Journal of Universal Computer Science, vol. 7, no. 11 (2001), 1024-1049
submitted: 7/6/01, accepted: 1/10/01, appeared: 28/11/01 © Springer Pub. Co.

Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models

Robert Eschbach
(Department of Computer Science, University of Kaiserslautern
eschbach@informatik.uni-kl.de)

Uwe Glässer
(Microsoft Research, Redmond[1]
glaesser@sdl-forum.org)

Reinhard Gotzhein
(Department of Computer Science, University of Kaiserslautern
gotzhein@informatik.uni-kl.de)

Martin von Löwis
(Department of Computer Science, Humboldt-University Berlin
loewis@informatik.hu-berlin.de)

Andreas Prinz
(Department of Computer Science, Humboldt-University Berlin
prinz@dresearch.de)

Abstract: In November 1999, the current version of SDL (Specification and Description Language), commonly referred to as SDL-2000, has passed ITU-T, an international standardization body for telecommunication. The importance and acceptance of SDL in the telecommunication industry surpasses that of UML, which can be seen as the major competitor. A crucial difference between SDL and UML is the existence of a formal SDL semantics as part of the international standard, which has a positive impact on the quality of the entire language definition. In this paper, we treat fundamental questions concerning practicability, adequacy and maintainability of the formalization approach, provide insights into the formal semantics definition and point out several effects on the SDL standard.

Keywords: SDL, Specification and Description Language, ASM, Abstract State Machines, FDT, Formal Description Technique, Formal Semantics

Category: D.3.1, F.3.2., F.4.3.

1 Introduction

The development of *SDL (Specification and Description Language)* [ITU 1999b] dates back to the early 70ies, when the need for a design language to specify the behavior of distributed real-time systems in general and telecommunication systems in particular

[1] On leave from Heinz Nixdorf Institute, University of Paderborn, Germany

became apparent. Since 1976, SDL is standardized by ITU[2], and in intervals of 4 years, upgrades of this first SDL standard are officially released. Over a period of more than 20 years, SDL has matured from a simple graphical notation for describing a set of communicating finite state machines to a sophisticated specification technique with graphical syntax, data type constructs, structuring mechanisms, object-oriented features, support for reuse, companion notations, tool environments and a formal semantics. Thus, SDL satisfies the primary needs of system developers, and is being broadly applied in industry.

It took more than 10 years of language development until the semantics of SDL has been defined formally in 1988, upgrading the notation to a formal description technique. This formal semantics, which was based on a combination of the VDM meta language Meta-IV and a CSP-like communication mechanism, has been maintained and extended for subsequent versions of SDL in 1992 and 1996. Essentially, the formal semantics is given by a set of Meta-IV programs that take an SDL specification as input, determine its static correctness, perform transformations to replace certain language constructs, and interpret the specification.

In November 1999, a new version of SDL referred to as SDL-2000 has been approved by ITU. SDL-2000 incorporates important new features, including object-oriented data type definitions, a unified agent concept, hierarchical states, and exception handling. Based on the assessment that the existing Meta-IV programs 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 [ITU 2001], consisting of experts from Germany and China including the authors of this paper, was formed in 1998. The formal semantics defined by this group has been officially approved by ITU in November 2000, when it has become Annex F to Z.100, the SDL standard, and thus part of the SDL language definition [ITU 2000].

Before defining the SDL semantics formally, several design objectives have been identified. Among these are intelligibility and conciseness, correctness and maintainability. Additionally, executability has become a key issue, which calls for an operational formalism with readily available tool support. For this and other reasons, *Abstract State Machines (ASMs)* introduced by Yuri Gurevich [Gurevich 1993] have finally been selected as the underlying formalism. To support executability, the formal semantics defines, for each SDL specification, *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.

The paper treats fundamental questions concerning the practicability, suitability and robustness of our formalization in detail. Technically, this is achieved by exemplifying the treatment of an illustrative fragment of the formal semantics, namely a dynamic process creation embedded in a simple signal-triggered transition. Going down all the way from the *attributed Abstract Syntax Tree (AST)* to an executable behavior description, resulting from the generated SDL abstract machine model, illustrates steps (3) to (5) of the following main steps, namely: (1) mapping of

[2] SDL is defined by ITU-T, the Telecommunication Sector of the International Telecommunication Union, in Recommendation Z.100.

non-basic language constructs to the core language, (2) checking of static semantics conditions, (3) compilation of the AST to the *SDL Abstract Machine (SAM)* model, (4) definition of the *SAM programs*, and (5) their execution by the *SDL Virtual Machine (SVM)*. For a survey of related work, the reader is referred to [Eschbach et al. 2000].

Section 2 addresses fundamental questions concerning the practicability, suitability and robustness of the formalization approach. In Section 3, we illustrate some basic features of the SDL language by an example, which is used throughout the paper to provide some deeper insights into the formal SDL semantics. The formal semantics can be roughly divided into the static semantics dealt with in Section 4, and the dynamic semantics treated in Section 5. We summarize the impact of this work on the SDL-2000 standard in Section 6, and give a brief outlook in Section 7.

2 Challenges and Needs

This section addresses fundamental questions concerning the practicability, suitability and robustness of our formalization approach. The emphasis here is on pragmatic aspects rather than on technical ones. To illuminate the rational behind our choices for the underlying conceptual framework, we focus on those arguments that we believe are meaningful beyond the scope of the work outlined in the following sections.

