On the Formal Semantics of SDL-2000: A Compilation Approach using Abstract State Machines

Robert Eschbach[1], Reinhard Gotzhein[1] and Andreas Prinz[2]

[1] Department of Computing Sciences, University of Kaiserslautern, D-67653 Kaiserslautern, Germany

[2] Institut für Informatik, Humboldt-University Berlin, Rudower Chaussee 25, D-12489 Berlin, Germany

March 1, 2000

Abstract. In November 1999, a new version of *SDL (Specification and Description Language)* called *SDL-2000* has passed ITU, an international standardization body for telecommunication. SDL is a fairly complex, graphical formal description technique for the development of distributed systems, and has been broadly used in industry for many years. Efforts to define the semantics of SDL-2000 formally have started early in 1998. By now, a draft formal semantics is available, which is determined to become the official formal SDL semantics after its approval in 2000. It is based on the formalism of *Abstract State Machines (ASMs)*, which has been selected for several reasons including intelligibility and executability.

The formal semantics of SDL addresses the static semantics, transformation rules, and the dynamic semantics. The approach taken to define the dynamic semantics is particularly interesting. Although basically being operational, it differs from existing approaches in several ways. In this paper, we address and highlight some of these differences, using a simplified specification language called SSL instead of SDL. In defining a formal dynamic semantics for SSL, we formally describe an abstract machine, a compilation function mapping SSL specifications to code of this machine, and an operational definition of the set of initial states, using ASM as the underlying formalism. Furthermore, we present in some detail the semantics of SSL procedure calls.

1 Introduction

Since 1976, SDL (Specification and Description Language) is an ITU standardized language for the development of distributed systems in general, and telecommunication systems in particular. With its graphical syntax, its object-oriented features, its direct support for reuse, and its integration with other development languages, SDL satisfies the primary needs of system developers. Today, SDL is being broadly applied in industry, and is being supported by powerful commercial tool environments.

In 1988, the semantics of SDL has been defined formally, upgrading the language to a formal description technique (FDT). This formal definition has been updated and maintained for subsequent versions of the SDL language in 1992 and 1996. Essentially, the formal semantics is given by a set of Meta IV programs that take an SDL specification as input, determine the correctness of its static semantics, perform a number of transformations to replace several SDL language constructs, and interpret the specification. It has been argued that this style of defining the formal semantics is particularly suitable for tool builders.
In November 1999, a new version of the language called SDL-2000 has been approved as international standard. In addition to the usual language maintenance, SDL-2000 offers important new features, including object-oriented data type definitions, a unified concept for blocks, processes, and composite states, and exception handling. Based on the assessment that the existing Meta IV programs would be too difficult to update and maintain, it was decided to conceive a new formal semantics for SDL-2000 from scratch. For this purpose, a special task force, the *SDL semantics group*, consisting of experts from Germany and China including the authors of this paper, was formed. The draft formal semantics defined by this group ([PGGE]) has been determined to become the official formal SDL semantics after its approval, which is expected for November 2000. Among the design objectives for the formal semantics are intelligibility, maintainability, conciseness, and in particular *executability*, which has become a key issue. Executability calls for an operational formalism with readily available tool support. For this reason, *Abstract State Machines (ASMs)* have finally been selected as the underlying formalism. To support executability, the formal semantics defines, for each SDL specification, corresponding ASM code. This differs substantially from the interpreter view taken in previous work, and enables SDL-to-ASM-compilers. Thus, given the availability of an SDL-to-ASM-compiler, it will be straightforward to generate reference implementations.
The approach taken by the SDL semantics group to define the dynamic semantics of SDL-2000, though basically operational, differs in several aspects from the approach taken to define the semantics of SDL-88 through SDL-96, and also from the style of other ASM approaches (cf. [GGP99]):

- The style of the semantics has the characteristics of a *compiler approach*, as compared to the interpreter approach chosen, for instance, to define the semantics of SDL-88 through SDL-96. A compiler usually performs lexical and syntactic analysis functions, just like an interpreter does. Unlike an interpreter, compilers produce executables for some kind of machine, instead of executing a version of the source directly. Thus, the source is viewed as a collection of instructions rather than a data structure.
- In practice, processing specifications into an abstract syntax representation is simpler and faster than compiling it into some kind of executable code. On the other hand, compilation into code directly enables executability. There is a technique to combine the advantages of both approaches, namely the generation of abstract code "running" on an abstract machine. The SDL semantics group has devised an *SDL Abstract Machine*, which is defined in

terms of ASMs. SDL specifications are then compiled using the instruction set of this machine.
– In ASM models, the set of initial states is defined by constraints imposed on functions, which are defined by formulae. These constraints are required to hold in the first state of each run of the ASM. In the ASM model of the SDL semantics, these constraints determine only what is called *set of pre-initial states*. The actual set of initial states is then determined in an operational way by an *initialization phase* that starts in a pre-initial state. This way the initialization phase and the execution phase may even overlap.

Due to the complexity of SDL-2000, the formal semantics is of substantial size. Therefore, in order to present and illustrate the application of the basic techniques above to the definition of the SDL semantics, we have chosen a very small language called *SSL (Simple Specification Language)*. In the following, we define an abstract machine for SSL, a compilation function mapping SSL specifications to code running on this machine, and the initialization and execution phases. Moreover, we present in more detail the semantics of SSL procedure calls. Our model of procedure semantics allows for call-by-value and call-by-reference semantics and supports the concept of static binding.
Please note that the work reported here is not an academic exercise, but takes place in a real-life industrial setting. For this reason, some decisions taken in defining the SDL (and hence SSL) semantics are rather pragmatic or others are strongly influenced by the SDL experts. In our opinion, such work is crucial to demonstrate the usefulness of formalisms such as ASMs. In this sense, the paper reports on work that is still in progress, but which has reached a stage where most "design" decisions have been taken, and a sufficiently stable draft is available (cf. [PGGE]).
The rest of this paper is organized as follows. Section 2 provides an overview of the formal semantics for SDL-2000. Section 3 introduces the language SSL. Section 4 defines the abstract compilation of SSL specifications to SSLM programs. Section 5 defines the SSL abstract machine (SSLM). Section 6 describes the initialization. Section 7 presents conclusions. Throughout this paper we assume the reader to be familiar with Gurevich's concept of ASMs (cf. [Gur95,Gur97,Gur99]).

