The formal semantics of SDL-2000: Status and perspectives

U. Glässer a, R. Gotzhein b,*, A. Prinz c

a School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6
b Department of Computer Science, University of Kaiserslautern, FB Informatik, Postfach 3049, D-67653 Kaiserslautern, Germany
c DResearch Digital Media Systems GmbH, D-10319 Berlin, Germany

Abstract

In November 1999, the current version of specification and description language (SDL), commonly referred to as SDL-2000, passed through ITU-T. In November 2000, the formal semantics of SDL-2000 was officially approved to become part of the SDL language definition. It covers both the static and the dynamic semantics, and is based on the formalism of abstract state machines (ASMs). To support executability, the formal semantics defines, for each SDL specification, reference ASM code, which enables an SDL-to-ASM-compiler.

In this paper, we briefly survey and compare existing approaches to define the semantics of SDL formally. The ITU-T approach is then outlined in more detail, addressing the following steps: (1) mapping of non-basic language constructs to the core language, (2) checking of static semantics conditions, (3) definition of the SDL abstract machine (SAM), and (4) definition of the SDL virtual machine (SVM). The paper concludes with experiences from the SDL-to-ASM-compiler project. It is proposed that the SDL-2000 semantics can be adapted and extended to formally define the meaning of UML 2.0 class, composite structure, and statechart diagrams.

© 2003 Elsevier Science B.V. All rights reserved.

Keywords: SDL; Distributed systems; System design; Formal semantics; Abstract state machines

1. Introduction

Over a period of more than 20 years, Specification and Description Language (SDL) [1] has matured from a simple graphical notation for describing a set of communicating finite state machines to a sophisticated specification technique with graphical syntax, (abstract) data types, structuring mechanisms, object-oriented features, support for reuse, companion notations, and commercial tool environments. It took more than 10 years of language development until the semantics of SDL became defined formally in 1988, upgrading the notation to a formal description technique (FDT) [2]. This formal semantics, which was based on a combination of the VDM meta language Meta-IV and a CSP-like communication mechanism, had been maintained and extended for subsequent versions of SDL.

In November 1999, a new version of SDL, referred to as SDL-2000, was approved by ITU. SDL-2000 incorporates important new features, including object-oriented data types, a unified agent concept, hierarchical states, and exception handling. Based on the assessment that the existing
Meta-IV programs [2,3] would be too difficult to extend 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 [4], consisting of experts from Germany and China including the authors of this paper, was formed in 1998. The formal semantics defined by this group was officially approved by ITU in November 2000, when it became Annex F to Z.100, the SDL standard, and thus part of the SDL language definition [5].

In this paper, we briefly survey and compare existing approaches to define the semantics of SDL formally, based on a number of design objectives (Section 2). The ITU-T approach will then be outlined in more detail (Section 3), addressing the following steps: (1) mapping of non-basic language constructs to the core language, (2) checking of static semantics conditions, (3) definition of the SDL Abstract Machine (SAM) model, and (4) definition of the SDL Virtual Machine (SVM). Furthermore, experiences from the SDL-to-ASM-compiler project are presented (Section 4). It is expected that the SDL-2000 semantics can be adapted and extended to formally define the meaning of UML 2.0 class, composite structure, and behavior diagrams (Section 5).

2. Design objectives and related work

Standardization is an ongoing and potentially open-ended activity so that there is considerable dynamics in the development and maintenance of a standard. A major challenge for the development of the SDL formal semantics was the necessity to deal with a moving target. The SDL-2000 language definition itself indeed has been revised continuously introducing a number of substantial changes and extensions. Such dynamics naturally demands robustness of the formalization as a prerequisite for practicability. Below, we discuss fundamental questions concerning the practicability and suitability of the chosen formalization approach.

2.1. Design objectives

Ideally, correctness and completeness of both the static semantics and the dynamic semantics may be considered the two main objectives of the SDL formal semantics definition. However, starting from the informal language definition as formulated in Z.100 with the intuition of the language design experts, there is no way of proving absolute evidence, neither for correctness nor for completeness. Thus, the best one can achieve is a certain degree of confidence that the resulting semantic model is coherent and consistent, and that it captures all required language properties.

To this end, one can give a justification of the underlying semantic model in three basically different ways [6]: conceptually, experimentally, and, to some extent, mathematically with regard to its internal consistency. A conceptual justification, e.g., based on common sense, basically results from the standardization activity itself as any design decision is checked over and over again. On the contrary, experimental validation as well as inspection by analytical means require an additional effort for the formalization of the semantic model, i.e., for turning English into mathematics.

