS representation of an input SLCO model and the second transformation produces an LTS from the CS representation (Figure 1). As these transformations are implemented in the ASF+SDF Meta-Environment [6], we briefly describe it here as well.

ASF+SDF The language ASF+SDF is a combination of the two formalisms ASF [4] and SDF [22]. SDF stands for *Syntax Definition Formalism*. It is a formalism for defining syntax of context-free languages. ASF stands for *Algebraic Specification Formalism*. It is a formalism for the definition of conditional rewrite rules. Given a syntax definition in SDF of the source and target language, ASF can be used

to define a transformation from the source language to the target language. In ASF, conditional rewrite rules are specified using the concrete syntax of the input and output languages.

ASF in combination with SDF guarantees syntax safety of the implementation of the transformations. A transformation is syntax safe if it only accepts input that adheres to the syntax definition of the input language, and it always produces output that adheres to the syntax of the output language.

The ASF+SDF Meta-Environment is an IDE for ASF+SDF. It has a graphical user interface that offers syntax-highlighting for the specification of SDF and ASF definitions, and an interpreter and debugger for the execution and debugging of ASF specifications. The Meta-Environment can be used to create a command-line tool that parses and rewrites input adhering to the syntax definition of the input language, and outputs the result. These command-line tools employ memoization, which ensures that the result of a rewrite rule applied to a given term is computed only once. Both the ASF+SDF Meta-Environment and the command-line tools it generates use *Annotated Terms* (ATerms) [5] to represent terms internally. Because ATerms offer maximal subterm sharing, the internal representation of terms uses as little space as possible.

SLCO2CS An `SLCO` model is translated into `CS` in three phases. First, the initial configuration of the `SLCO` model is constructed. The list of active states of this initial configuration consists of the initial states of each of the state machines of the objects in the model, the valuation maps all variables to their initial values, and the buffers corresponding to all asynchronous channels are empty. Second, the set of all reachable configurations is generated. This phase is described in more detail below. Third, the list of configurations is traversed to find the configurations containing only active states that are final, and these configurations are marked as final.

In the second phase, first all configurations that are reachable from the initial configuration are created, as well as all the steps from the initial configuration to the reachable configurations. Then, all configurations that are reachable from these new configurations and the corresponding steps are created, and so on, until no new configuration is found. The configurations that are reachable from a source configuration are computed based on the active states of this source configuration. A step from one configuration to another is possible if one of the active states of the source configuration has an outgoing transition that is enabled or can execute a statement, in the corresponding `SLCO` state machine. Whether a transition is enabled depends on the valuation of the variables and the contents of the buffers of the source configuration. The valuation of the variables is used to determine whether the optional guard of such a transition holds and the content of the buffers is used to determine whether any signal receptions are possible. To determine all the configurations that are reachable from a given source configuration, all outgoing transitions of all active states of the source configuration are considered. The SDF functions discussed next are selected from the set of all functions that all together implement the generation of configurations and steps within the second phase of the transformation.

Listing 6 shows the signature of the functions *takeStepTransition* and *takeStepPartialTransition*. Applying these functions to terms representing a model, a configuration, an active step, and a transition results in a term representing a list of configurations and a list of steps. The function *takeStepTransition* is meant for handling plain active states, whereas the function *takeStepPartialTransition* is meant for handling partial active states.

Listing 7 shows one of the conditional rewrite rules in ASF that implement the function *takeStepTransition*. In the rules in Listing 7, 8, and 9, all variable names start with a dollar sign. In ASF+SDF, each term conforms to a *sort* and variables represent arbitrary terms of some sort. Variable names that end with a plus symbol represent lists of more than one terms of a sort and variable names that end with

```
takeStepTransition(
  Model, Configuration, ActiveState, Transition
) -> <Configuration*, Step*>

takeStepPartialTransition(
  Model, Configuration, ActiveState, Transition
) -> <Configuration*, Step*>
```

Listing 6: SDF definition of the functions that compute all possible steps from configurations

a question mark represent zero or one term. For example, the variables `$Statement+` and `$Guard?` in Listing 7 represent one or more statements and an optional guard, respectively. The first part of an ASF rule consists of the conditions of that rule. After an arrow (`====>`), the left-hand side and right-hand side of the rule follow, separated by an equal sign. If all the conditions hold, the left-hand side can be replaced by the right-hand side. The rules described in this paper use only one kind of condition: a matching condition. A matching condition consists of a right-hand side and a left-hand side, separated by a colon and an equal sign. The condition holds if both sides can be matched. In this case, the variables occurring at the right-hand side are instantiated such that both sides match.