2 Overview of the Formal SDL Semantics

The definition of the formal semantics of SDL-2000 is structured into the following major parts:

– grammar,
– well-formedness conditions (static semantics),
– transformation rules, and
– dynamic semantics.

The *grammar* defines the set of syntactically correct SDL specifications. In the SDL standard, a concrete textual, a concrete graphical, and an abstract grammar are defined using BNF with some extensions to capture graphical language

elements. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.
The *well-formedness conditions (static semantics)* impose additional context constraints on syntactically correct specifications, such as which names it is allowed to use at a given place, which kind of values to assign to variables, etc.
For some language constructs, the formal semantics is not provided directly, but through *transformation rules*. These rules define how a given specification is to be transformed in order to replace the language constructs by other language elements, and are formally represented as rewrite rules.
Finally, the *dynamic semantics* is given to syntactically correct SDL specifications satisfying the well-formedness conditions, after application of the transformation rules. The dynamic semantics consists of the following parts as illustrated by figure 1.

- The *SDL Abstract Machine (SAM)* defines basic signal flow concepts of SDL such as signals, timers, exceptions, and gates in terms of the ASM model. Furthermore, ASM agents are specialized to model agents in the context of SDL. Finally, several signal processing and behavior primitives - in a sense the abstract machine instructions of the SAM - are predefined.
- The *SDL Abstract Machine Programs* defines the set of computations. Computations consist of an initialization phase and an execution phase. SAM programs have predefined parts that are the same for all SDL specifications, and variable parts that are generated from the abstract syntax representation of a given SDL specification.
  - The *initialization* handles static structural properties of the specification by recursively unfolding all the static objects of the specification. In fact, the same happens during the execution phase when new SDL agents are created dynamically. From this point of view, the initialization is merely the instantiation of the SDL system agent.
  - The *compilation function* maps behavior representations into the SDL Abstract Machine primitives. This function amounts to an abstract compiler taking the abstract syntax tree (AST) of an SDL specification as input and transforming it to the abstract machine instructions.
- The *data semantics*, which is separated from the rest of the dynamic semantics by an interface. The use of an interface is intentional at this place. It will allow to exchange the data model, if for some domain another data model is more appropriate than the SDL built-in model. Moreover, also the SDL built-in model can be changed this way without affecting the rest of the semantics, enhancing its maintainability.

As in the past, the new formal semantics is defined starting from the abstract syntax of SDL, which is documented in the SDL standard. From this abstract syntax, a behavior model that can be understood as abstract code generated from an SDL specification is derived. This approach differs substantially from the interpreter view taken in the previous SDL semantics definition, and will enable SDL-to-ASM compilers.

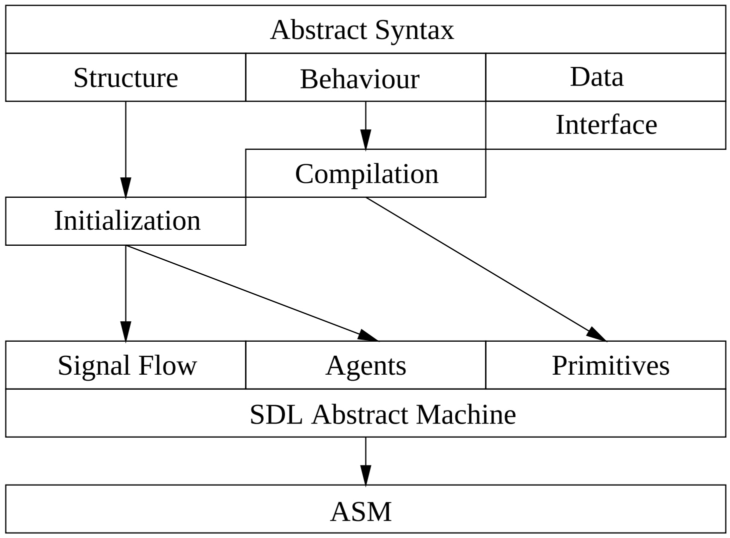

Fig. 1. Overview of the Dynamic Semantics

The dynamic semantics associates, with each SDL specification, a particular multi-agent, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents cooperatively performing concurrent machine runs. The behavior of agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible computations (called "runs" in the context of ASM). Each agent has its own partial view on a global state, which is defined by a set of static and dynamic functions and domains. By having non-empty intersections of partial views, interaction among agents can be modeled.

3 SSL: A Simple Specification Language

In order to illustrate our approach to the formal SDL-2000 semantics and the application of the basic techniques introduced in section 1 we have chosen a very small language called SSL (Simple Specification Language). SSL is untyped, and is based on a small number of statements including assignment and the conditional statement. Furthermore, an SSL-Specification may contain procedure definition and procedure calls. A procedure definition can contain reference parameters and value parameters.

3.1 Basics

The syntax of SSL is given below. We use Name as a predefined lexical unit and describe the syntax using a variant of BNF (Backus-Naur-Form). More precisely we will just give an *abstract syntax* of the language. We assume the concrete

syntax to be handled by an ordinary parser that provides us with the abstract representation in terms of the abstract syntax.
Please note, that we use prefixes in the syntax to make the role of the items clear. Prefixes are separated from the nonterminal name by a "-". Prefixes do not have a semantics.