The ultimate purpose of formalization is to reveal and eliminate ambiguities, loose ends, and inconsistencies hidden in natural language definitions, thereby sharpening requirements into specifications that state even the most subtle semantic properties with mathematical precision. Also, formalization is a prerequisite for executability as required for experimental validation, which often is the only way of discovering undesirable behavior and hidden side effects. In combination with inspection by analytical means, high-level experimental validation of key language properties calls for an abstract operational formalism with readily available tool support. ¹

Additionally, there are several objectives addressing pragmatic needs imposed by the industrial specification and design context in which this work

¹ Inspection by analytical means requires the formalism itself to be defined in terms of mathematical logic so that the underlying semantic basis is well established and understood. To cope with complexity, the formalism needs to be abstract allowing one to focus on key properties rather than on technical details. Finally, being operational, it supports the execution of abstract models in a direct manner avoiding implementation overhead.
is carried out; thereby ensuring the practicability of the approach and the acceptance of the resulting formalization. Among those, the three major issues are intelligibility, conciseness and flexibility. A formal semantics that is cryptic or too complex is essentially useless as documentation. Also, conciseness and flexibility are crucial for maintainability: the evolving nature of the language definition simply requires that a formal model can be modified and extended with a reasonable effort.

2.1.1. Basic design choices

A pragmatic approach to achieve conciseness is “bottom-up”: the starting point is a set of source language concepts for which a formal semantics is directly defined. This core is then extended by additional abbreviations that are defined “syntactically” by transformation to the core language, sometimes called normalization. In case of abbreviations, the relationship between core language and syntactical extensions is straightforward. In other cases, this may be not so obvious and may have a substantial impact on the organization of the language definition. For instance, in an object oriented language, inheritance can either be considered an abbreviation and consequently be defined by transformations, or be given a formal semantics directly.

A very important design choice is the mathematical modeling paradigm that underlies the formal definition of SDL. The formalism should fit SDL, not vice versa. Clearly, it must be sufficiently expressive to capture every language aspect. Avoiding formalization overhead (that is any gap between the language and its formal semantics) is another crucial requirement. The fundamental semantic concepts of SDL need to be identified, as far as possible, with related semantic concepts of the formalism resulting in a one-to-one mapping of semantic entities and relations in the source language to their mathematical images in the formal model. 2

For these and other reasons, the Abstract State Machine (ASM) paradigm introduced by Gurevich [7] was finally selected as the underlying formalism. An ASM formalizes discrete dynamic system behavior in terms of runs of an abstract machine. Based on a simple but universal language, system behavior is expressed in the form of a non-deterministic state transition system over abstract data structures. Building on well known concepts and structures of discrete mathematics and computer science, the ASM formalism offers a sound compromise between mathematical elegance and practical relevance. Indeed, sequential, parallel and distributed ASMs have been used extensively for modeling the most complex architectures, languages and protocols as documented in the literature (e.g., [8]; see also the ASM home page at www.eecs.umich.edu/gasm/).

In order to support executability, the formal semantics defines, for each SDL specification, a mapping to reference ASM code, which enables SDL-to-ASM compilers. For a substantial subset of SDL, such a compiler has been developed to show the feasibility of this approach (see Section 4).

2.1.2. How to establish correctness?

The compilation of SDL actions to SAM instructions (see Section 3.2) defines semantics by a mapping to a well-defined semantic basis. Such mapping is a well-known technique, but is often considered to be as dangerous as helpful. In particular, the question arises: “Who validates and/or verifies the correctness of this mapping and with respect to which origin?”. There are actually two answers:

Current situation. The current situation indeed requires validating the SDL formal semantics against the informal language definition of Z.100, except for those parts where the semantics is exclusively defined by the formal model. Of course, both semantic versions have to be consistent with each other. However, neither of these versions takes precedence over the other.

Perspective. Once the reliability of the formal model has been established with sufficient confidence, the SDL formal definition could become the basis for standardization, whereas the informal

---

2 For instance, this includes semantic aspects such as discrete system states, asynchronous interaction, concurrent execution, interface mechanisms, non-determinism and timing behavior.
part then would have the role of providing additional explanations that may be sufficient as reference whenever an ultimate degree of detail and precision is not needed. Obviously, this would avoid the notorious correctness problem, achieving “correctness” by construction.

2.2. Related work

There have been a variety of competing attempts to formalize the SDL semantics using various formal methods. According to their principle objectives, one can distinguish two basically different directions of research activities: (1) machine-supported analysis and verification of SDL system specifications; (2) documentation, maintenance and validation of the language definition.

2.2.1. Analysis and verification

Bozga et al. define the SDL intermediate representation format IF [9] as the basis for a systematic integration of the ObjectGeode tool set with different validation tools supporting a wide spectrum of validation methods, including formal verification and automatic test case generation. In IF, a system is represented as a set of timed automata communicating asynchronously through a set of buffers or by rendezvous through a set of synchronization gates. A well defined set of APIs allows to access IF programs and the associated models at different representation levels.

Fischer, Dimitrov and Tauber propose an extended Petri Net model, so-called SDL time nets, as the formal basis to verify SDL protocol specifications [10, 11]. 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 tool environment SITE. Though these transformations, in principle, remain invisible for the user, the authors concede that certain knowledge is necessary about the internal structures of the analysis tools.

In [12], Bergstra and Middleburg define a process algebra semantics of a restricted version of SDL, called φSDL, not covering structural aspects. They also restrict the 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”.