Unlike traditional engineering disciplines that are well established, e.g. like mechanical and electrical engineering, systems engineering deeply relies on informal documentation. Indeed, such informal documentation may be informative and necessary. Still, it is informal and as such it often is as problematic as useful when it comes to specifying properties of technical systems with mathematical precision. In fact, informal descriptions may be, and often are, *ambiguous*, *incomplete*, and even *inconsistent*. Furthermore, they are not executable and thus provide only very limited support for experimental validation, which is ultimately needed for checking the accuracy of a model against our intuitive understanding of the expected system behavior; indeed, it often is the only way to fully understand the implications of the specified system behavior.

Consequently, reliable specifications of complex technical systems must go beyond informal descriptions. Additional information is needed for a clear separation of concerns, to fix the loose ends and to resolve ambiguities and inconsistencies. Guided by pragmatic needs, we construct our formal model of SDL starting from the informal language definition of Z.100. To formalize the static and dynamic properties of SDL appropriately, we have to take into account the constraints imposed by the industrial design context in which this work is carried out.

2.1 Some Design Decisions

Before and during the design of a formal semantics, a number of decisions have to be made. In order to be intelligible and maintainable, the formal semantics should be *concise*, which is of particular importance for “rich” languages. A pragmatic approach to achieve this objective is “bottom-up”: starting point is a set of core language constructs for which a formal semantics is directly defined. Subsequently, this core language is extended by language constructs that are defined “syntactically”, i.e. by transformation to the core language, sometimes called "normalization". In case of abbreviations, the distinction between core language and syntactical extensions is straightforward. In other cases, the decision may be not 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.

An important design decision concerns the *mathematical model* that underlies the formal semantics. As a general guideline, the model should fit the language, not vice versa. If, for instance, the language has structuring mechanisms, it should be possible to represent system structure in the formal model. If a particular interaction paradigm, for instance, asynchronous interaction among system components, is used in the language, this should be reflected in the formal model. If real time is an issue, the formal model should support a suitable notion of time. If the language addresses concurrent systems, the model should provide the desired model of concurrency. Finally, the mathematical formalism should be sufficiently expressive to capture every language aspect. Any deviation from this guideline will create a gap between the language and its formal semantics, reducing both intelligibility and maintainability.

Another design decision is the choice of an appropriate level of atomicity according to the language description. The SDL standard, for instance, claims the SDL level of atomicity to be on expression level, i.e. each part of an expression results in a separate step. However, this is slightly unfortunate as there are two conditions to set the level of atomicity. The first condition is time, which comes in at expression level by the *now* expression. It also comes in at transition level by timers. The second condition is communication, which comes in at transition level as well. Based on these observations, the SDL semantics has chosen the finest of these two levels, i.e. expressions. The formal semantics reflects exactly the chosen level of atomicity.

2.2 How to Establish Correctness?

The problem of turning English into mathematics and mathematical definitions into executable models raises the question of "*how to establish the correctness of the resulting formalization?*". Defining semantics by a mapping to a well-defined semantic basis, for instance, as done by the compilation of SDL actions to SAM code, is a well-known technique, but is often considered to be as dangerous as helpful. In particular, this leads us to the question of "*who validates and/or verifies the correctness of this mapping and with respect to which origin?*". Regarding the standardization process there are actually two answers:

- Considered as a starting point for the development of a comprehensive formal semantics, the current situation indeed requires validating the correctness of the formal model against the informal language definition of Z.100 (except for those parts, where the semantics is exclusively defined by the formal model). Given that there is no way to prove correctness in a strict mathematical sense, the best one can do to make the resulting model clear, concise and intelligible is avoiding formalization overhead as far as possible.

- Once the formal semantics is in use and its accuracy has been established with a sufficient degree of confidence, the formal model should become the basis for standardization. The informal description then would have the role of providing additional explanations that may be sufficient as reference whenever the ultimate degree of detail and precision is not needed. The obvious advantage is that one avoids the notorious validation problem by relying on an approach that implies correctness by construction.

Experimental Validation. Executability plays a key role in validating the formal semantics: practitioners of SDL can be given a computer program that represents the SDL formal semantics. Without detailed knowledge of the internal structure of the formal semantics, users can still validate whether a given system definition behaves as they expect. If there is a conflict between the expectation and the behavior observed, the source of this mismatch needs to be analyzed, potentially with the help of a language expert. Possible causes of such a mismatch are:

- The user expectations do not match the understanding of SDL in the community.
- The user expectations do match the understanding of SDL, but the standard says that SDL works in a different way. The language committee will have to decide whether to change the standard, or whether to educate users.
- The user expectations match the English definition of the language, but not the formal semantics, and the committee considers the English definition as the correct one. This means there is an error in the formal semantics, which must be corrected.
- The user expectation matches the formal semantics, still the executable behaves differently. That means there is an error in the tool chain producing the executable (or in the computer system executing it), which must be corrected by the authors of the tools.

Considering these potential sources of incorrectness, it becomes clear that the generated computer program should not be normative (i.e. it is not a reference implementation). Since there are many different sources of error, the standard currently says that neither the English text nor the formal semantics take precedence over one another.[3] Instead, in case of conflict, the committee must decide what the intended semantics is, leading to a revision of the standard.