3.2 Specification

An SSL specification describes a concurrent system. It is given as a list of variables, a list of local specifications, a list of locally defined procedures, and a main expression. To execute an SSL specification means to first start the execution of the enclosed SSL specifications and then to evaluate the main-expression. The enclosed specifications evaluate their main-expression concurrently with.

$$\begin{aligned}
\mathsf{sslSpec} &::= \mathsf{spec\text{-}Name}\ \mathsf{variableDecl}^*\ \mathsf{sslSpec}^*\ \mathsf{procedureDef}^*\\
&\quad\ \ \mathsf{main\text{-}expression}\\
\mathsf{variableDecl} &::= \mathsf{Name}
\end{aligned}$$

3.3 Procedures

A procedure is given as a list of its parameters and an expression for the body.

$$\begin{aligned}
\mathsf{procedureDef} &::= \mathsf{proc\text{-}Name}\ \mathsf{refParam}^*\ \mathsf{valParam}^*\ \mathsf{variableDecl}^*\\
&\quad\ \ \mathsf{procedureDef}^*\ \mathsf{main\text{-}expression}\\
\mathsf{refParam} &::= \mathsf{Name}\\
\mathsf{valParam} &::= \mathsf{Name}
\end{aligned}$$

3.4 Expressions

Expressions in the scope of SSL include also the usual behavior of statements. In this case, their return value is simply ignored. The nonterminal function denotes constants (like 0 and 1) as well as functions (as + and -). The meaning of these constructs is supposed to be part of the language. The semantics of ifExpr, variable, assignment, procCall, procReturn and equality is as usual. For sequence, the semantics is the sequential interpretation of the subexpressions. The resulting value of the sequence is the last one.

$$\begin{aligned}
\mathsf{expression} &::= \mathsf{function} \mid \mathsf{ifExpr} \mid \mathsf{variable} \mid \mathsf{equality} \mid \mathsf{sequence} \mid\\
&\quad\ \ \mathsf{assignment} \mid \mathsf{procCall} \mid \mathsf{procReturn}\\
\mathsf{function} &::= \mathsf{Name}\ \mathsf{expression}^*\\
\mathsf{ifExpr} &::= \mathsf{test\text{-}expression}\ \mathsf{then\text{-}expression}\ [\mathsf{else\text{-}expression}]\\
\mathsf{variable} &::= \mathsf{Name}\\
\mathsf{equality} &::= \mathsf{lhs\text{-}expression}\ \mathsf{rhs\text{-}expression}\\
\mathsf{sequence} &::= \mathsf{fst\text{-}expression}\ \mathsf{nxt\text{-}expression}
\end{aligned}$$

$$\begin{aligned} \mathsf{assignment} &::= \mathsf{Name\ expression} \\ \mathsf{procCall} &::= \mathsf{Name\ ref\text{-}Name^*\ expression^*} \\ \mathsf{procReturn} &::= \mathsf{expression} \end{aligned}$$

Please note, that the syntax domain names are also used as the constructors

```
specification Hanoi
procedure hanoi (ref x,y,z, val n)
  if n = 1 then
     x := x - 1; z := z + 1; return;
  else
     call hanoi(x,z,y,n-1); call hanoi(x,y,z,1);
     call hanoi(y,x,z,n-1); return;
end hanoi

var x,y,z,n
begin Hanoi
  x := 4; y := 0; z := 0; n := 4; call hanoi(x,y,z,n);
end Hanoi
```

Fig. 2. Specification Hanoi

for the syntax tree. A sample SSL specification can be found in figure 2. Its representation in abstract syntax is given in figure 3. The values of the names *hanoiProc*, *hanoiProcMain* and *HanoiSpecMain* are given separately for better presentation, although they should be considered to be inline.

4 Compilation

The compilation defines a mapping from the behavioral part of the abstract syntax into ASM behavior. In fact, an abstract code is defined as the result of this mapping. The behavior belonging to the behavior primitives in the abstract code is defined within section 5.

4.1 Basics

In this section, we define a compilation function for SSL specifications and for SSL procedures.

$$\begin{aligned} &compileSpec : \mathsf{sslSpecification} \rightarrow AbstractCode \\ &compileProc : \mathsf{procedureDef} \rightarrow AbstractCode \end{aligned}$$

For the representation of code, we use a special labeling. Each instruction has a label attached to it. The specification keeps the information of the current

```
sslSpec(Hanoi, <x,y,z,n>, <>, <hanoiProc>, HanoiSpecMain)

hanoiProc = procedureDef(hanoi, <x,y,z>, <n>, <>, <>, hanoiProcMain)

hanoiProcMain =
  sequence(
    ifExpr(equality(variable(n),function(1,<>)),
      sequence(
        assignment(x,function(-,<variable(x),function(1,<>))),
        sequence(
          assignment(z,function(+,<variable(z),function(1,<>))),
          procReturn(undefined) ) ),
      sequence(
        procCall(hanoi,<x,z,y>,
          <function(-,<variable(n),function(1,<>)>)> ),
        sequence(procCall(hanoi,<x,y,z>,<function(1,<>)>),
          sequence(
            procCall(hanoi,<y,x,z>,
              <function(-,<variable(n),function(1,<>)>)> ),
            procReturn(undefined) ) ) ) ) )

HanoiSpecMain =
  sequence(assignment(x,function(4,<>)),
    sequence(assignment(y,function(0,<>)),
      sequence(assignment(z,function(0,<>)),
        sequence(assignment(n,function(4,<>)),
          procCall(hanoi,<x,y,z>,<variable(n)>) ) ) ) )
```

Fig. 3. Abstract Syntax of Specification Hanoi

label and selects the instruction to process according to this label. Therefore, we introduce a static domain *Label*. The abstract code is then a set of instructions.

$$AbstractCode = Instruction\textbf{-set}$$

where *Instruction*-**set** denotes a set of instructions, and an instruction is given by

$$Instruction = Label \times Primitive$$

The compiled code of all specifications and procedures is merged together in the initial state, and specifications and procedures are characterized by their entry label.

$$entrySpec : \mathsf{sslSpecification} \rightarrow Label$$
$$entryProc : \mathsf{Name} \rightarrow Label$$

For SSL we introduce the following primitives. Again, the prefixes are without semantic relevance.

