# On the Formal Semantics of SDL-2000: A Compilation Approach based on an Abstract SDL Machine

Robert Eschbach[1], Uwe Glässer[2], Reinhard Gotzhein[1] and Andreas Prinz[3]

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

[2] Heinz Nixdorf Institute, University of Paderborn, D-33102 Paderborn, Germany

[3] Department of Computer Science, Humboldt-University Berlin, D-12489 Berlin, Germany

**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[1] for the development of distributed real-time systems in general,

[1] SDL is defined by the ITU-T, the Telecommunication Standardization Sector of the International Telecommunication Union (ITU), in their *Recommendation Z.100* [IT99].

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. Building on a combination of the VDM meta language Meta-IV with a CSP-like communication mechanism, the formal semantic model provides a comprehensive formalization of all static and dynamic aspects of SDL, though, it is hardly manageable because of its size[2]. 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 non-basic 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 (see [PGGE]) has been determined to become the official formal SDL semantics after its approval, which is expected for November 2000.

Among the primary design objectives of this approach is *intelligibility* and conciseness of the formal description as a prerequisite for acceptance, correctness and maintainability. Additionally, *executability* has become a key issue. Executability calls for an operational formalism with readily available tool support. For this and other reasons (cf. [GGP99]), *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 formalize 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. [GK97,Glä98]):

[2] The complete formal semantics for SDL-96, as defined in Annex F to Z.100, is documented on more than 500 pages of Meta-IV descriptions.

- 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*, SAM for short, which is defined in terms of a distributed real-time ASM. SDL specifications are then compiled into the instruction set of this machine.
- Compared to the standard technique for defining the set of initial states of an ASM model—usually this is done in an axiomatic or declarative style[3]—a realistic semantic model of SDL requires a different approach. Following the view of Z.100 on how initial system states are derived from given SDL specifications, the SAM model starts its execution from a set of so-called *pre-initial* states that are canonically defined. The actual set of initial states is then determined in an operational way as result of an *initialization phase*. 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.

Taking into account that the work reported here is not an academic exercise, but takes place in a real-life industrial setting, pragmatic compromises are unavoidable. For this reason, some decisions taken in the formalization of the SDL (and hence SSL) semantics are rather pragmatic or 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]). A preliminary version of this paper can be found in [EGP00].

This paper is organized as follows. Section 2 gives an overview on related work. Pragmatic aspects of our formalization approach are briefly discussed in Section

[3] Quite often, an initial state $S_0$ is defined through a set of first-order formulas $\varphi_1, \ldots, \varphi_n$ such that $S_0 \models \varphi_1 \wedge \ldots \wedge \varphi_n$.

3, while Section 4 provides an overview of the formal Semantics for SDL-2000. Section 5 introduces the language SSL. Section 6 defines the abstract compilation of SSL specifications to SSLM programs. Section 7 defines the SSL abstract machine (SSLM). Section 8 describes the initialization. Section 9 presents conclusions. Throughout this paper we assume the reader to be familiar with Gurevich's notion of ASM (cf. [Gur95,Gur97,Gur99]).

## 2 Related Work

Regarding the importance of SDL as a practical tool for industrial systems engineering, it is certainly not astonishing that in addition to activities within ITU, there is a considerable variety of attempts to formalize the semantics of SDL using various formal methods. According to the principal objectives behind these attempts, one can distinguish two basically different aspects, namely: (1) approaches that are mainly concerned with analysis and verification of SDL system specifications; (2) approaches that are mainly concerned with formal documentation, maintenance and validation of the SDL language definition.