2.3 How to Model a Moving Target?

Regarding the very nature of standardization as an ongoing activity, there is a high dynamics in the development and maintenance of a standard. In fact, the formal SDL semantics has been conceived in parallel to the language definition itself. While developing the formal semantics, the definition of SDL has been revised continuously

[3] In all previous versions of Z.100, the informal language definition was considered to be the more reliable part and as such took precedence over the formal semantics.

introducing a number of substantial changes and extensions. Such dynamics demands for robustness of the formalization approach as a prerequisite for practicability. Conciseness and flexibility therefore is of primary importance for the choice of the modeling framework.

3 SDL-2000

In Figure 1, an excerpt of the SDL specification of a simple system *theBank* is shown. The system is structured into the process sets *aDepartment* and *anAccount.* Initially these process sets contain one and zero processes that are instances of the process types *Department* and *Account*, respectively. Interaction between process instances and the environment occurs only in the form of asynchronous signal exchange via typed channels. In the example, bi-directional channels *C1* and *C2* are defined and associated with signal lists that are explicitly declared.

In addition to the system structure, the process type *Department* is specified in Figure 1. It basically consists of a state machine definition with the following behavior. A start transition performs some initial actions leading to state *S.* which identifies another group of transitions that start in state *S.* The transition shown in Figure 1 is triggered by an input signal *Sig1*, and when fired, will create an instance of the process set *anAccount*, send signals *Sig2* and *Sig3* and return to state *S.* The destination of these signals is determined by the connection structure. Signal *Sig1* carries an *Integer* value, which is passed as a parameter to the creation of *anAccount*.

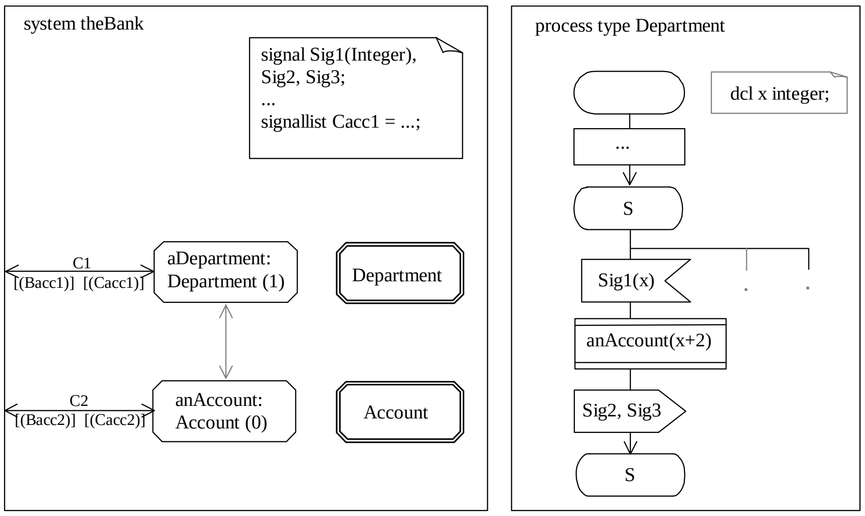

Figure 1: Excerpt of an SDL system specification

The complete language definition of SDL is given in ITU-T Recommendation Z.100 [ITU 1999b]. SDL possesses extensive structuring mechanisms to define hierarchical system architectures, which can evolve during system execution. It offers advanced control concepts like hierarchical states and exceptions. Reuse aspects can be expressed by inheritance as well as by defining packages, i.e., libraries of SDL types. Inheritance is also supported by the new data type definition part of SDL. Furthermore, SDL supports the specification of real-time behavior to some extend[4].

4 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:

- Correctness conditions: As usual, 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, a concrete graphical, and an abstract grammar are defined using Backus-Naur-Form (BNF) with extensions to capture the graphical language constructs. From such 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. This results in the following structure of the formalization (see Figure 2):

- The translation step from AS0 to AS1 is formally captured by a set of transformation rules. Transformation rules are described in so-called *model paragraphs* of Z.100, and are formally expressed as rewrite rules.
- After application of the transformations, the structure of the AS0 tree is almost the same as an AS1 tree. This means that the mapping from AS0 to AS1 is almost one-to-one.
- The correctness conditions are split into conditions on AS0 and AS1 (see Figure 2). They are formalized in terms of first-order predicate calculus.

[4] Current activities within ITU-T Study Group 10 include the development of further language constructs for dealing with real-time aspects.

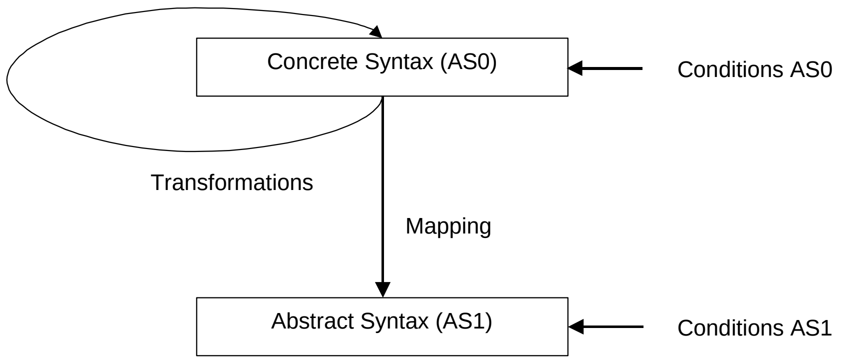

Figure 2: Static Aspects of SDL

To make this setup work, a number of auxiliary functions are defined on the syntax nodes. These functions serve as attributes of the nodes, resulting in an *attributed* abstract syntax tree. Most of the attributes are derived from other properties of the tree. Some attributes are dynamic and can be assigned.