$$
\begin{aligned}
\mathit{Primitive} &= \mathit{Pfun} \cup \mathit{Pif} \cup \mathit{Pvar} \cup \mathit{Peq} \cup \mathit{Pcopy} \cup \mathit{Passign} \cup \mathit{Pcall} \cup \mathit{Preturn} \\
\mathit{Pfun} &= \mathsf{Name} \times \text{value-}\,\mathit{Label}^{*} \times \text{next-}\,\mathit{Label} \\
\mathit{Pif} &= \text{condition-}\,\mathit{Label} \times \text{then-}\,\mathit{Label} \times \text{next-}\,\mathit{Label} \\
\mathit{Pvar} &= \mathsf{Name} \times \text{next-}\,\mathit{Label} \\
\mathit{Peq} &= \text{lhs-}\,\mathit{Label} \times \text{rhs-}\,\mathit{Label} \times \text{next-}\,\mathit{Label} \\
\mathit{Pcopy} &= \text{value-}\,\mathit{Label} \times \text{next-}\,\mathit{Label} \\
\mathit{Passign} &= \mathsf{Name} \times \text{value-}\,\mathit{Label} \times \text{next-}\,\mathit{Label} \\
\mathit{Pcall} &= \mathsf{Name} \times \text{value-}\,\mathit{Label}^{*} \times \text{ref-}\,\mathsf{Name}^{*} \times \text{next-}\,\mathit{Label} \\
\mathit{Preturn} &= \text{value-}\,\mathit{Label}
\end{aligned}
$$

In order to define the compilation function, we introduce two auxiliary functions.

$$\mathit{unique} : \mathsf{expression} \rightarrow \mathit{Label}$$

This function is provided by the compiler: a unique labeling of the expressions. Entry labels of expressions are determined by a recursive function

$$\mathit{entryLabel} : \mathsf{expression} \rightarrow \mathit{Label}$$

In the remaining text, we use the following convention: we often write $x.f$ instead of $f(x)$. Function *entryLabel* is recursively defined as follows.

$\mathit{entryLabel}(e : \text{expression}) \ \equiv$
  if $e \in$ sequence then $\mathit{entryLabel}(e.\mathsf{fst\text{-}expression})$
  elsif $e \in$ equality then $\mathit{entryLabel}(e.\mathsf{lhs\text{-}expression})$
  elsif $e \in$ assignment then $\mathit{entryLabel}(e.\mathsf{expression})$
  elsif $e \in$ ifExpr then $\mathit{entryLabel}(e.\mathsf{test\text{-}expression})$
  elsif $e \in$ procReturn $\wedge$ e.expression $\neq$ undef
    then $\mathit{entryLabel}(e.\mathsf{expression})$
  elsif $e \in$ function $\wedge$ e.expression-seq $\neq$ empty
    then $\mathit{entryLabel}(e.\mathsf{expression\text{-}seq}[1])$
  elsif $e \in$ procCall $\wedge$ e.expression-seq $\neq$ empty
    then $\mathit{entryLabel}(e.\mathsf{expression\text{-}seq}[1])$
  else $\mathit{unique}(e)$

Given these domains and functions, we can define the compilation function and the entry labels as below.

4.2 Compilation of Expressions

The compilation of expressions is the main part of the compilation function. We present below a part of it in order to take the idea clear. There is nothing complicated to the omitted parts. We have introduced appropriate comments marked with '//'.

$compileExpr(e : \mathsf{expression}, next : Label) \equiv$
  if $e \in$ variable // *evaluate the variable*
    then $\{\mathbf{mk}\text{-}Instruction(unique(e), \mathbf{mk}\text{-}Pvar(e.\mathsf{Name}, next))\}$
  elsif $e \in$ ifExpr // *first evaluate the test, then switch to then or to else*
      then $compileExpr(e.\text{test-expression}, unique(e)) \cup$
        $\{ \mathbf{mk}\text{-}Instruction(unique(e),$
          $\mathbf{mk}\text{-}Pif(unique(e.\text{test-expression}),$
          $entryLabel(e.\text{then-expression}), elseLabel)\} \cup$
        $compileExpr(e.\text{then-expression}, next) \cup$
        $compileExpr(e.\text{else-expression}, next)$
      else ...
where
$elseLabel \equiv$
if e.else-expression = undef then $next$
          else $entryLabel(e.\text{else-expression})$

4.3 Compilation of Specifications

The compilation of a specification is just the compilation of its main expression. The entry label of a specification is the entry label of its main expression.

$compileSpec(s : \mathsf{sslSpecification}) \equiv$
  $compileExpr(s.\text{main-expression}, \mathsf{undef})$
$entrySpec(p : \mathsf{sslSpecification}) \equiv$
  $entryLabel(s.\text{main-expression})$

4.4 Compilation of Procedures

The compilation of a procedure is just the compilation of its main expression. The entry label of a procedure is the entry label of its main expression.

$compileProc(p : \mathsf{procedureDef}) \equiv$
  $compileExpr(p.\text{main-expression}, \mathsf{undef})$
$\forall p \in \mathsf{procedureDef}$:
  $entryProc(p.\mathsf{Name}) = entryLabel(p.\text{main-expression})$

The simplified result of the compilation of the sample SSL specification (figures 2 and 3) can be found below in figures 4 and 5.
Apart from the generated code, the following equations hold:
specEntry(`Hanoi`) = *H0* and *procEntry*(`'hanoi'`) = *h0*.

5 SSLM: An Abstract Machine for SSL

In this section we introduce an abstract machine for SSL called SSLM, corresponding to the SDL Abstract Machine. The intention of our approach is to compile each SSL-instruction to a set of "simple" SSLM instructions, and to

Label	Primitive	Next	Comment
H0	fun(4,$\langle\rangle$)	H1	*value of 4*
H1	assign(x,H0)	H2	*x:=4*
H2	fun(0,$\langle\rangle$)	H3	*value of 0*
H3	assign(y,H2)	H4	*y:=0*
H4	fun(0,$\langle\rangle$)	H5	*value of 0*
H5	assign(z,H4)	H6	*z:=0*
H6	fun(4,$\langle\rangle$)	H7	*value of 4*
H7	assign(n,H6)	H8	*n:=4*
H8	var(n)	H9	*value of n*
H9	proc(hanoi,$\langle$H8$\rangle$,$\langle$x,y,z$\rangle$)	undef	*hanoi(x,y,z,n)*

Fig. 4. Compilation of Specification Hanoi

give these "simple" instructions a rigorous meaning. Like in [GH93], and [BS98], we present the definition of SSLM in several steps. In each step we add some details of a particular aspect of the language SSL. After introducing some basics, we present in the first step the semantics of "simple" primitives like the evaluation of assignments. In a second step we add the meaning of procedure calls and returns. In the last step we add the distribution to SSLM.