Broy [13], Holz and Stølen [14], and Hinkel [15] model various subsets of (essentially) Basic SDL using stream processing functions of FOCUS [16]. While it may be natural to model SDL process communication as discrete streams of signals, the functional view neither supports the concept of system states and state transitions nor allows the stream formalism for an adequate treatment of time. Even the most comprehensive model [15] builds on a fundamentally restricted notion of global system time which is inadequate for expressing time quantities explicitly.

2.2.2. Documentation and validation

Lau and Prinz started an ambitious attempt towards a comprehensive formalization of the SDL-92 semantics as basis for further development of SDL with their definition of Base SDL (BSDL) [17]. With the ultimate goal to simplify the language definition, they proposed an Object-Z based model to define a universal core for SDL as a conceptual framework for dealing with the main building blocks of the language. To this end, 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.

Gotzhein et al. proposed the generation of transition systems from attributed abstract syntax trees (ASTs) [18]. 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 some attributed AST containing all the information required to generate a behavior model. The underlying mathematical model and formal notation are not determined leaving room for subsequent choices.

In [19, 20], Gläser and Karges define an ASM model of the dynamic semantics of Basic SDL-92/96 based on an abstract language interpreter. Utilizing the fact that the underlying semantic notions of concurrency, state and time match those of SDL, the resulting formalization is fairly concise
yet readable and understandable. This work provides a conceptual framework which has further been developed and extended by combining it with the compilation-based view of [18] as well as fundamental concepts from [17] resulting in the formal semantics of SDL-2000 presented in more detail in Section 3.

3. The ITU-T approach

In November 2000, the formal semantics of SDL-2000, referred to as the ITU-T approach, was officially approved to become part of the SDL language definition [5]. Currently, it is the only comprehensive and complete formal semantics of SDL-2000, covering all static and dynamic language aspects. It is based on and extends earlier work, especially the core language approach of [17], the ASM approach of [19,20], and the compilation approach of [18]. The ITU-T approach meets the design objectives stated in Section 2.1, and enables true executability. In this section, the static and dynamic semantics of SDL-2000 are surveyed. Further details on the dynamic semantics and, in particular, on the use of the ASM formalism can be found in [21].

3.1. Static semantics

The static semantics covers transformations and checks that can be done before executing a specification. In the scope of SDL, there are two major parts of the static semantics:

- **Well-formedness conditions**: Like most languages, the SDL concrete syntax is given in a context-free way. Additional constraints are imposed using context conditions.
- **Transformations**: In order to cope with the complexity of the language SDL, the standard Z.100 identifies certain concepts to be core concepts and defines transformations of various other concepts into these core concepts.

Starting point for defining the static semantics of SDL is a syntactically correct SDL specification as determined by the SDL grammar. In Z.100, a concrete textual and a concrete graphical syntaxes are defined using Backus–Naur-Form (BNF) with extensions to capture the graphical language constructs. An abstract grammar is also defined. From a syntactically correct SDL specification, an AST is derived by standard compiler techniques (namely, parser construction for a context-free grammar). The structure of this AST is defined such that it resembles the concrete textual and the concrete graphical grammars. The correspondence between the concrete grammars and a first abstract syntax form, called AS0, is almost one-to-one, and removes irrelevant details such as separators and lexical rules. A second step translating AS0 to the final abstract syntax form, called AS1, is formally captured by a set of transformation rules together with a mapping. This results in the following structure of the formalization (see Fig. 1):

- Transformation rules modifying AS0 trees are described in so-called model paragraphs of

![Fig. 1. Overview of the static semantics.](image-url)
Z.100, and are formally expressed as rewrite rules.

- After application of the transformations, the structure of the AS0 tree is similar to an AS1 tree. This means that the mapping from AS0 to AS1 is almost one-to-one.
- The well-formedness conditions are split into conditions on AS0 and AS1 (see Fig. 1). They are formalized in terms of first-order predicate calculus.

### 3.1.1. Transformations

Z.100 prescribes the transformation of SDL specifications by a sequence of **transformation steps**. Each transformation step consists of a set of single transformations as stated in the Model sections of Z.100, and determines how to handle some of shorthand notations (or abbreviations). The result of one step is used as input for the next step. The following excerpt of Z.100 prescribes how to deal with a particular shorthand notation, namely a list of output signals:

If several `<signal identifier>`s are specified in an `<output body>`, this is derived syntax for specifying a sequence of `<output>`s in the same order as specified in the original `<output body>`.

The rule states that a list of output signals is shorthand (called “derived syntax”) for a sequence of single outputs.