The transformations are formalized by rewrite rules. These rules can be applied in any order as long as they are matching. However, this brings the usual problems of termination and confluence. Both problems are treated by defining a sequence of disjoint transformation phases. In each phase, the number of applicable rewrite rules is sufficiently small such that it can be efficiently checked for termination and confluence. In most cases, this check is simple because the rules apply to disjoint parts of the tree, and they never restore the conditions for their application.

5 Dynamic Semantics

The dynamic semantics is formalized starting from the *abstract syntax AS1* of SDL [see Figure 3]. Basically, there are three different aspects to deal with: *structure*, *behavior* and *data*. We restrict to those SDL specifications in AS1 that comply to the static semantics of SDL. For those specifications, a generic behavior model is derived using the ASM formalism as underlying mathematical framework for a rigorous semantic definition of the dynamic properties of SDL. The core of this model is called the *SDL Abstract Machine (SAM)*. Conceptually, we model the possible behaviors associated with a given SDL specification in terms of a set of abstract machine *runs*. The definition of the SAM splits into three main parts:

(1) basic *signal flow* concepts (signals, timers, exceptions, gates, channels),

(2) various types of ASM *agents* (modeling corresponding SDL agents), and

(3) behavior *primitives* (SAM instructions).

Technically, the definition of the SAM comes as a special *distributed real-time ASM*, a computation model explained informally at the end of this section.

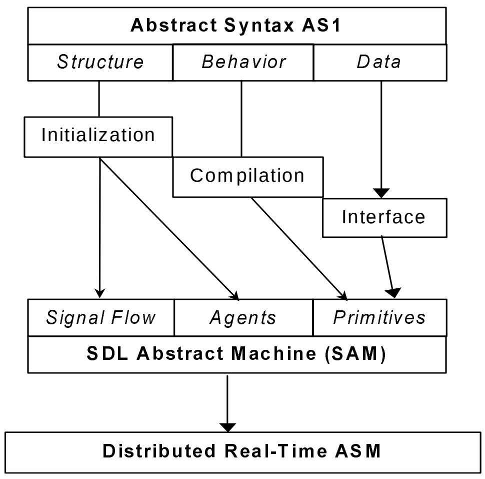

Figure 3: Overview of the Dynamic Semantics

Figure 3 outlines the treatment of structural, behavioral and data related aspects in our definition of the mapping of SDL specifications to the SAM model. We can identify three fundamental concepts:

- The *compilation* defines two compilation functions over the AST derived from an SDL specification. It yields a mapping of SDL transitions to sequences of SAM primitives. More specifically, the result of the compilation is a set of behavior primitives in combination with the control flow information needed for modeling the transitions of the SDL agents. Conceptually, the compilation is based on standard compiler techniques, therefore, it will not be elaborated further in this paper (see [ITU 2000] for details).
- The *initialization* defines a pre-initial state of an SDL system and several initialization programs. The initial system state is then reached by creating a distinguished SDL system agent, and by activating this agent in the pre-initial state. The initialization recursively unfolds the static structure of the system, creating further SDL agents as specified. In fact, the same process is initiated in the subsequent execution phase, whenever new SDL agents are created. From this point of view, the initialization merely describes the instantiation of the SDL system agent. This way, the initialization is even interleaved with the execution.
- The definition of the data semantics is encapsulated and separated from the rest of the semantics by a well-defined *interface*. The use of an interface allows us to replace the data model, if for some other application domain another data model is more appropriate than the built-in model. Moreover, also the built-in model can be changed the same way without affecting the rest of the semantics.

On top of the “logical hardware”, as defined by the SAM, the *SDL Virtual Machine (SVM)* provides typical operating system functionality (see Figure 4). As such, it provides suitable abstractions by a set of macros and functions, which determine the structure of an SDL system at runtime, the structure of agents, the selection of transitions and their firing.

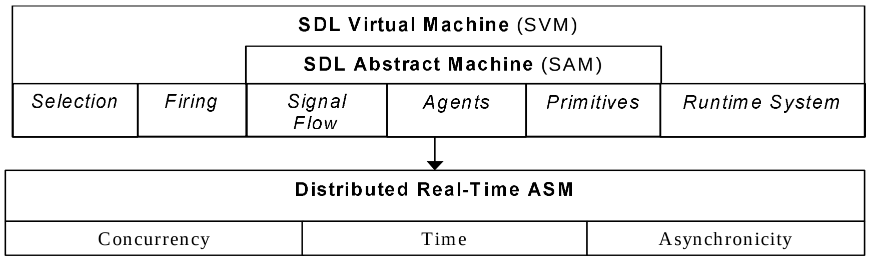

Figure 4: Overview of the Dynamic Semantics

Formal Semantic Model. The dynamic semantics associates with each SDL specification a particular *distributed real-time ASM*. This computation model is based on fairly general notions of *concurrency* and *time*; furthermore it is *asynchronous* and directly supports the abstract operational view of the informal language definition. Indeed, it is a reasonable choice for a straightforward treatment of fundamental SDL concepts. We explain here the underlying mathematical model in an informal style at an intuitive level of understanding. For a rigorous mathematical definition of the theory of Abstract State Machines, see the original literature [Gurevich 1993] and [Gurevich and Huggins 1996].