The rule in Listing 7 deals with transitions with a signal reception trigger and an effect consisting of more than one statement. The source state of such a transition is represented by a plain active state. Because the effect of the transition consists of several statements, the plain active state provided as input is replaced by a partial active state in the resulting configuration. This partial active state indicates that none of the statements that form the effect have been executed yet and that the transition that has been taken from state *$ActiveState* has identifier *$NatCon*. The first step in Listing 4 is produced by applying this rule.

```
[takeStepTransition-reception-1]
  <$IdCon0, $IdCon1, $IdCon2> := $ActiveState,
  $IdCon3 from $IdCon2 to $IdCon4 {
    trigger $SignalReception
    $Guard?
    effect $Statement+
  } := $Transition,
  $NatCon := getTransitionNumber($Model, $Transition, $ActiveState),
  $ActiveState0 := <$IdCon0, $IdCon1, $IdCon2, 0, $NatCon>,
  <$Configuration0, $Step0> := processSignalReception(
    $SignalReception, $ActiveState, $ActiveState0, $Model, $Configuration,
    $IdCon0, $IdCon1
  )
  ====>
  takeStepTransition(
    $Model, $Configuration, $ActiveState, $Transition
  ) = <$Configuration0, $Step0>
```

Listing 7: ASF rule that specifies how a signal reception trigger on a transition from a plain active state is processed

Listing 8 shows yet another of the conditional rewrite rules that implements the function *takeStepTransition*. This rule applies to transitions that have no trigger and only one single assignment statement that forms its effect. In the configuration resulting from this rule, the original active state is replaced by another plain active state, because the effect consists of only one statement. The function *processAssignmentStatement*, used by the function *takeStepTransition*, produces a configuration and a step. The

configuration is an updated version of the configuration provided as input, in which the active state *$ActiveState* is replaced by *$ActiveState0* and the valuation is adapted according to the assignment statement. The step consists of the original configuration and the updated configuration.

```
[takeStepTransition-assignment-0]
  <$IdCon0, $IdCon1, $IdCon2> := $ActiveState,
  $IdCon3 from $IdCon2 to $IdCon4 {
    $Guard?
    effect $AssignmentStatement
  } := $Transition,
  $ActiveState0 := <$IdCon0, $IdCon1, $IdCon4>,
  <$Configuration0, $Step0> := processAssignmentStatement(
     $AssignmentStatement, $ActiveState, $ActiveState0, $Configuration,
     $IdCon0, $IdCon1
  )
  ====>
  takeStepTransition(
    $Model, $Configuration, $ActiveState, $Transition
  ) = <$Configuration0, $Step0>
```

Listing 8: ASF rule that specifies how a single assignment statement on a transition from a plain active state is processed

Listing 9 deals with transitions with an optional guard, an optional trigger, and an effect consisting of more than one statement. The source state of this transition is a partial active state. The function *getStatements* that is used by this rule takes a list of statements and an transition identifier and returns the statement with the given index and all following statements. One of the matching conditions states that this rule only applies if the statement with index *$NatCon0* is an assignment statement and that there are no statements after this statement. The second step of Listing 4 is the result of applying this rule to the target configuration of the first step.

```
[takeStepsPartialTransition-assign-0]
  $IdCon3 from $IdCon2 to $IdCon4 {
    $Trigger?
    $Guard?
    effect $Statement+
  } := $Transition,
  $AssignmentStatement := getStatements($Statement+, $NatCon0),
  $ActiveState0 := <$IdCon0, $IdCon1, $IdCon4>,
  <$Configuration0, $Step0> := processAssignmentStatement(
    $AssignmentStatement, <$IdCon0, $IdCon1, $IdCon2, $NatCon0, $NatCon1>,
    $ActiveState0, $Configuration, $IdCon0, $IdCon1
  )
  ====>
  takeStepPartialTransition(
    $Model, $Configuration, <$IdCon0, $IdCon1, $IdCon2, $NatCon0, $NatCon1>, $Transition
  ) = <$Configuration0, $Step0>
```