### 2.1 Analysis and Verification

In [BM96], Bergstra and Middleburg define a *process algebra* semantics of a restricted version of SDL, which they call $\varphi$SDL (where $\varphi$ stands for "flat"), as it does not cover the structural aspects of SDL. Additionally, they introduce a restricted notion of time to simplify time related concepts of SDL. The authors claim to have convincing pragmatic justification for their choices; for instance, they argue that *"a dramatically simplified version of SDL"* and an adequate semantics for it *"are prerequisites for advanced analysis and formal verification"*. After all, they deploy an impressive mathematical apparatus making it difficult to see whether $\varphi$SDL is relevant in practice.
Broy [Bro91], Holz and Stølen [HS94], and Hinkel [Hin98] model various subsets of (essentially) Basic SDL using denotational semantics based on *stream processing functions* of *FOCUS* [BDD+93]. While it may be natural to model SDL process communication as discrete streams of signals, the functional view does neither support the concept of system states and state transitions nor allows the stream formalism for an adequate treatment of time. Even the most comprehensive model [Hin98] builds on a fundamentally restricted notion of *global system time* not allowing to express time quantities explicitly. In particular, it is not possible to specify any quantitative delays as required for dealing with the timer concept of SDL. Since the expiration time of a timer does not refer to a global system time (as represented by the SDL expression **now**), the meaning of an active timer is reduced to the fact that a timer signal will eventually be generated (after some finite delay). This results in a severe oversimplification.[4]

[4] It is certainly right that the current definition of SDL needs further improvements allowing a more detailed and more precise inspection of timing behavior (see for

Fischer, Dimitrov and Tauber propose an extended Petri Net model, so-called *SDL Time Nets*, as a formal basis to verify SDL protocol specifications [FD95], [FDT95]. The transformation of SDL specifications into corresponding net models as well as the transformation of results back to the SDL level is done automatically within the SDL Integrated Tools Environment *SITE*. Though these transformations, in principle, remain invisible for the user, the authors concede that certain knowledge is necessary about the internal data structures of the analysis tools. Moreover, the integration of further net analysis algorithms is difficult and requires detailed knowledge about the internals (like interfaces, structure and representation formats) of the tools.

### 2.2 Documentation and Validation

An ambitious attempt aiming at a comprehensive formal semantics for SDL-92 as a basis for further development of SDL was started by Lau and Prinz with their definition of BSDL (*Base SDL*) [LP95]. With the ultimate goal to simplify the language definition, they proposed a modeling approach using *Object-Z* to define a universal *core* for SDL as a conceptual framework to deal with the main building blocks of the language. This core should be as clear and concise as possible, yet it should include the basic model of processes and communication, all the object-oriented concepts and partly also structuring concepts. The meaning of additional language constructs (not handled within the core) should then be fixed by defining their transformation to the core. In that respect, the BSDL approach is similar to the original formal model of SDL, but suggests a more systematic and robust construction of a formal semantic model. However, the Object-Z definition of BSDL was rejected by the ITU even before the work was completed.
Regarding further developments of SDL towards SDL-2000, an approach based on *transition systems* generated from attributed abstract syntax trees (ASTs) is outlined in [GGRS98]. An attribute grammar is formed by adding evaluation rules to the abstract grammar of the language, where attributes for instance may represent *actions*. This way, one obtains for each complete SDL specification an attributed AST containing all the information required to generate a behavior model. The underlying mathematical model and the notation are not determined, which leaves room for subsequent choices.
In [GK97] and [Glä98], an ASM behavior model for Basic SDL-92 is defined. Starting from an abstract operational view, the dynamic properties of Basic SDL-92 are formalized in terms of an *abstract SDL interpreter* based on a *distributed real-time ASM* model. This work provides a conceptual framework that has further been developed and extended by combining it with the compiler-based approach from [GGRS98], as well as certain BSDL concepts from [LP95],

instance [MT98]), e.g. to determine delays caused by time consuming operations; on the other hand, any reasonable approach to overcome these deficiencies cannot mean to replace the current time concept by an even more primitive one, as done in [Hin98].

resulting in a robust formal basis for the definition of an ASM behavior model for SDL-2000, as discussed here.