To formalize the transformation rules of Z.100, rewrite rules are used. A single transformation is realized by the application of a rewrite rule to the specification, which essentially means to replace parts of the specification by other parts as defined by the rule. The following rewrite rule formalizes the transformation above:

$$<<\text{output}>(<\text{output body}>(<\text{output}>(r)))>$$

**provided** $r \neq \text{empty}$

$$= 1 \Rightarrow$$

$$<<\text{output}>(<\text{output body}(<\text{o}>)),<\text{output}>(<\text{output body}>(r)))>$$

### 3.1.2. Well-formedness conditions

The **well-formedness conditions** define additional constraints that a specification has to satisfy. These constraints cannot be expressed in BNF, though they are static and can be defined and checked independently of the dynamic semantics definition. An SDL specification is valid if and only if it satisfies the syntactical rules and the static conditions of SDL.

Here is a sample resolution rule taken from Z.100, and its formalization:

There is again an application context for this rule given after the `\forall` symbol—the rule is applicable for `<output body>` AS0 nodes. The `<signal identifier>` selected from this node must refer to a `<signal definition>`. The actual resolution is done by the function `\text{refersto}_0`, which is formally defined in [5] covering about four pages.

The application of this rule is shown in Fig. 3: Sig2 and Sig3 indeed denote `<signal definition>`s.
3.2. Dynamic semantics

The dynamic semantics defines the dynamic properties resulting from the execution of valid SDL specifications, i.e., their legal behavior. It is based on ASMs introduced by Yuri Gurevich [7], and defines, for each SDL specification, a distributed real-time ASM, covering concurrency, asynchronicity, and time (see Fig. 4). The core of this model is a logical hardware called SAM, providing a dynamic architecture according to the SDL specification under execution, various types of agents, and an instruction set (see Section 3.2.1). On top of this logical hardware, typical operating system functionality (e.g., scheduling of agents, runtime support for the execution of complex actions) is provided by the SVM, surveyed in Section 3.2.2. In this paper, we restrict ourselves to conceptual aspects of the SAM and SVM. Details on the use of the ASM formalism can be found in [21].

3.2.1. SDL abstract machine

The SDL Abstract Machine, or SAM, consists of the following parts (see Fig. 4):

- **Signal Flow Model**, which defines a uniform treatment of signal flow related aspects, in particular, asynchronous communication between agents through exchange of signals via channels connected to gates,
- **SAM Agents**, which model the SDL concepts ‘SDL agent’, ‘SDL agents set’, and ‘SDL channel’, and
- **Behavior Primitives**, which can be seen as the instructions of the SAM.

In the following, we outline the signal flow model in some detail.

3.2.1.1. Overall organization. The signal flow model is based on a decentralized control mechanism for defining asynchronous communication between SDL agents (such as processes, blocks or a system). Basically, it deals with the transportation of signals over delaying and non-delaying channels connecting SDL agents via their gates. This model also defines timers and exceptions as special kinds of signals. As such, it forms the core of the SDL communication architecture.

A static domain **Signal** represents the set of all signal types as declared by an SDL specification. This also includes timers and exceptions, which are modeled as signals, too. Dynamically created signal instances belong to a dynamic domain **Signal-Inst**. Basic functions on signals are **signalSender**, **toArg**, and **viaArg** yielding the sender process, the destination, and optional constraints on admissible communication paths, respectively.

Exchange of signals between SDL agents and their environment is modeled by means of **gates** from a static domain **Gate**. A gate forms an interface for serial and unidirectional communication between two or more agents. Accordingly, gates are either classified as **input gates** or **output gates**.

Signals received at an input gate of an agent set are appended to the input port of an agent according to the value of **toArg**. Simultaneously arriving signals matching the same agent instance are appended, one at a time, in an arbitrary order. Signals are discarded if no matching receiver instance exists.

3.2.1.2. Signal delaying mechanism. Signals may be (and often are) subject to arbitrary but finite delays caused by SDL channels. To model signal delays, we associate an arrival time with each
signal. The signal is appended to the next gate, from which it may be removed only after the arrival time has been reached. In case of concatenated channels, a new arrival time is determined, and the signal is forwarded.

Fig. 5 illustrates this solution. With each gate \( g \), we associate some finite but possibly empty sequence of signals, denoted as \( \text{schedule}(g) \). Thus, gates are capable of holding signals that are in transit, i.e., have not yet arrived. Intuitively, \( \text{schedule} \) specifies, for each gate \( g \) in \( \text{GATE} \), the corresponding signal arrivals at \( g \). A signal is not available at its destination prior to the specified signal arrival time, where \( \text{now} \) represents the global system time in a given SAM state.

### 3.2.1.3. Unidirectional communication

SDL channels consist of either one or two unidirectional channel paths. In the SAM architecture, each channel path is identified with a link entity from a static domain \( \text{LINK} \). The elements of \( \text{LINK} \) are SAM agents representing point-to-point connection primitives for the transport of signals between related gates. Several such links may be connected to the same output gate so that a signal arriving at this output gate can be forwarded to several possible destinations depending on the signal type, the path constraints for the signal and the signal constraints of the links.

In general, this means that one or more link agents applicable to a given signal indeed may compete in making simultaneous attempts to forward the same signal to different destination gates. To understand how this situation is handled making sure that only one of the links will succeed and the signal will not be duplicated, we briefly outline below the asynchronous execution model as associated with the underlying distributed real-time ASM forming the semantic basis of our formal definition. This also serves to exemplify the formalization approach using an illustrative example.