Intuitively, the computation model of distributed ASMs consists of a collection of autonomously operating ASM *agents*. Starting from a distinguished initial state, the agents perform concurrent computations interacting with each other by reading and writing shared locations of global machine states. The underlying semantic model regulates interaction between agents such that potential conflicts are resolved according to the definition of *partially ordered runs* [Gurevich 1993].

Formally, agents come as elements of a dynamically growing and shrinking universe, or domain, *AGENT*., where we associate with each agent a *program* defining its behavior in terms of some transition rule. Complex transition rules are inductively defined as compositions of guarded update instructions using simple rule constructors. We distinguish different types of agents according to different types of programs as represented by a static domain *PROGRAM*. Collectively these programs form the distributed program of a distributed ASM.

By introducing a notion of global system time and imposing additional constraints on machine runs, one obtains real-time behavior, with agents performing instantaneous actions in continuous time. In this model, agents fire their rules at the moment they are enabled, i.e. react immediately (see 5.1.1).

5.1 SDL Abstract Machine

The SDL Abstract Machine (SAM) consists of the following parts:

- *Signal Flow Model*, which defines a uniform treatment of signal flow related aspects, in particular, the communication of 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.

We start by introducing the underlying notion of real time.

5.1.1 Real Time

SDL is promoted for the specification and design of distributed real-time systems. However, its actual support for dealing with real-time behavior is essentially limited to delaying mechanisms and timeout mechanisms. Such instruments are descriptive rather than prescriptive or regulating; in particular, there is no way of enforcing a desired timing behavior. Based on the underlying notion of global system time, signal communication over channels as well as operations of process instances by default are subject to arbitrary but finite delays. SDL provides a simple timer concept for handling timeout events.[5]

We introduce a discrete notion of time for the abstract representation of global system time as represented by the SDL expression **now**.

Z.100, Section 12.3.4.1 Now Expression

The **now** expression is an expression which accesses the system clock variable to determine the absolute system time.

Our notion of time reflects the view that one can only observe, but not control, how physical time evolves. Time values are represented as real numbers by the elements of a linearly ordered domain *TIME*. We can assume that *TIME* ⊆ *REAL* and define the relation "≥" on time values through the corresponding relation on real numbers. A domain *DURATION* represents finite time intervals as differences between time values.

domain *TIME*
domain *DURATION*

The global system time, as measured by some discrete clock, is represented by a so-called *monitored*, nullary function *now* taking values in *Time*. A monitored

[5] Current activities within ITU aim at extensions of SDL that allow for more sophisticated specification and analysis of timing behavior [ITU 1999a]. Conceptually, our model of real time underlying the formal definition of SDL takes such extensions already into account.

function represents an interface to the external world and, as such, may have different values in different states depending on actions and events in the system environment. We can assume here (as an integrity constraint on valid runs) that the values of *now* change monotonically over machine runs.

monitored *now* : *TIME*

Similar to the real-time ASM model introduced in [Gurevich and Huggins 1996], ASM agents in our model react instantaneously, i.e. they fire their rules as soon as they reach a state in which the rules are enabled. Strictly speaking, one must assume here some non-zero delay to preserve the causal ordering of actions and events; though, such minimal delay is immaterial from a practical perspective of view. Computation steps of agents are *atomic* but, nevertheless, they are considered as time-consuming actions such that the following condition holds:

Z.100, Section 11.12.1 Transition Body

An undefined amount of time may pass while an action is interpreted. It is valid for the time taken to vary each time the action is interpreted. It is also valid for the time taken to be the same at each interpretation or for it to be zero (that is, the result of **now** is not changed).

From an SDL point of view, a purely monitored time is not sufficient. In the current SDL standard, there are non-delaying channels that are supposed to transfer their signals without time delay. In the extensions discussed for a timed version of SDL, specific restrictions on the time consumption can be placed on all kinds of actions. Conceptually, this is solved by defining integrity constraints on the time progress, thus restricting the set of possible runs.

5.1.2 Signal Flow Model

The signal flow model defines communication primitives for signal-based communication between SDL agents. In particular, it defines the transportation of *signals* through delaying and non-delaying *channels* connecting agents via their *gates*. Furthermore, this model defines *timers* and *exceptions* as special kinds of signals. Thus, the signal flow model forms the core of SDL's asynchronous communication model.

There is a derived domain *SIGNAL* representing the set of signal types as declared by an SDL specification. *SIGNAL* also includes timers and exceptions, which are modeled as signals, too. Dynamically created signal instances are elements of a derived domain *SIGNALINST*. Functions on signals are *signalSender*, *toArg*, and *viaArg* yielding the sender process, the destination, and optional constraints on admissible communication paths, respectively. 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.