5.1 Basics

In this section we describe how states are realized in SSLM. Usually, a state is a mapping of variable names to values. In SSLM locality and scope of variables are given by a model in which variable names are mapped to (memory) locations containing values. This model allows for call-by-value and call-by-reference semantics, i.e. procedures can have reference and value parameters. To this aim we introduce a dynamic domain *Location* for locations and dynamic, controlled functions

$$\begin{aligned} &loc : \mathsf{VarName} \rightarrow Location \\ &cont : \mathsf{Location} \rightarrow Value. \end{aligned}$$

We use here and in the following $\mathsf{VarName}$, $\mathsf{FuncName}$, and $\mathsf{ProcName}$ as synonyms for Name. We introduce the following auxiliary static functions in order to evaluate conditions. In the SDL context, these would be part of the data interface.

$$\begin{aligned} &isTrue : Value \rightarrow Boolean \\ &equal : Value \times Value \rightarrow Value \end{aligned}$$

To simplify matters, we use in the following a static, controlled function

$$splmInstruction : Label \rightarrow Primitive$$

which treats the sets of pairs returned by the compilation functions as a function. This is possible since the compilation functions always compute a graph of a function. Furthermore, we introduce a kind of program counter, modeled by the function

Label	Primitive	Next	Comment
h0	var(n)	h1	*value of n*
h1	fun(0,⟨⟩)	h2	*value of 0*
h2	eq(h0,h1)	h3	*n=0*
h3	if(h2,h4)	h13	*if n=0 then ...*
h4	var(x)	h5	*value of x*
h5	fun(1,⟨⟩)	h6	*value of 1*
h6	fun(-,⟨h4,h5⟩)	h7	*x-1*
h7	assign(x,h6)	h8	*x:=x-1*
h8	var(z)	h9	*value of z*
h9	fun(1,⟨⟩)	h10	*value of 1*
h10	fun(+,⟨h8,h9⟩)	h11	*z+1*
h11	assign(z,h10)	h12	*z:=z+1*
h12	return	—	*return*
h13	var(n)	h14	*value of n*
h14	fun(1,⟨⟩)	h15	*value of 1*
h15	fun(-,⟨h13,h14⟩)	h16	*n-1*
h16	proc(hanoi,⟨h15⟩,⟨x,z,y⟩)	h17	*hanoi(x,z,y,n-1)*
h17	fun(1,⟨⟩)	h18	*value of 1*
h18	proc(hanoi,⟨h17⟩,⟨x,z,y⟩)	h19	*hanoi(x,y,z,1)*
h19	var(n)	h20	*value of n*
h20	fun(1,⟨⟩)	h21	*value of 1*
h21	fun(-,⟨h19,h20⟩)	h22	*n-1*
h22	proc(hanoi,⟨h21⟩,⟨y,x,z⟩)	h23	*hanoi(y,x,z,n-1)*
h23	return	—	return

Fig. 5. Compilation of Procedure hanoi

$$currentLabel :\to Label,$$

which returns the label associated with the instruction to be executed next. During the dynamic execution of a program intermediate values can be stored at labels. More precisely, we introduce the dynamic, controlled function

$$labelValue : Label \to Value,$$

for this purpose. In case of a sequence $seq = (l_1, \ldots l_n)$ of labels we simply write $labelValues(seq)$ for the sequence $(labelValue(l_1), \ldots, labelValue(l_n))$.
Function names are evaluated by a static function that is provided by the data semantics of the language. This would again be part of the data interface in the case of SDL.

$$evalFunc : \mathsf{FuncName} \times Value^* \to Value$$

5.2 Simple Primitives

In this section we describe the semantics of simple primitives of SSLM. These primitives are used for the evaluation of expressions and conditions.

Functions Constant names and function names are evaluated with the macro EvalFunc. This macro requires three formal parameters: a function name *funcName*, a sequence of labels *labels*, and a continue label *next*. The intention is that values stored at *labels* are extracted with function *valueLabel*. Function *funcName* is evaluated with these values, and the resulting value is stored at *currentLabel*.

$\mathsf{EvalFunc}(funcName, labels, next) \ \equiv$
  $\mathit{labelValue}(\mathit{currentLabel}) :=$
  $\mathit{evalFunc}(funcName, \mathit{labelValues}(labels)),$
  $\mathit{currentLabel} := next$

Variables Variable names are evaluated with the macro EvalVar in a similar way as function names.This macro requires two formal parameters: a variable name *varName*, and a continue label *next*.

$\mathsf{EvalVar}(varName, next) \ \equiv$
  $\mathit{labelValue}(\mathit{currentLabel}) := \mathit{cont}(\mathit{loc}(varName)),$
  $\mathit{currentLabel} := next$

Assignments Assignments are evaluated with the macro EvalAssign. This macro requires three formal parameters: a variable name *varName*, a value label *valueLabel*, and a continue label *next*. Macro EvalAssign essentially sets the value of variable name *varName* to the value stored in *label* and jumps to continue label *next*.

$\mathsf{EvalAssign}(varName, label, next) \ \equiv$
  $\mathit{cont}(\mathit{loc}(varName)) := \mathit{labelValue}(label),$
  $\mathit{currentLabel} := next$

Conditions Conditions are evaluated with the macro EvalCondition which essentially evaluates the value stored in *conditionLabel* and, depending on this value, jumps to *thenLabel* or *elseLabel*.

$\mathsf{EvalCondition}(conditionLabel, thenLabel, elseLabel) \ \equiv$
  $\mathsf{if}\ \mathit{isTrue}(\mathit{labelValue}(conditionLabel))$
    $\mathsf{then}\ \mathit{currentLabel} := thenLabel$
    $\mathsf{else}\ \mathit{currentLabel} := elseLabel$

Macro EvalEq checks whether values stored in *lhs* and *rhs* are equal.