## 3 Pragmatic Foundation

This section addresses fundamental questions concerning the practicability, suitability and robustness of our formalization approach. The primary focus here is on pragmatic aspects rather than on technical details. That is, we try to explain at a conceptual level distinctive features of our SDL behavior model. In particular, we try to answer the question, "Why and how does the abstract operational view of the ASM method in combination with the compiler approach practiced here naturally meet the demands on formal documentation, language maintenance and tool development as arising from the SDL standardization process?". Industrial engineering of complex distributed systems based on SDL, as already outlined in Sect. 1, has quite a tradition in telecommunication technology. Meanwhile, one can as well observe an increasing proliferation of SDL applications in various fields of information technology (e.g., engineering of distributed control systems). Like the development of SDL over its life cycle, this progress has been influenced by the apparent convergence of information technology and communication technology.
To meet the needs of system design experts, the language has been continuously improved over a period of more than 20 years, evolving from a primitive graphical notation to a sophisticated formal description technique, according to the demands on industrial systems technology. Consequently, the underlying language concepts closely reflect the view and the intuitive understanding of system behavior and system structure of those who have mainly contributed (through their practical experience) to the development of SDL (cf. the standard literature, e.g. [OFMP+94,EHS97]). Hence, in order to gain acceptance, any realistic attempt to formalize the semantics of SDL with the intention to facilitate language design and validation of implementations therefore cannot succeed without supporting the intuitive understanding of well established SDL concepts (including functional, structural and timing aspects) as defined in Z.100.

### 3.1 Abstract Operational View

The problem of "turning English into mathematics" is considerably simplified by a close correspondence between the specification method and the semantic model to be formalized. In fact, this allows for more natural and adequate abstractions resulting in more comprehensible and more reliable descriptions.[5] Given the definition of SDL in Z.100, it makes perfect sense to formalize dynamic properties of SDL specifications as abstract machine runs since this style of modeling directly

[5] For instance, was it possible to eliminate a number of cumbersome in-language transformations (specified by so-called *Model Sections*) from Z.100 by defining the meaning of certain non-basic SDL constructs directly in the formal behavior model.

reflects the abstract operational view of the informal language description. Conceptually, the effect of actions and events as associated with *active* SDL objects (i.e., those objects that have a behavior) on the global system state is stated in terms of the effect of SAM operations on abstract machine states.
More specifically, the resulting abstraction depends on the *granularity* of abstract machine operations specified by the SAM instruction set. Ideally, this granularity is such that all relevant effects on the system state are highlighted, whereas minor operational details remain hidden in the functions and predicates on top of which these operations are formulated. Additional details may then be introduced through stepwise refinements such that the style of representation gradually turns from declarative to operational while crossing abstraction levels.[6]
Essential for a direct and concise encoding of SDL behavior primitives in ASM notation is of course a close correspondence of the underlying computation models. In that respect, there are two observations concerning our modeling approach:

- The SDL view of distributed systems with real-time constraints and the semantic modeling concept of distributed real-time ASM clearly coincide. That is, essential properties of the underlying computation models – namely, the notions of concurrency, reactivity and time as well as the notion of state – are so tightly related that the common understanding of SDL can directly be converted into a formal semantic model avoiding any formalization overhead.
- Even without direct support of object-oriented features in ASMs, the resulting ASM model of the dynamic semantics is particularly concise, readable and understandable. Furthermore, this model can easily be extended and modified as required for an evolving technical standard. Beyond the purpose addressed here, the model may as well be utilized for defining a *bridging semantics* in order to combine SDL with other modeling languages (e.g., UML and VHDL).

### 3.2 Correctness and Reliability

Defining semantics by a mapping to a well defined semantic basis, as realized by the compilation of SDL specifications into SAM code, is a well-known technique, but is often considered to be as dangerous as helpful. Of course one may argue that this leads to the old "the compiler defines the language semantics" problem: "Who validates and/or verifies the correctness of this mapping and with respect to which origin?".