### 3.2.1.4. Formal communication model

A distributed ASM defines an asynchronous computation model consisting of some finite collection of autonomously operating agents. Each agent executes a program, where agents that execute the same program are considered to be of the same type. For instance, all link agents execute the program \( \text{LINKPROGRAM} \). Agents interact with each other by reading and writing shared locations of global machine states. The underlying semantic model resolves potential conflicts by making non-deterministic choices according to the definition of partially ordered machine runs [7].

The behavior of a link is stated by the state transition rule \( \text{FORWARDSIGNAL} \) forming the program \( \text{LINKPROGRAM} \). For improved readability of ASM programs, complex transition rules are structured by means of rule macros that often have formal parameters. In the below definition of the rule \( \text{FORWARDSIGNAL} \), \( \text{Self} \) identifies a particular link agent (the one executing the rule).

### ForwardSignal

\[
\text{FORWARDSIGNAL} \equiv \\
\text{if } \text{Self/from.queue} \neq \text{empty then} \\
\quad \text{let } si = \text{Self/from.queue.head} \text{ in} \\
\quad \text{if } \text{Applicable}(si, \text{signalType}, si, \text{toArg}, si, \text{viaArg}, \text{Self/from,Self}) \text{ then} \\
\quad \quad \text{DELETE}(si, \text{Self/from}) \\
\quad \quad \text{INSERT}(si, \text{now} + \text{Self.delay}, \text{Self/to}) \\
\quad \text{endif} \\
\text{endlet} \\
\text{endif}
\]

A link agent \( l \) basically performs a single operation, namely: signals received at gate \( l/from \) are forwarded to gate \( l/to \). Whenever \( l \) is applicable to a waiting signal \( si \) (as identified by the \( l/from\ queue\ head \)), it removes \( si \) from \( l/from\ queue \) and inserts it into \( l/to\ schedule \). Competing attempts of two or more link agents to forward the same signal
cannot cause a duplication of the signal. Technically, this property is ensured by the underlying concurrency model (cf. the coherence condition in the definition of partially ordered runs [7]).

Note that the resulting signal flow model architecture is fairly robust allowing for the incorporation of additional features in future versions of SDL. For instance, one may have channels with more complex properties (like unreliable transmission behavior) and a dynamically changing communication infrastructure (with channels being added and removed at run time). Such extensions can be easily expressed on the basis of the decentralized signal flow model without any major revision of the current definitions.

3.2.1.5. Real time. SDL is promoted for the specification and design of distributed real-time systems. However, its support for real-time behavior is essentially limited to the use of timers and the underlying notion of global system time. Taking into account current activities within ITU focusing on more sophisticated concepts for the specification and analysis of timing behavior [22] (see also [23]), our modeling framework builds on a notion of dense time, where time values are represented as real numbers. For further details, see [21] and [5].

3.2.1.6. Timers and exceptions. A particularly concise way of modeling timers is by identifying timer objects with timer signals. More precisely, each active timer is represented by a corresponding timer signal in the schedule associated with the input port of the related process instance. Like timers, exceptions are identified with exception signals.

3.2.2. SDL virtual machine

The SVM provides typical operating system functionality on top of the logical hardware of the SAM. Under the control of the SVM, ASM programs that are associated with link agents, SDL agents and SDL agent sets, respectively, are run. The SVM defines suitable abstractions by a set of macros and functions, which capture the structure of an SDL system at runtime and thus determine the dynamic architecture of the SAM, the structure of agents, the selection of transitions and their firing. We will now sketch some aspects of the SVM. For a detailed treatment of these complex issues, we refer to [21] and [5].

SDL agents are the most complex active components of an SDL system at runtime. Therefore, several activity phases are introduced, which in turn have sub phases and so on. At the top level, the phases initialization and execution are distinguished. After an SDL agent has been created—either at system initialization time or dynamically—it enters the initialization phase. During this phase, the structure of the agent, which may consist of a hierarchical inheritance state graph (see below), connection structure, and further agents, is created. Then, the agent enters the execution phase, where it remains until its termination.

The behavior of SDL agents is defined by an ASM program. Depending on the current top level agent mode, an ASM macro defining the corresponding activities is selected. Macros are hierarchically structured and thus provide useful abstractions. Execution of agents is modeled by alternating phases, namely transition selection and transition firing, preceded by a start phase.

In the following, we outline the transition selection, and in particular the structure on which the transition selection is based and which is built up dynamically, called hierarchical inheritance state graph.