$\mathsf{EvalEq}(lhs, rhs, next) \ \equiv$
  $\mathit{labelValue}(\mathit{currentLabel}) := \mathit{equal}(\mathit{labelValue}(lhs), \mathit{labelValue}(rhs)),$
  $\mathit{currentLabel} := next$

Macro EvalCopy saves the value stored in *label* as value of *currentLabel*.

$\mathsf{EvalCopy}(label, next) \ \equiv$
  $\mathit{labelValue}(\mathit{currentLabel}) := \mathit{labelValue}(label),$
  $\mathit{currentLabel} := next$

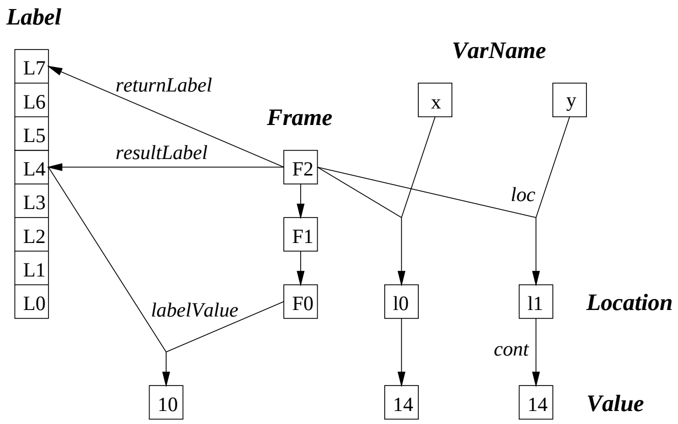

Fig. 6. State and Control Information

5.3 Procedures

Before we can define macros for procedure call and return we must have some means for storing multiple values of several incarnations of a function. To this aim we introduce frames which are organized as stacks. The universe *Frame* comprises at each point of time a finite set of frames which initially contains one element. A dynamic, controlled function *topFrame* $:\to$ *Frame* indicates the current top of the stack. The function *prev* : *Frame* $\to$ *Frame* returns for each frame its predecessor (if there is any). The following dynamic, controlled functions are introduced for the procedure call and return mechanism. The former function corresponds to a kind of return address, the latter gives the information at which label the result of a procedure must be stored.

$$\begin{array}{l} \mathit{returnLabel} : \mathit{Frame} \to \mathit{Label} \\ \mathit{resultLabel} : \mathit{Frame} \to \mathit{Label} \end{array}$$

We change the declaration of function *loc* to

$$\mathit{loc} : \mathsf{Name} \times \mathit{Frame} \to \mathit{Value}.$$

This means we have to rewrite almost every rule defined so far. Instead of doing so we simply state that every previous occurrence of $f(x)$ should be replaced by $f(x, \mathit{topFrame})$, where f is one of the functions $\{\mathit{loc}, \mathit{labelValue}\}$ (cf. figure 6). For example, rule EvalVar becomes now

$$\begin{array}{l} \mathsf{EvalVar}(varName, next) \ \equiv \\ \quad \mathit{labelValue}(\mathit{currentLabel}, \mathit{topFrame}) := \\ \quad \mathit{cont}(\mathit{loc}(varName, \mathit{topFrame})), \\ \quad \mathit{currentLabel} := next \end{array}$$

Macro EvalCall requires four formal parameters: a procedure name *procName*, a sequence of labels *labels*, a sequence of variable names *varNames*, and a return label *label*. A procedure call leads to the creation of a new frame. The new frame is pushed on the stack of frames, and return-label and result-label for this frame are set. New Variable bindings are established by a parameter passing mechanism or by the creation of local variables.

$\mathsf{EvalCall}(procName, labels, varNames, label) \;\equiv$
$\quad \mathsf{extend}\ \mathit{Frame}\ \mathsf{with}\ frame$
$\qquad \mathsf{PushNewFrame}(frame, label),$
$\qquad \mathsf{ParameterPassing}(procName, labels, varNames, frame),$
$\qquad \mathsf{CreateLocalVars}(procName, topFrame),$
$\qquad \mathit{currentLabel} := \mathit{procEntry}(procName)$

Macro PushNewFrame pushes *frame* on the stack of frames, and updates *returnLabel* and *resultLabel*.

$\mathsf{PushNewFrame}(frame, label) \;\equiv$
$\quad \mathit{topFrame} := frame, \mathit{prev}(frame) := \mathit{topFrame},$
$\quad \mathit{returnLabel}(frame) := label, \mathit{resultLabel}(frame) := \mathit{currentLabel}$

Macro ParameterPassing realizes the parameter passing mechanism, i.e. it handles passing of value and reference parameters.

$\mathsf{ParameterPassing}(procName, labels, varNames, frame) \;\equiv$
$\quad \mathsf{PassRefParameters}(procName, varNames),$
$\quad \mathsf{PassValParameters}(procName, labels),$
$\quad \mathsf{StaticBinding}(procName, frame)$

Macro PassRefParameters has two formal arguments: the procedure name *procName* and the sequence of actual reference parameters *varNames*. This macro realizes the parameter passing mechanism for reference parameters. This means references of variable names to locations in current state are changed appropriately. We use a function $\mathit{refPar} : \mathsf{ProcName} \times \mathit{Nat} \rightarrow \mathsf{VarName}$ to determine reference parameters. Function $|.|$ denotes the length function for sequences.

$\mathsf{PassRefParameters}(procName, varNames) \;\equiv$
$\quad \mathsf{forall}\ i \in \{1, ..., |varNames|\}\ \mathsf{do}$
$\qquad \mathit{loc}(\mathit{refPar}(procName, i), \mathit{topFrame}) :=$
$\qquad \mathit{loc}(varNames(i), \mathit{topFrame})$

Macro PassValParameters creates for each value parameter a new location. Each label in the sequence *labels* has a value in frame. We write simply $\mathit{labels}(i)$ to project on the i-th component of this sequence. The value of a new location *location* is set to the value stored in *labels* and frame where i ranges over $\{1, \ldots, |labels|\}$. The location for the i-th value parameter given by a function $\mathit{valPar} : \mathsf{ProcName} \times \mathit{Nat} \rightarrow \mathsf{VarName}$ is set to a new location. We assume that the size of *labels* equals the number of value parameters.