Listing 9: ASF rule that specifies how an assignment statement on a transition from a partial active state is processed

CS2LTS The tool `CS2LTS` translates lists of configurations and steps from `CS` to `LTS`, as shown in Figure 1, and it is also implemented using ASF+SDF. The way we have defined `CS` makes this translation

rather straightforward. Each configuration is mapped to a unique natural number and, possibly, an optional status. The optional status indicates whether a state in the LTS is an initial or a final state and it is equal to the status of the configuration that corresponds to the state. Each step is transformed to a pair of natural numbers representing its configurations, possibly decorated by an optional label. The label of a transition is equal to the label of the corresponding step.

4 Verification and Visualization

4.1 Visualization

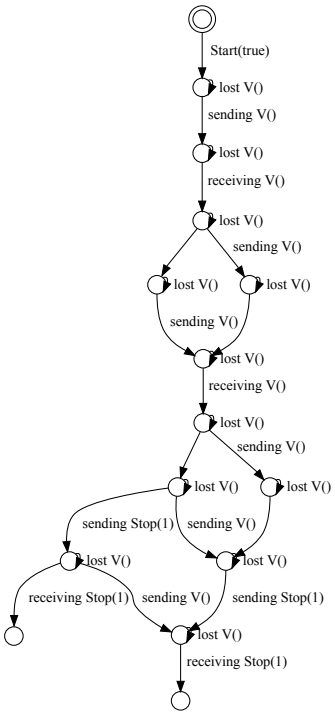


Figure 6: An LTS visualized using `Dot`

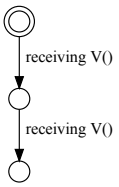


Figure 7: A reduced LTS visualized using `Dot`

The `Dot` language is a language for graph visualization that is part of the graphviz toolset [10]. We use this language to visualize LTSs, by translating LTSs to graphs in the format of the `Dot` language. This translation is straightforward, because the way graphs are described in this language is similar to the description of LTSs. A graph description in `Dot` is a list of nodes and edges from node to node, combined with attributes that specify how particular nodes and edges should be displayed. These attributes define, for example, the color, width, height, and the type of line used to draw these nodes and edges. LTSs are transformed to `Dot` graphs by translating all states to nodes and all transitions to edges, and specifying how initial states, final states, and normal states should be visualized. Figure 6 shows the LTS that corresponds to the model in Figure 4 and 5, visualized using `Dot`. In the visual representation, a transition label is always place on the right-hand side of the transition. Several transitions do not have labels, as they correspond to assignment statements.

4.2 Verification

For small LTSs, like the one in Figure 6, it is easy to verify properties manually by inspecting the graph. In case of larger LTSs, reduction techniques can be applied to first reduce the LTS before verifying properties manually. The tool *ltsconvert*, which is part of the mCRL2 toolset [13], takes an LTS in various formats as input and converts it to an equivalent LTS in another format. One of the formats that *ltsconvert* is able to process is the `Dot` format, which although meant to describe graphical representations of graphs, is also used to represent LTSs. The tool is also capable of reducing LTSs, by means of an equivalence relation, and several equivalence relations are supported. Figure 7 shows the LTS obtained after reduction has been applied to the LTS in Figure 6, by means of the branching bisimilarity relation [12]. The LTS has been reduced by, first, turning all labels except `receiving V()` to internal unobservable labels and then removing all redundant states and transitions using *ltsconvert*. A similar procedure can be applied for other reductions this tool supports. After the reduction, it is obvious that the following property holds in our running example: "signal V is received exactly two times."

As mentioned in Section 1, there exists a number of transformations that transform `SLCO` models into other `SLCO` models with equivalent observable behavior. By producing an LTS for the input and the output model of such a transformation, and reducing both LTSs using the technique described above, the correctness of this transformation for the given input model can be verified by comparing the reduced LTSs. If reducing the LTS of both models leads to the same LTS, the transformation has indeed preserved the observable behavior.