[6] An illustrative example is the abstract definition of the behavior model for SDL channels (see [PGGE]). By exploiting the decentralized organization of the underlying signal flow model, the entire static information on the system structure and the reachability constraints are completely encapsulated into a predicate *compatible*, which can then be refined by introducing technical details on how this information is to be derived from a given specification.

Now, the question here is: "How can one establish that a *formal* semantic model like the ASM behavior model of SDL faithfully reflects the intuitive understanding of SDL, i.e. yields a valid interpretation of the definitions in the language reference manual?". Since there is no way of proving correctness (in a strict mathematical sense), the approach taken here is to construct an ASM behavior model that reflects the dynamic properties of the language so closely that correctness can be established by observation and experimentation. In ASM terms, such a model is called a *ground model*, cf. [Bör99]. The quality of ground models considerably depends on the underlying abstractions for dealing with real-world phenomena, i.e. on the way in which basic objects and operations of a ground model are related to basic entities and actions as observed in the real world.
Investigating the construction of ground models (as part of the analysis and design process), one can identify general principles for justifying their appropriateness and validity. In [Bör99], Egon Börger convincingly argues that this justification process has three basic dimensions, namely: a conceptual justification, an experimental justification, and a mathematical justification. According to this understanding, the best we can achieve is a solid pragmatic foundation.

## 4 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* 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 basic 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.

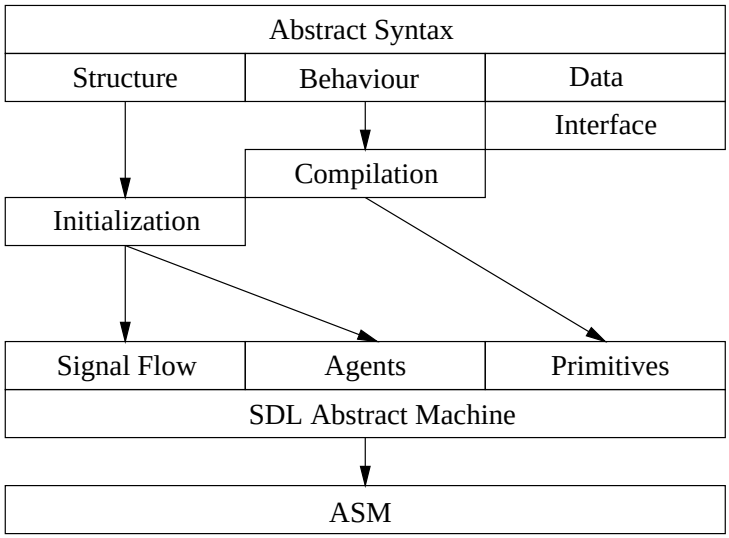


**Fig. 1.** Overview of the Dynamic Semantics

- 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.
- An *SDL Abstract Machine Program* defines a 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 SAM 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* 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, thus 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. The dynamic semantics associates, with each SDL specification, a particular distributed, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents cooperatively performing concurrent machine runs. This approach differs substantially from the interpreter view taken in the previous SDL semantics definition, and will enable SDL-to-ASM compilers. Conceptually, one can identify three basic layers at which our semantic model of SDL is actually defined (either explicitly or implicitly), namely:

(2) the compilation function,
(1) the SAM architecture, and
(0) the semantic model of distributed real-time ASM.

Regarding layers (2),(1), it is not a priori fixed which aspects of behavior are subject to modeling at which one of these two layers. Though, in principle, specific aspects depending on a given SDL specification typically ought to be encoded into the compilation function, whereas invariant dynamic properties are more naturally expressed in terms of the SAM model. Finally, to keep the formalization as concise and elegant as possible, all aspects that both computation models (SDL and ASM) have in common are preferably given implicitly by mapping SDL behavior primitives directly onto corresponding primitives of the underlying ASM model at layer (0).
By splitting the definition of the SDL behavior model into the above layers, one obtains a flexible framework for semantic modeling that allows for natural and adequate abstractions, transparent and well defined formalization, and flexible rearrangements on the basis of an executable semantics.