The solution adopted in the formal semantics avoids transformations entirely and defines the meaning of transition selection directly, based on a so-called hierarchical inheritance state graph (his-graph), which is associated with the state machine definition of an SDL agent (see Fig. 6). The his-graph can be understood as the control structure of an SDL agent. The static part of that control structure is unfolded during the initialization phase. This static part can be extended dynamically, as the agent performs procedure calls, and shrinks when a procedure execution terminates. It is one of the findings of the formal semantics group that SDL procedure calls can be modeled by dynamic composite state graphs.
In previous versions of SDL, selection of a transition consisted of checking a single major state of an SDL agent, as defined informally in Z.100 (Z.100, Section 11.2, see above). With the incorporation of inheritance in SDL-92, this became more complicated (Z.100, Section 8.3.3, see above), but was resolved by a transformation step removing inheritance to keep the dynamic semantics stable. With the addition of composite states in SDL-2000, transformations are no longer feasible (Z.100, Section 11.11, see below). Also, the complexity of the selection process can be substantial, as the formal semantics has to cover the most general cases with all possible combinations of transition triggers, composite states, and inheritance.

In Fig. 6, an example of a his-graph together with input signals that trigger transitions associated with the corresponding states is shown. This graph is the result of instantiating a state machine definition, which is defined by composite state type definition 1 (CSTD1 for short). This composite state type inherits from CSTD2, which in turn inherits from CSTD3 (horizontal arrows). State S11 of CSTD3 is refined by an instance of CSTD4 (vertical arrow), which inherits from another composite state type, and so on. In addition to composite states consisting of state nodes, state aggregation nodes formed by state partitions can be represented in the his-graph (not shown in this example).

When the SDL agent enters state S1111, it still is in the containing states of S1111, as shown in Fig. 6. When selecting a transition, these states have to be checked in a particular order implied by Sections 11.2, 11.11 and 8.3.3 of Z.100 (see above). For the example in Fig. 6 and current state S1111, the resulting order is marked by the numbering of

---

**Z.100, Section 11.2 State**

For each state, the **Save-signals, Input-nodes, Spontaneous-signals, and Continuous-signals** are interpreted in the following order:

(a) if the input port contains a signal matching a priority input of the current state, the first such signal is consumed; otherwise,
(b) in the order of the signals on the input port:
   (1) the Provided-expressions of the Input-node corresponding to the current signal are interpreted in arbitrary order, if any;
   (2) if the current signal is enabled, this signal is consumed; otherwise,
   (3) the next signal on the input port is selected.
(c) if no enabled signal was found, in priority order of the Continuous-signals, if any, with Continuous-signals of equal priority being considered in an arbitrary order and no priority being treated as the lowest priority:
   ... 
   (d) if no enabled signal was found, ...

At any time in a state which contains Spontaneous-transitions, the state machine may interpret the Provided-expression of a Spontaneous-transition and subsequently, the Transition.

**Z.100, Section 8.3.3 Virtual Transition/Save**

A virtual priority input or input transition can be redefined to a new priority input or input transition or to a save.

A virtual save can be redefined to a priority input, an input transition or to a save.

A virtual spontaneous transition can be redefined to a new spontaneous transition.

... 

A virtual continuous transition can be redefined to a new continuous transition. The redefinition is indicated by the same state and priority (if present) as the redefined continuous transition.

---

**Z.100, Section 11.11 Composite State**

A transition emanating from a sub state has higher priority than a conflicting transition emanating from any of the containing states.

Conflicting transitions are transitions triggered by the same input, priority input, save or continuous signal.

...
state graphs. In order to select a transition in state S1111, e.g., the transitions attached to up to seven states have to be examined, based on the current contents of the input queue. For instance, when the current input signal to be consumed is i5, then the transition attached to S1111, which has a higher priority than the conflicting transition attached to S11 in CSTD3, is selected. For current input signal i1, the transition attached to S11 in CSTD1, which redefines the transition attached to S11 in CSTD2, is chosen. In general, priority signals, enabling conditions, spontaneous transitions, and continuous signals are to be considered, too.

Firing of a transition is decomposed into the firing of individual actions, where for each action, a SAM behavior primitive is executed. Firing of an action may in turn consist of a sequence of steps. At the beginning of a transition, the current state node is left, which may entail the leaving of inner state nodes and the execution of exit procedures and exit transitions. At the end, either a state node is entered, or a termination takes place.

4. True executability

A central design objective for the SDL formal semantics definition is executability. This objective has guided the choice of the underlying mathematical formalism. Finally, the ASM formalism was selected, due to the existence of tools for the execution of ASM models. The formal semantics defines, for each SDL specification, a mapping to reference ASM code, which can be executed with these tools. Since size and complexity of the mapping rule out the manual translation of SDL specifications to ASM code, a compiler generator was conceived and implemented. Taking the SDL formal semantics definition (a Word document) as input, it automatically generates an SDL-to-ASM compiler. Thus, modifications to the formal semantics definition can easily be taken into account. Below, we briefly comment on the steps to generate a compiler (see Fig. 7).

As a first step, all formal text is extracted from the SDL standard (a Word document). This step has to be performed for all the formal parts of the SDL standard, i.e., the syntax descriptions within the main body of the SDL standard (Z.100) and the formal texts within the formal semantics annex Z.100.F. The result of this step are several plain text files.