When LTSs get too large for reduction and manual inspection, other tools can be used for verification. One approach is converting LTSs to the *BCG* and *AUT* file formats that are used by the CADP toolset [11] to represent LTSs. The CADP toolset offers tools that take an LTS and a temporal logic property as input and perform on-the-fly verification of the property on the LTS. Alternatively, the previously mentioned mCRL2 toolset can be used for verification too. This toolset includes a tool that can transform LTSs to the proprietary format of the toolset, and tools that can be used to analyze, simulate, manipulate, and visualize models described using this format. These two example toolsets clearly show the added benefit of producing LTSs from `SLCO` models. Transforming models to this common description format makes it possible to verify properties of models using existing tools, without additional effort.

5 Related Work

Hooman and Van der Zwaag [15] used the interactive theorem prover PVS to define the semantics of a subset of the UML. In this subset, the behavior of objects is specified using state machines that communicate with each other both synchronously and asynchronously. Proving properties of models in this approach is done manually using PVS. This is a complex task that requires expertise in PVS, which can be simplified using certain predefined strategies. A disadvantage of this approach is that is does not offer the reusability of other existing tools that our approach offers. An advantage of this approach is that it does not suffer from the state-space explosion problem, because the complete state space of models does not have to be generated for property verification.

Di Ruscio [8] et al. define the semantics of a DSL for the development of telephony services using Abstract State Machines (ASM). Because ASMs can be executed, this definition can be used to simulate models specified in their DSL. The approach is meant for the specification of the behavioral semantics of the DSL only, and does not offer verification of models. Proving properties for all models in general or any specific model is not investigated. In theory, however, properties of models could be verified in the domain of ASMs.

Sadilek and Wachsmuth [19] propose a technique for defining the semantics of DSLs that uses model instances as configurations and QVT relations to define steps between configurations. Configurations, representing model instances, can be visualized using the same editors used to create models. By reusing the existing editors, visual interpreters and visual debuggers can be created with relatively little effort. Although this technique is suited for simulation of models, it is not efficient enough for state-space generation. Because each configuration is represented by a model, a lot of memory is needed to store all possible configurations.

A number of approaches use Maude to specify the operational semantics of DSLs [18, 17]. Given the operational semantics of a DSL in Maude, other techniques can be applied to verify properties of DSL models. Both an LTL model checker [9] and a μ-calculus model checker [23] are available for rewrite systems specified in Maude. Although it is clear that model checking techniques can be implemented in Maude and applied to specifications of the semantics of DSLs, not all techniques applicable to LTSs which we aim to exploit, such as reduction and visualization, have been implemented in Maude. It might be the case, therefore, that a given technique must first be implemented in Maude before it can be used in combination with a specification of the semantics of a DSL. With our approach, we can connect to various tools and apply existing techniques only by adapting the representation of LTSs, if needed.

6 Conclusions and Future Work

Defining the semantics of SLCO by implementing a transformation that transforms SLCO models to LTSs has a number of advantages. Existing tools for verification and visualization can be reused, because LTSs are a common input representation used by a number of tools. The biggest advantages of using the ASF+SDF Meta-Environment for this implementation are ATerms, used for representing terms, and the command-line tools that can be automatically generated. Using ATerms guarantees efficient use of memory and the command-line tools offer efficient execution of rewrite rules, without any additional effort during the implementation. Both execution speed and efficient use of memory are important in this case because the state spaces of models represented by LTSs are typically very large.

As future work we want to define a formal semantics of SLCO using, for instance, structural operational semantics. The prototype of the semantics of SLCO and its implementation as presented here, make a solid ground for this work, as we got better understanding of the semantics of this DSL and were able to investigate a number of design decisions. As SLCO has the notion of time delays, future work is also going to investigate possibilities to connect our SLCO environment to languages and tools for time analysis. We also consider to apply the approach taken in this paper to other DSLs. The approach lends itself well for the creation of state-space generators based on the operational semantics of a given DSL, and prototyping the semantics of languages with informal or incompletely defined operational semantics.

References

[1] M. van Amstel, M. van den Brand & L. Engelen (2010): *An Exercise in Iterative Domain-Specific Language Design*. In: *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ACM, Antwerp, Belgium, pp. 48–57, doi:10.1145/1862372.1862386.

[2] M.F. van Amstel, M.G.J. van den Brand, Z. Protić & T. Verhoeff (2008): *Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap?* In: *Proceedings of the First International Conference on Model Transformation (ICMT'08)*, *LNCS* 5063, Springer, Zürich, Switserland, pp. 61–75, doi:10.1007/978-3-540-69927-9.