## 5 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.

### 5.1 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.

### 5.2 Specifications and Procedures

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.

sslSpec ::= spec-Name variableDecl* sslSpec* procedureDef*
main-expression
variableDecl ::= Name

```
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

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

procedureDef ::= proc-Name refParam* valParam* variableDecl*
procedureDef* main-expression
refParam ::= Name
valParam ::= Name

### 5.3 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.

```
expression ::= function | ifExpr | variable | equality | sequence |
               assignment | procCall | procReturn
  function ::= Name expression*
     ifExpr ::= test-expression then-expression [else-expression]
  variable ::= Name
  equality ::= lhs-expression rhs-expression
  sequence ::= fst-expression nxt-expression
assignment ::= Name expression
  procCall ::= Name ref-Name* expression*
procReturn ::= expression
```

Please note, that the syntax domain names are also used as the constructors for the syntax tree. A sample SSL specification can be found in figure 2.

## 6 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 7.

### 6.1 Basics

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

$compileSpec$ : sslSpecification $\rightarrow$ $AbstractCode$
$compileProc$ : procedureDef $\rightarrow$ $AbstractCode$

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 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 and an instruction a tuple consisting of a label and a primitive.

$AbstractCode = Instruction\text{-}\mathbf{set}$
$Instruction = Label \times Primitive.$

The compiled code of all specifications and procedures is merged together in the initial state. Specifications and procedures are characterized by the functions

$entrySpec : \mathsf{sslSpecification} \to Label,$
$entryProc : \mathsf{Name} \to Label,$

which return the entry labels. For SSL we introduce the following primitives. Again, the prefixes are without semantic relevance.

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

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

$unique : \mathsf{expression} \to Label$

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

$entryLabel : \mathsf{expression} \to Label$

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.

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

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

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

### 6.3 Compilation of Specifications and Procedures

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.

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

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.

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

The simplified result of the compilation of the sample SSL specification (see figure 2) without procedure hanoi can be found below in figure 3. Apart from the generated code, the following equation hold:
$specEntry($`Hanoi`$) = H0$

## 7 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,⟨⟩) | H1 | *value of 4* |
| H1 | assign(x,H0) | H2 | *x:=4* |
| H2 | fun(0,⟨⟩) | H3 | *value of 0* |
| H3 | assign(y,H2) | H4 | *y:=0* |
| H4 | fun(0,⟨⟩) | H5 | *value of 0* |
| H5 | assign(z,H4) | H6 | *z:=0* |
| H6 | fun(4,⟨⟩) | H7 | *value of 4* |
| H7 | assign(n,H6) | H8 | *n:=4* |
| H8 | var(n) | H9 | *value of n* |
| H9 | proc(hanoi,⟨H8⟩,⟨x,y,z⟩) | undef | *hanoi(x,y,z,n)* |

**Fig. 3.** 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.

### 7.1 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

$$loc : \mathsf{VarName} \to Location$$
$$cont : \mathsf{Location} \to Value.$$

We use here and in the following VarName, FuncName, and ProcName as synonyms for Name. We use in the following a static, controlled function

$$splmInstruction : Label \to Primitive,$$

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

$$currentLabel :\to Label,$$

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.

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

### 7.2 Simple Primitives

In this section we describe the semantics of simple primitives of SSLM. These primitives are used for the evaluation of expressions, assignments, and conditions. The evaluation of expressions is realized by macros EvalFunc and EvalVar. 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 $currentLabel$.