Input Gates and Output Gates. Exchange of signals between SDL agents (such as processes, blocks or a system) and the environment is modeled by means of *gates* from a controlled 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 need not reach their destination instantaneously, but may be subject to delays. Therefore, it must be possible to send signals to arrive in future. Although those signals are not available at their destination before their arrival, they are already associated with their destination gates. That is, a gate must be capable of holding signals that are in transit (have not yet arrived). Hence, to each gate a possibly empty *signal queue* is assigned [see Figure 5]. One can now represent the relation between signals and gates in a given SAM state by means of a dynamic function *schedule* defined on gates: *schedule* specifies, for each gate *g* in *GATE*, the corresponding *signal arrivals* at *g.*

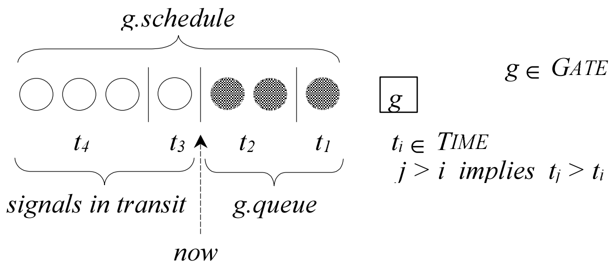

Figure 5: Signal instances at a gate

Channel Behavior. SDL channels consist of either one or two unidirectional *channel paths.* In the SAM model, each channel path is identified with an object of a derived domain *LINK*. The elements of *LINK* are SAM agents. Their behavior is defined through LINKPROGRAM consisting of a single rule FORWARDSIGNAL as defined below. Think of *LINK* elements as point-to-point connection primitives for the transport of signals.

A link agent *l* performs a single operation: signals received at gate *l.from* are forwarded to gate *l.to.* That means, *l* permanently watches *l.from* waiting for the next deliverable signal in *l.from.queue*. Whenever *l* is applicable to a waiting signal *si* (as identified by the *l.from.queue.head*), it attempts to remove *si* from *l.from.queue* in order to insert it into *l.to.schedule*. This attempt needs not necessarily be successful as, in general, there may be several link agents competing for the same signal *si.*

But, how does a link agent *l* know whether it is applicable to a signal *si*? Now, this decision does of course depend on the values of *si.toArg*, *si.viaArg*, *si.signalType* and *l.with*. In other words, *l* is a legal choice for the transportation of *si* only, if the following two conditions hold: (1) *si.signalType* $\in$ *l.with* and (2) there exists an applicable path connecting *l.to* to some final destination matching with the address information and the path constraints of *si*. Abstractly, this decision can be expressed using a predicate *Applicable,* where we refer to [ITU 2000] for the definition of further details.

```
FORWARDSIGNAL ≡
    if $Self.from.queue \neq empty$ then
        let $si = Self.from.queue.head$ in
            if $Applicable(si.signalType, si.toArg, si.viaArg, Self.from, Self)$ then
                DELETE($si, Self.from$)
                INSERT($si, now+Self.delay, Self.to$)
                $si.viaArg := si.viaArg \setminus \{ Self.from.nodeAS1.node.AS1ToId,$
                            $Self.nodeAS1.node.AS1ToId\}$
            endif
        endlet
    endif
```

Timers and Exceptions. A particular 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. Below we present the solution for the timer model.

A static domain *TIMER* represents the set of all timer types as identified by the AST of a given SDL specification. Another dynamic domain *TIMERINST* holds the respective timer instances created at run time.

$$TIMER =_{\text{def}} \{tid \in Identifier : tid.idToNodeAS1 \in Timer\text{-}definition\} \subset SIGNAL$$

$$TIMERINST =_{\text{def}} PID \times TIMER \times VALUE^* \subset SIGNALINST$$

The information associated with timer signals is accessed using the functions defined on *SIGNAL*.

Active Timers

SDL defines a timer to be *active* if the timer has expired but the resulting timer signal has not yet been consumed by the related process. Each *active* timer is represented by a corresponding timer signal in the schedule associated with the input port of the related process instance. To indicate whether a timer instance *tmi* is active or not, there is a corresponding derived predicate *Active*:

$$Active(tmi{:}TIMERINST): BOOLEAN =_{\text{def}} tmi \in Self.inport.schedule$$

Timer Operations

There are two operations on timers as specified below. The process agent to which the timer belongs executes these operations one at a time. A static function *duration* is used to represent default duration values as defined by an SDL specification under consideration. A default duration value is used to calculate the expiration time if not specified otherwise.

static *duration*: TIMER → DURATION

```
SETTIMER(tm: TIMER, vSeq : VALUE*, t: TIME) ≡
    let tmi = mk-TIMERINST(Self.self, tm, vSeq ) in
        if t = undefined then
            Self.inport.schedule := insert(tmi, now+ tm.duration,
                                delete(tmi, Self.inport.schedule))
            si.arrival := now + tm.duration
        else
            Self.inport.schedule:=insert(tmi,t,delete(tmi, Self.inport.schedule))
            si.arrival := t
        endif
    endlet

RESETTIMER(tm: TIMER, vSeq : VALUE*) ≡
    let tmi = mk-TIMERINST(Self.self, tm, vSeq ) in
        if Actice(tmi) then
            DELETE(tmi, Self.inport)
        endif
    endlet
```

5.1.3 SAM Agents

SAM agents define the SDL concepts ‘SDL channel’, ‘SDL agent’, and ‘SDL agent set’ (see Figure 6). The state information of an SDL agent is collected in an *agent control block*. The agent control block is partially initialized when an SDL agent (set) is created, and completed/modified during its initialization and execution. Since part of the state information is valid only during certain activity phases, the agent control block is structured accordingly.