[3] D. Baum (2003): *NQC Programmer's Guide*.

[4] J. A. Bergstra (1989): *Algebraic Specification*, chapter 1, pp. 1–66. ACM.

[5] M.G.J. van den Brand & P. Klint (2007): *ATerms for manipulation and exchange of structured data: It's all about sharing*. Information and Software Technology 49, pp. 55–64, doi:10.1016/j.infsof.2006.08.009.

[6] M. van den Brand et al. (2001): *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment*. In: Proceedings of Compiler Construction 2001 (CC 2001), LNCS 2027, Springer-Verlag, London, UK, pp. 365–370, doi:10.1007/3-540-45306-7_26.

[7] A. van Deursen, P. Klint & J. Visser (2000): *Domain-Specific Languages: An Annotated Bibliography*. SIGPLAN Notices 35(6), pp. 26–36, doi:10.1145/352029.352035.

[8] D. Di Ruscio et al. (2006): *A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development*. Technical Report, Laboratoire d'Informatique de Nantes-Atlantique (LINA), doi:10.1.1.101.6120.

[9] S. Eker, J. Meseguer & A. Sridharanarayanan (2002): *The Maude LTL Model Checker*. In: Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002), ENTCS 71, Elsevier, Amsterdam, doi:10.1016/S1571-0661(05)82534-4.

[10] J. Ellson et al. (2002): *Graphviz Open Source Graph Drawing Tools*. In: International Symposium on Graph Drawing, LNCS 2265, Springer-Verlag, pp. 594–597, doi:10.1007/3-540-45848-4_57.

[11] H. Garavel, F. Lang, R. Mateescu & W. Serwe (2011): *CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes*. In: TACAS, LNCS 6605, Springer, pp. 372–387, doi:10.1007/978-3-642-19835-9_33.

[12] R.J. van Glabbeek & P. Weijland (1996): *Branching time and abstraction in bisimulation semantics*. Journal of the ACM 43, pp. 555–600, doi:10.1145/233551.233556.

[13] J.F. Groote et al. (2006): *The Formal Specification Language mCRL2*. In: MMOSS, Dagstuhl Seminar Proceedings 06351. Available at `http://drops.dagstuhl.de/opus/volltexte/2007/862`.

[14] G.J. Holzmann (1997): *The Model Checker SPIN*. IEEE Transactions on Software Engineering 23(5), pp. 279–295, doi:10.1109/32.588521.

[15] J. Hooman & M.B. van der Zwaag (2006): *A Semantics of Communicating Reactive Objects with Timing*. International Journal on Software Tools for Technology Transfer (STTT) 8(2), pp. 97–112, doi:10.1007/s10009-005-0207-8.

[16] Prototyping SLCO Semantics Project: `http://code.google.com/p/prototyping-slco-semantics/`.

[17] J.E. Rivera, F. Durán & A. Vallecillo (2009): *Formal Specification and Analysis of Domain Specific Models Using Maude*. Simulation 85, pp. 778–792, doi:10.1177/0037549709341635.

[18] V. Rusu (2011): *Embedding Domain-Specific Modelling Languages in Maude Specifications*. SIGSOFT Softw. Eng. Notes 36, pp. 1–8, doi:10.1145/1921532.1921557.

[19] D.A. Sadilek & G. Wachsmuth (2008): *Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages*. In: Proceedings of European conference on Model Driven Architecture: Foundations and Applications, LNCS 5095, Springer-Verlag, pp. 63–78, doi:10.1007/978-3-540-69100-6_5.

[20] D. C. Schmidt (2006): *Model-Driven Engineering*. Computer 39(2), pp. 25–31, doi:10.1109/MC.2006.58.

[21] B.D. Theelen et al. (2007): *Software/Hardware Engineering with the Parallel Object-Oriented Specification Language*. In: Proceedings of IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'07), IEEE, Nice, France, pp. 139–148, doi:10.1109/MEMCOD.2007.371231.

[22] E. Visser (1997): *Syntax Definition for Language Prototyping*. Ph.D. thesis, University of Amsterdam.

[23] B.-Y. Wang (2005): *μ-Calculus Model Checking in Maude*. ENTCS 117, pp. 135–152, doi:10.1016/j.entcs.2004.06.025.