From the extracted texts, an SDL lexer and a parser are constructed using the standard compiler construction tools lex and yacc. Moreover, to easily handle syntax trees, a tool called kc
(kimwitu) is used for all syntax tree related activities: construction of syntax trees, tree parsing, tree transformation and tree rewriting. This step already allows the construction of an AS0 syntax tree from a given syntactically correct SDL specification.

In a subsequent step, the AS0 tree is processed according to the static semantics rules: checking and transformation (cf. Section 3.1). In the implementation, this is all covered by the kc tool, which generates a tree checking function from the well-formedness conditions and tree rewriting functions from the transformation rules. The result of this step is an AS1 tree, if the specification is semantically correct.

The last step of the generated SDL compiler is the ASM format generation according to the compilation function of the semantics, which translates tree nodes into SAM behavior primitives. Moreover, the complete AS1 tree is output in ASM representation. This results in the specification in ASM format. The result of the formal SDL compiler is ASM text, which is joined with the ASM text of the formal semantics. All resulting ASM code is executed using the AsmL tool [24], providing final evidence for true executability.

For sample specifications, the formal semantics behaves as described in the informal text. In general, there might be differences between the two descriptions, which have to be corrected (see also Section 2.1.2).

The trace below of a sample specification is listed. Sending and receiving of signals is marked by underlining.
5. Experiences and perspectives

5.1. Impact on the SDL-2000 informal definition

The formal SDL semantics has been conceived in parallel to the language definition itself. While developing the formal semantics definition, there have been numerous discussions with the SDL experts in order to reach a common understanding of the Z.100 document, to resolve ambiguities, and to remove inconsistencies. As it turned out, this provided valuable feedback, as problems with formalizing certain language aspects often led to discussions that revealed problems with the language definition itself. Also, the feasibility to treat certain aspects directly in the formal semantics made a number of complex transformations obsolete and thus helped to make the documents more concise (see [21] for details). In contrast to the past, it is now official policy that if there is an inconsistency between the main body of Z.100 and Annex F, then neither the main body of Z.100 nor Annex F take precedence when this is corrected.

5.2. Debugging the SDL-2000 formal definition

Making the formal semantics truly executable was an interesting exercise. It showed, that a formal semantics behaves just like any formal text, e.g., a computer program. We detected about 2000 errors, most of which can only be found using tools to check the formal texts. In the scope of the SDL semantics, about 35% of the errors detected were very simple spelling errors or syntax errors. About 25% of the errors were typing errors, which again shows that a type checker can help a lot to produce good formal documents. The remaining errors resulted from problems with the structure of the syntax trees (10%), with parameters (10%), and with syntactically correct misspellings (15%). Only about 2% of the errors were “real” errors in that they needed a semantic change in the formal text.

5.3. Unifying SDL and UML

In 1997, the first version of the unified modeling language (UML) was standardized by the object
management group (OMG), an international body responsible for the development of computer industry specifications. In mid-2001, after several years of industrial experience and a series of revisions, OMG members started work on a major upgrade to UML 2.0. UML 2.0 is expected to resolve a number of problems identified in UML 1.x, concerning, for instance, scalability, architectural modeling, component-based development, dynamic behavior, and executability. In essence, this shall be achieved by merging UML 1.x with a subset of SDL-2000, and by defining an SDL profile on the basis of UML 2.0.

A major improvement of UML 2.0 is the distinction and integration of the following diagram types:

- **Class diagrams** define the class structure of a system, in particular, by introducing classes with attributes and operations, and by defining associations among classes. Some associations (aggregation, composition, and specialization) are predefined, others can be introduced by the system developer. Also, classes can be defined as active, and then refined in architecture and statechart diagrams.

- **Composite structure diagrams** define the internal structure of active classes, by identifying their active components and their communication ports, and by associating ports through typed connectors.

- **Statechart diagrams** define the dynamic behavior of active classes in terms of an extended finite state machine.

These diagram types and concepts of UML 2.0 have counterparts in SDL-2000. The UML concepts abstract the SDL ones which are often more elaborate and precise. Class diagrams correspond to signature definitions of SDL types, augmented by further information on associations between classes. From a conceptual view point, active classes abstract SDL agent types. Composite structure diagrams abstract SDL system and block type diagrams, where ports and connectors are used instead of gates and channels, respectively. It is noted here that there is some overlap between class and architecture diagrams, as the is-part-of relationship can be expressed in both diagram types. Finally, statechart diagrams can abstract SDL process and state machine graphs, with corresponding actions.

The strong relationship between UML 2.0 and SDL-2000, apart from the structural aspects, is founded on the use of the same concept of a system: a system is modeled as a set of hierarchical, asynchronously communicating extended finite state machines. Furthermore, similar basic concepts are used, for instance, UML active classes are related to SDL agent types, UML active components are related to SDL agent instances, UML ports and connectors are related to SDL gates and channels, respectively.