$\mathsf{PassValParameters}(procName, labels) \;\equiv$
$\quad \mathsf{forall}\ i \in \{1, ..., |labels|\}\ \mathsf{do}$

$\textsf{extend } Location \textsf{ with } location$
$\quad cont(location) := labelValue(labels(i), topFrame),$
$\quad loc(valPar(procName, i), topFrame) := location$

Let p be a procedure name. We introduce a static function

$$binding : \mathsf{VarName} \times \mathsf{ProcName} \rightarrow \mathsf{ProcName}$$

which returns for each variable name and each procedure name the name of a procedure to which variable name is bound according to the static scope rules. Function $findLocation : \mathsf{VarName} \times \mathsf{ProcName} \times Frame \rightarrow Location$ evaluates for a given variable name x the stack starting with $frame$ in order to determine the first frame F such that the equation $binding(x, \mathsf{procName}) = proc(F)$ holds. The dynamic, controlled function $proc : Frame \rightarrow \mathsf{ProcName}$ returns for each frame the name of the corresponding procedure.

$\mathsf{findLocation}(varName, procName, frame) \equiv$
$\quad \textsf{if } binding(varName, procName) = proc(frame)$
$\quad\quad \textsf{then } loc(varName, frame)$
$\quad\quad \textsf{else } findLocation(varName, procName, prev(frame))$

Macro StaticBinding saves *procName* in *frame* creates a variable binding for those variables which are not defined in the body of *procName*.

$\mathsf{StaticBinding}(procName, frame) \equiv$
$\quad proc(frame) := procName,$
$\quad \textsf{forall } varName \in \mathsf{VarName} \textsf{ do}$
$\quad\quad \textsf{if } binding(varName, procName) \neq procName$
$\quad\quad\quad \textsf{then } loc(varName, topFrame) :=$
$\quad\quad\quad\quad findLocation(varName, procName, topFrame)$

Macro CreateLocalVars creates new location for local variables associated with a procedure or a program. Local variables are determined by a means of a static function $locVars : \mathsf{Name} \rightarrow \mathsf{Name\text{-}set}$.

$\mathsf{CreateLocalVars}(name, frame) \equiv$
$\quad \textsf{forall } varName \in locVars(name) \textsf{ do}$
$\quad\quad \textsf{extend } Location \textsf{ with } location$
$\quad\quad\quad loc(varName, frame) := location$

Macro EvalReturn essentially removes the top frame of the stack, saves the result according to the result label, and jumps to the return label.

$\mathsf{EvalReturn}(label) \equiv$
$\quad topFrame := prev(topFrame),$
$\quad labelValue(resultLabel(topFrame), prev(topFrame)) :=$
$\quad\quad labelValue(label, topFrame),$
$\quad currentLabel := returnLabel(topFrame)$

5.4 Distribution

In this section we refine the SSLM defined so far to a distributed abstract machine. The underlying distribution model consists of several agents which execute their programs and communicate by means of shared variables. The refinement step is done in a similar way as in the preceding section, namely by changing the declaration of some existing functions. We provide each agent with its own program counter. To this aim we change the declaration of functions *currentLabel*, and *topFrame* to

$$currentLabel : Agent \rightarrow Label,$$
$$topFrame : Agent \rightarrow Frame.$$

Every previous occurrence of f inside macros should be replaced by $f(Self)$, where f is one of the functions $\{currentLabel, topFrame\}$.

6 Initialization and Execution

With the abstract machine for SSL being defined, we can now assign a formal semantics to SSL specifications by associating, with each specification, a set of initial states and an ASM program. Please note that conceptually, it is better to think of SSLM programs, because the instructions considered here are SSLM instructions. Formally, the programs below are ASM programs.

6.1 Initialization

In the ASM model, the set of initial states is usually defined by constraints imposed on domains, functions, and predicates. These constraints are required to hold in the first state of each run of the abstract state machine. Initial constraints are stated by formulae that are preceded by the keyword initially, i.e. in a property-oriented style.

As it turned out during work on the formal SDL semantics, a property-oriented definition of the set of initial states, given by the set of SDL agents, SDL agent sets, and link agents together with the initial values of all functions and predicates defined on these agents is rather complex and hard to follow. Therefore, a different approach has been chosen, which may be characterized as a combination of constraints and initial computations: several (simple) constraints determine the so-called *set of pre-initial states*. The actual set of initial states is determined by the initialization phase that starts in a pre-initial state, and creates a set of agents as well as the system structure.

To give a flavor of the approach taken in the formal SDL semantics, we adapt it here to define the set of initial states associated with an SSL specification. A nullary function

$$main : \rightarrow Agent$$

is introduced to refer to the pre-initial system agent. Furthermore, we define

$$topLevelSpec : \rightarrow \mathsf{sslSpec}$$
$$mySpec : Agent \rightarrow \mathsf{sslSpec}$$

The set of pre-initial states is defined by the following set of constraints (strictly speaking, the set of pre-initial states is the set of initial states of the ASM model):

$$
\begin{aligned}
&\textbf{initially } Agent = \{main\}\\
&\textbf{initially } sslmInstruction =\\
&\qquad\qquad \bigcup\{p \in \mathsf{sslSpec} : p.compileSpec\} \cup\\
&\qquad\qquad \bigcup\{p \in \mathsf{procedureDef} : p.\mathsf{proc\text{-}Name}.compileProc\}\\
&\textbf{initially } main.mySpec = topLevelSpecification\\
&\textbf{initially } main.currentLabel = topLevelSpecification.entrySpec\\
&\textbf{initially } main.Mod = \mathsf{InitModule}\\
&\textbf{initially } Frame = \{fstFrame\}\\
&\textbf{initially } main.topFrame = fstFrame
\end{aligned}
$$