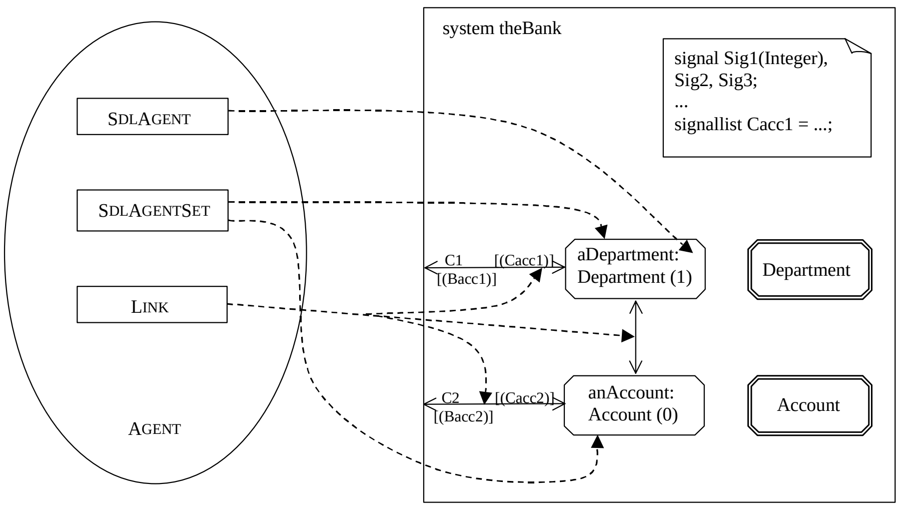

Figure 6: SAM Agents

The structure of the agent's *state machine* is directly modeled, and built up during agent initialization. The state machine structure is exploited in the execution phase, when transitions are selected, and states are entered or left.

5.1.4 Behavior Primitives

Primitives are modelled as actions with attached labels. An action is an element of a derived domain *ACTION*, which is defined as the disjoint union of basic action domains such as *CREATE* or *OUTPUT.* An element of a basic action domain provides the required information in order to fire and evaluate the action.

$$PRIMITIVE =_{\text{def}} LABEL \times ACTION$$
$$ACTION =_{\text{def}} CREATE \cup OUTPUT \cup ...$$

The *create* primitive specifies the dynamic creation of an SDL agent. An action of type *CREATE* is defined as a tuple consisting of an agent definition, a sequence of value labels, and a continue label. An *output* primitive is defined as a tuple consisting of a signal, a sequence of value labels, a value label, a set of identifier which determines the receiver of the message, and a continue label.

$$CREATE =_{\text{def}} Agent\text{-}definition \times VALUELABEL^{*} \times CONTINUELABEL$$
$$OUTPUT =_{\text{def}} SIGNAL \times VALUELABEL^{*} \times VALUELABEL \times VIAARG \times CONTINTUELABEL$$

Firing of actions is defined by the selection and evaluation of the corresponding SAM primitives, resulting from the compilation. The function *currentLabel* uniquely identifies a behavior primitive; therefore, the choose in the rule below does not introduce non-determinism. The evaluation of the macro FIREACTION finally leads to an update set that is executed in a single state transition of the distributed real-time ASM.

```
FIREACTION ≡
        choose $b$: $b \in BEHAVIOR \wedge b$.s-$LABEL = Self.currentLabel$
            EVALUATE ($b$.s-$PRIMITIVE$)
```

The evaluation of an action is defined by the macro EVALUATE. Depending on the action, a specific macro, such as EVALCREATE or EVALOUTPUT, is selected.

```
EVALUATE ($a$: $ACTION$) ≡
        if $a \in CREATE$ then EVALCREATE ($a$)
        if $a \in OUTPUT$ then EVALOUTPUT ($a$)
        ...
```

The macro EVALCREATE defines the evaluation of the *create* primitive. As part of this evaluation, the SDL agent set *sas* where an additional agent is to be added is determined (first line of macro EVALCREATE). For this agent set, it is checked whether there is a maximum number of SDL agents, and whether this maximum number has not yet been reached. In the latter case or if there is no maximum number, a new SDL agent of the type defined in the action is created (macro CREATEAGENT, see below). In addition, *currentLabel* is set to the next action.

```
EVALCREATE (a: Create) ≡
        let sas = take({sas ∈ SDLAGENTSET: sas.nodeAS1 =
            a.s-Agent-definition}) in
            if sas.nodeAS1.s-Number-of-inst.s-Max-number ≠undefined then
                let n = |{sa ∈ SDLAGENT: sa.owner = sas}| in
                    if n < sas.nodeAS1.s-Number-of-inst.s-Max-number then
                        CREATEAGENT (sas,Self.self,
                            sas.nodeAS1.s-Agent-type-definition)
                    else
                        Self.offspring := null
            else
                CREATEAGENT (sas,Self.self, sas.nodeAS1.s-Agent-type-definition)
        Self.currentLabel := a.s-CONTINUELABEL
```