It is expected that these similarities will make it straightforward to adapt the formal semantics of SDL-2000 to UML 2.0. While it seems to be sufficient to apply renaming in case of corresponding concepts, extensions will be needed to incorporate the more general definition of class structures, in particular, the use of general associations. Also, the integration of further UML diagram types into the formal semantics, e.g., sequence and collaboration diagrams, should be considered at this point.

The existence of an executable formal semantics for SDL-2000 offers an opportunity for the standardization of UML 2.0, as it is expected that, for the stated reasons, adaptation can be done with a moderate effort. However, unlike in the SDL community, the demand for a formal semantics may not be so strong in OMG, as the advantages of a mathematically precise meaning of an industrial language are often a matter of debate. Therefore, to avoid that the definition of the formal semantics of UML 2.0 becomes just an academic exercise, OMG members should explicitly request and support this effort, and adopt the policy that if there is an inconsistency between the informal definition and the formal semantics of UML 2.0, then neither takes precedence when this is corrected.

5.3.1. Further issues

Conceiving the nature of standardization as an ongoing activity, even the most recent version of SDL can only be a snapshot of an evolving language definition. To meet the needs of system design experts in a rapidly developing segment of
systems technology, the language has been improved over the past 25 years, evolving from a primitive graphical notation to a sophisticated formal description technique. Typically, every 4 years a new version of SDL is released (e.g., SDL-88, SDL-92, SDL-96, SDL-2000). Such dynamics in the definition of a rich language like SDL clearly demands robustness of the formalization approach as a prerequisite for practicability. Conciseness and flexibility therefore were of primary importance for the choice of the modeling framework.

Despite of the richness of SDL, the SVM model is intelligible and maintainable. This is a direct result of the innovative modeling concepts, namely: the abstract operational view, the compiler-based approach, the organization of the abstract machine model, and the consequent use of parameterized ASM rule macros. To further improve the maintainability of the SDL semantics definition, we intend to introduce an even more concise formal model defined on top of the existing one as a means for illustrating the overall organization of the language definition. This model is supposed to be readable and understandable without requiring any formal background.

Finally, it should be stressed that the definition of the formal semantics has not just been an academic exercise, but has taken place in a real-life industrial setting. In our opinion, it is this kind of result that academic efforts should eventually lead to. The successful application of mathematical formalisms to real-world problems and their approval by industry is a strong selling point for having formalisms at all. In this sense, the work reported in this paper is an important achievement.

Acknowledgements

We thank Egon Börger and Yuri Gurevich for inspiring and valuable discussions on fundamental aspects of our ASM model of SDL, Joachim Fischer and Franz Rammig for their continuous support throughout the entire project, Anders Olsen for providing input on the former SDL semantics, and Rick Reed and Thomas Weigert as responsible SDL Rapporteurs of ITU-T Study Group 17 WP 3/17 for providing valuable clarifications and continuous pressure to keep the deadlines. Furthermore, we thank the members of the SDL Expert Group, in particular Robert Eschbach, Martin von Löwis, and Ying Wang. We also thank the anonymous reviewers for their valuable and detailed comments that helped us to improve the quality of the paper. Finally, we extend our gratitude to Microsoft Research, Tellogic, Ericsson, and Motorola for their financial support of part of this work.

References

Reinhard Gotzhein received his Diplom (M.Sc.) and Dr.-Ing. (Ph.D.) Degrees in Computer Science from the University of Erlangen, Germany, in 1982 and 1985, respectively. From 1985 until 1986, he was a post-doctoral fellow at the University of Montreal, Canada. From 1987 until 1993, he worked as Assistant Professor at the University of Hamburg, Germany, where he also completed his Habilitation in 1992. From 1991 until 1992, he was a visiting professor at the University of Montreal. Since 1993, he is an Associate Professor at the University of Kaiserslautern, heading the Computer Networks Group. His research interests include communication systems, protocol engineering, software reuse technologies, and formal description techniques. Dr. Gotzhein has published more than 100 scientific papers. Since 1998, he is acting as Associate Rapporteur of the ITU-T on the semantics of SDL. He chaired the FORTE/PSTV96 International Conference, has served on program committees for numerous international symposia, on the editorial board and as guest editor for several scientific journals, and is scientific advisor of a research institute.

Andreas Prinz studied Mathematics and Computer Science at the Humboldt-University in Berlin, Germany. He received his M.Sc. in Mathematics (1988) and Ph.D. (1990) in Computer Science there. From 1990 until 1993 he was a post-doctoral fellow at the Humboldt-University. From 1993 to 1994 he worked at the Software Verification Research Centre (SVRC) in Brisbane, Australia. From 1994 to 1997 he again worked at the Humboldt-University. Since 1997 he works at the DRe search GmbH, a company in Berlin.

His research interests include formal methods together with their application and use in tools. So he has also strong interest in software technology and compiler construction. Dr. Prinz has worked in several projects with Siemens dealing with the development of modern telecommunication systems using advanced technology. He was the leader of the SDL-2000 formal semantics project and led the implementation of the formal semantics.