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

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 $next$.

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

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 $next$.

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

Furthermore, there exists a macro EvalCondition which evaluates conditions. Due to its simplicity we omit the corresponding definition.

### 7.3 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 $topFrame :\rightarrow Frame$ indicates the current top of the stack. The function $prev : Frame \rightarrow Frame$ 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.

$returnLabel : Frame \rightarrow Label$
$resultLabel : Frame \rightarrow Label$

We change the declaration of function *loc* to

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

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 $f(x, topFrame)$, where $f$ is one of the functions $\{loc, labelValue\}$ (cf. figure 4). For example, rule EvalVar becomes now

EvalVar$(varName, next) \equiv$
  $labelValue(currentLabel, topFrame) :=$
  $cont(loc(varName, topFrame))$,
  $currentLabel := next$

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.

EvalCall$(procName, labels, varNames, label) \equiv$
  extend $Frame$ with $frame$
    PushNewFrame$(frame, label)$,
    ParameterPassing$(procName, labels, varNames, frame)$,
    CreateLocalVars$(procName, topFrame)$,
    $currentLabel := procEntry(procName)$

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

PushNewFrame$(frame, label) \equiv$
  $topFrame := frame, prev(frame) := topFrame,$
  $returnLabel(frame) := label, resultLabel(frame) := currentLabel$

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

ParameterPassing$(procName, labels, varNames, frame) \equiv$
  PassRefParameters$(procName, varNames)$,
  PassValParameters$(procName, labels)$,
  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 $refPar : \mathsf{ProcName} \times Nat \rightarrow \mathsf{VarName}$ to determine reference parameters. Function $|.|$ denotes the length function for sequences.

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

Macro PassValParameters creates for each value parameter a new location. Each label in the sequence $labels$ has a value in frame. We write simply $labels(i)$ 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 $valPar : \mathsf{ProcName} \times Nat \rightarrow \mathsf{VarName}$ is set to a new location. We assume that the size of $labels$ equals the number of value parameters.

PassValParameters$(procName, labels)$ $\equiv$
  forall $i \in \{1, ..., |labels|\}$ do
    extend $Location$ with $location$
      $cont(location) := labelValue(labels(i), topFrame)$,
      $loc(valPar(procName, i), topFrame) := location$

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.

findLocation$(varName, procName, frame)$ $\equiv$
  if $binding(varName, procName) = proc(frame)$
    then $loc(varName, frame)$
    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 $procName$.

StaticBinding$(procName, frame)$ $\equiv$
  $proc(frame) := procName$,
  forall $varName \in \mathsf{VarName}$ do
    if $binding(varName, procName) \neq procName$

then $loc(varName, topFrame) :=$
$findLocation(varName, procName, topFrame)$

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 $locVars$ : Name $\rightarrow$ Name-**set**.

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

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

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

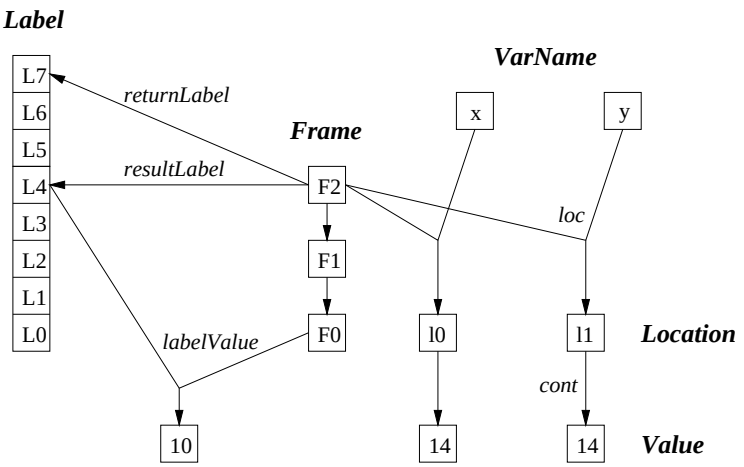


**Fig. 4.** State and Control Information

### 7.4 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 $\{currentLabel, topFrame\}$.

# 8 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.

## 8.1 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

$$topLevelSpec :\rightarrow \mathsf{ss|Spec}$$
$$mySpec : Agent \rightarrow \mathsf{ss|Spec}$$

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):

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

The value of the static function *sslmInstruction* is determined by the compilation defined in Section 6. It is the complete code of all behavior 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 6. Finally, the behavior of agent *main* is determined by assigning the ASM module InitModule, which determines the steps of *main* during the initialization phase:

```
InitModule ≡
  forall p ∈ Self.mySpec.sslSpec-seq do
     extend Agent with a
        extend Frame with frame
           a.mySpec := p, a.topFrame := frame
           a.currentLabel := p.entrySpec, a.Mod := InitModule,
  CreateLocalVars(Self.mySpec.spec-Name, Self.topFrame)
  Self.Mod := ExecModule
```

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 a recursive unfolding of the tree of SSL specifications defined by the top level specification. Macro CreateLocalVars is defined in section 7.3.
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.

### 8.2 Execution of SSL specifications

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

ExecModule ≡ EvalInstruction(*Self.currentLabel.sslmInstruction*)

The evaluation of an instruction is given by macro EvalInstruction. Depending on the type of an instruction a corresponding macro is executed. For example, an instruction of type *Passign* leads to the execution of macro EvalAssign as illustrated in the following.

```
EvalInstruction(p) ≡
  if ...
  elsif p ∈ Passign
      then EvalAssign(p.Name, pvalue-Label, p.next-Label)
  elsif p ∈ Pcall
      then EvalCall(p.Name, p.value-Label-seq, p.ref-Name-seq, p.next-Label)
  ...
```

## 9 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 behavioral 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.

**Acknowledgment.** We thank the anonymous referees and the organizers of the ASM2000 workshop for their constructive criticism on a preliminary version of this paper.

## References

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems – an introduction to FOCUS. Technical Report TUM-19202-2 (SFB-Bericht Nr. 342/2-2/92/A), Institut für Informatik, Technische Universität München, January 1993.

[BM96] J. A. Bergstra and C. A. Middleburg. Process Algebra Semantics of $\varphi$SDL. Technical Report UNU/IIST Report No. 68, UNU/IIST, The United Nations University, April 1996.

[Bör99] E. Börger. High Level System Design and Analysis using Abstract State Machines". In Traverso Ullman Hutter, Stephan, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS. Springer, 1999.

[Bro91] M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing 3*, (3):21–57, 1991.

[BS98] E. Börger and W. 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.

[EGP00] R. Eschbach, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach using Abstract State Machines. In *Local Proc. of ASM2000, Monte Verità, Switzerland*, 2000.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe, 1997.

[FD95] J. Fischer and E. Dimitrov. Verification of SDL protocol specifications using extended Petri Nets. In *Proc. of the Workshop on Petri Nets and Protocols of the 16th Intern. Conf. on Application and Theory of Petri Nets*, pages 1–12. Torino, Italy, 1995.

[FDT95] J. Fischer, E. Dimitrov, and U. Taubert. Analysis and formal verification of SDL'92 specifications using extended Petri Nets, 1995. internal report.

[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, Proc. of the 9th SDL Forum*. Elsevier Science B.V., 1999.

[GGRS98] R. Gotzhein, B. Geppert, F. Rößler, and P. Schaible. Towards a new formal SDL semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 55–64, 1998.

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

[GK97] U. Glässer and R. Karges. Abstract State Machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.

[Glä98] U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 2, pages 271–280, 1998.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford Univ. Press, 1995.

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

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

[Hin98] U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*. Ph.D., Techn. Univ. München, 1998.

[HS94] E. Holz and K. Stølen. An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus. In St. Leue D. Hogrefe, editor, *Proc. of Forte '94*, pages 324–339. Chapmann & Hall, 1994.

[IT99] ITU-T. Recommendation Z.100: Specification and Description Language (SDL), International Telecommunication Union (ITU). Geneva, 1999.

[LP95] St. Lau and A. Prinz. BSDL: The Language – Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995.

[MT98] A. Mitschele-Thiel. Performance evaluation of SDL systems. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 35–44, 1998.

[OFMP+94] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. Elsevier Science B. V., 1994.

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