The value of the static function *sslmInstruction* is determined by the compilation defined in Section 4. It is the complete code of all behaviour parts within the specification. Furthermore, the pre-initial values of certain functions are defined. The function *mySpec* associates with each agent the abstract syntax description of its specification. For agent *main*, this is defined by the nullary function *topLevelSpecification*, that is, the main specification. The start label of a specification is obtained from the function *entrySpec*, as defined in Section 4. Finally, the behaviour of agent *main* is determined by assigning the ASM module InitModule, which determines the steps of *main* during the initialization phase:

$$
\begin{aligned}
&\mathsf{InitModule} \;\equiv\\
&\quad \mathsf{forall}\ p \in Self.mySpec.\mathsf{sslSpec\text{-}seq}\ \mathsf{do}\\
&\qquad \mathsf{extend}\ Agent\ \mathsf{with}\ a\\
&\qquad\quad \mathsf{extend}\ Frame\ \mathsf{with}\ frame\\
&\qquad\qquad a.mySpec := p,\ a.topFrame := frame\\
&\qquad\qquad a.currentLabel := p.entrySpec,\ a.Mod := \mathsf{InitModule},\\
&\quad \mathsf{CreateLocalVars}(Self.mySpec.\mathsf{spec\text{-}Name},\ Self.topFrame)\\
&\quad Self.Mod := \mathsf{ExecModule}
\end{aligned}
$$

Firing of InitModule creates, for each SSL specification defined in the specification of the current agent, one agent executing the InitModule. Furthermore local variables of the corresponding SSL specification are created. This leads to the recursive unfolding of the tree of SSL specifications defined by the top level SSL specification. The description of macro CreateLocalVars can be found in section 5.3.

In the SDL semantics, it is important that the initialization phase is completed before the execution phase starts. This is achieved by a mechanism that blocks agents after their initialization until the initialization phase has ended. For simplicity, we have not modeled this mechanism in the solution presented here.

6.2 Execution of SSL specifications

The execution of an SSL specification is given by the following ASM module:

ExecModule $\equiv$
  EvalInstruction(*Self.currentLabel.sslmInstruction*)

The evaluation of an instruction is given by macro EvalInstruction.

EvalInstruction($prim$) $\equiv$
  if $prim \in Pfun$
    then EvalFunc($prim$.Name, $prim$.value-Label-seq, $prim$.next-Label)
  elsif $prim \in Pif$
    then EvalIf($prim$.condition-Label, $prim$.then-Label, $prim$.next-Label)
  elsif $prim \in Pvar$ then EvalVar($prim$.Name, $prim$.next-Label)
  elsif $prim \in Peq$ then EvalEq($prim$.lhs-Label, $prim$.rhs-Label, $prim$.next-Label)
  elsif $prim \in Pcopy$
    then EvalCopy($prim$.value-Label, $prim$.next-Label)
  elsif $prim \in Passign$
    then EvalAssign($prim$.Name, $prim$value-Label, $prim$.next-Label)
  elsif $prim \in Pcall$
    then EvalCall($prim$.Name, $prim$.value-Label-seq,
      $prim$.ref-Name-seq, $prim$.next-Label)
  elsif $prim \in Preturn$ then EvalReturn($prim$.value-Label)
  else /* does not happen */

7 Conclusions

In this paper, we have presented the transformational/operational approach which has been successfully applied to define the semantics of SDL-2000 formally. We have illustrated some basic techniques of the approach using SSL, a simple specification language.
The following topics have been covered:

- *Abstract Machine:* The semantics is based on an abstract machine. This machine is an extension of the ASM model and is specifically tailored towards the needs of the specification technique. It introduces an abstract code for the FDT and could therefore also be used for its implementation. The abstract machine comprises a set a machine instructions and their semantics as well as some domains and functions. An important part of the abstract machine is the built-in procedure model that allows for call-by-value and call-by-reference parameter passing semantics. Furthermore, the concept of static binding is built into the abstract machine.
- *Compilation:* Any program/specification of the specification technique is given a semantics by transforming it into the abstract machine. This is essentially an abstract compilation function. The compilation transforms the domain of the abstract syntax trees into the domain of abstract machine instruction sets. The execution of these sets is then covered within the description of the abstract machine.

– *Initialization:* In order to run a specification on the abstract machine, a special initialization has to be performed. To this end, the abstract syntax tree is used to determine the internal parts of the specification to be initialized. These parts are then initialized as well. This way, the initialization takes care of structural information, while the behavioural information is handled by the compilation function. The same initialization behavior could also be used for dynamic creation of specifications.

The subdivision of the semantics into the parts presented above has proved very useful in the context of SDL, as it is essential here to be able to adjust the semantics to the development of the FDT. This is achieved by using two layers of abstraction: (1) abstract the concrete syntax into an abstract syntax, (2) transform the abstract syntax to an abstract machine.
It should be noted that the formal SDL semantics has been conceived in parallel to the language definition itself. During this process, several substantial changes to the language definition, which turned out to be a "moving target", were made. These changes of course affected the formal semantics definition, but usually did not touch the SAM.
Finally, it should again be stressed that the work reported here is not an academic exercise, but takes place in a real-life, industrial setting. In our opinion, it is this kind of work academic efforts should eventually lead to. The successful application of mathematical formalism to real-world problems and their approval by industry is a strong selling point for having formalisms at all. In this sense, the work in progress reported in this paper is one important step in this direction.

References

[BS98] Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[GGP99] U. Glässer, R. Gotzhein, and A. Prinz. Towards a New Formal SDL Semantics Based on Abstract State Machines. In *SDL'99 - The Next Millennium, Proceedings of the 9th SDL Forum*. Elsevier Science B.V., 1999.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, LNCS, pages 274–308. Springer, 1993.

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur97] Yuri Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, 1997.

[Gur99] Yuri Gurevich. The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124, February 1999. Columns: Logic in Computer Science.

[PGGE] Andreas Prinz, Reinhard Gotzhein, Uwe Glässer, and Robert Eschbach. SDL Formal Semantics, Z.100.F, ITU-T, International Telecommunication Union. `www.informatik.hu-berlin.de/~prinz/Semantics/Z100F_Geneva.doc`.