To create an agent, the controlled domain *AGENT* is extended. The control block (see Section 5.2) of this new agent is initialised. An input port for receiving signals from other agents is created and attached to the new agent. Setting of agent modes and assignment of the SAM program AGENT-PROGRAM (see Section 5.2) complete the creation of the agent.

```
CREATEAGENT (ow: SDLAGENTSET,pa: PID,atd: Agent-type-definition) ≡
        extend AGENT with sa
            INITAGENTCONTROLBLOCK (sa, ow, pa, atd)
            CREATEINPUTPORT (sa)
            sa.agentMode1 := initialisation
            sa.agentMode2 := initialising1
            sa.program := AGENT-PROGRAM
        endextend
```

The output primitive specifies the sending of a signal. A signal output operation causes the creation of a new signal instance, and is defined by the macro SIGNALOUTPUT.

```
EVALOUTPUT (a: OUTPUT) ≡
        SIGNALOUTPUT(a.s-SIGNAL, values(a.s-VALUELABEL-seq, Self),
            value(a.s-VALUELABEL, Self), a.s-VIAARG)
            Self.currentLabel := a.s-CONTINUELABEL
```

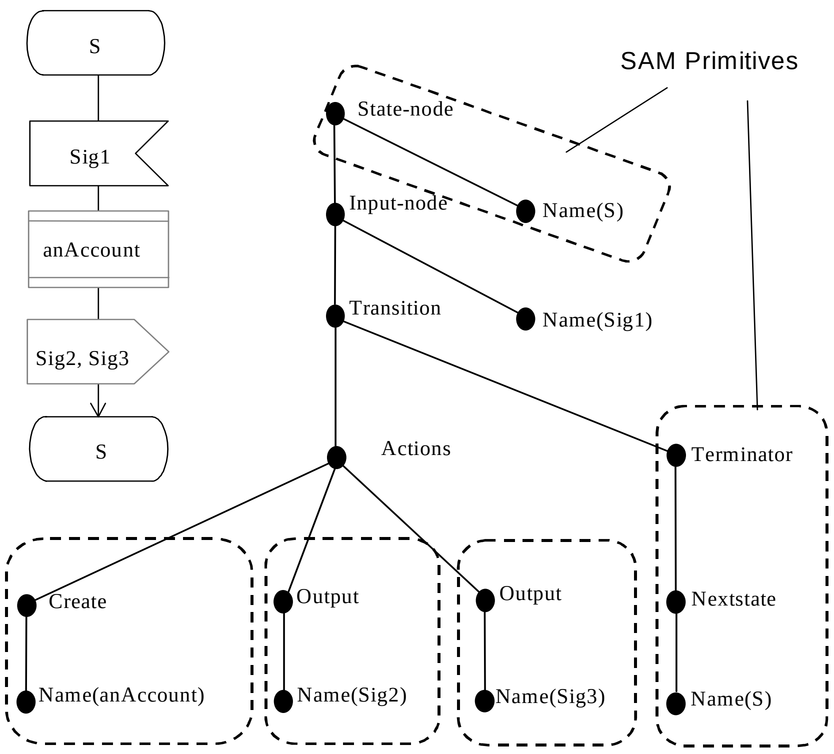

Figure 7: AST and compilation of SAM primitives

An excerpt of the Abstract Syntax Tree for the example in Figure 1 with some of the involved SAM primitives is shown in Figure 7.

5.2 SDL Virtual Machine

The *SDL Virtual Machine* (*SVM*) provides typical operating system functionality on top of the logical hardware as defined by the SAM. Under the control of the SVM, the programs LINK-PROGRAM, AGENT-PROGRAM and AGENT-SET-PROGRAM, which are associated with link agents, SDL agents and SDL agent sets, respectively, are executed. The SVM defines suitable abstractions by a set of macros and functions, which determine the structure of an SDL system at runtime, the structure of agents, the selection of transitions and their firing, finally leading to update sets. We will now sketch some aspects of the SVM, focussing on SDL agents and AGENT-PROGRAM. The complete definition of the SVM, comprising approximately 1200 lines of ASM definitions, is given in [ITU 2000].

SDL agents are the most complex active components of an SDL system at runtime. Therefore, we distinguish several activity phases, which in turn have several levels of sub phases. Phases and sub phases are identified by corresponding control states, as shown in the control state graphs of Figure 9, Figure 10, and Figure 11. On the top level, the phases *initialization* and *execution* are distinguished (see Figure 8). 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, connection structure and further agents, is created in consecutive sub phases. Then, the agent enters the execution phase, where it remains until its termination.

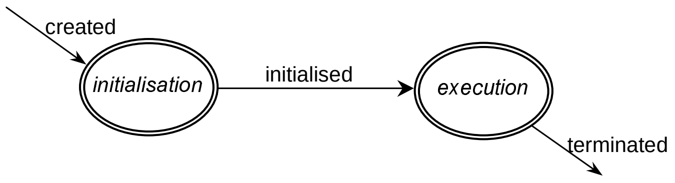

Figure 8: Control state graph of SDL agents (level 1)

With each SDL agent, an *agent control block*, which keeps all information necessary to determine the current activity state of that agent, is associated. Formally, the agent control block is represented as a group of controlled functions, including:

- *agentMode1*: top level control state of an SDL agent; depending on the activity phase, there can be up to 4 additional levels of control states, represented by further functions; identifying the control states of agents is good practice in operating systems design;
- *currentLabel*: during the firing of a transition or the evaluation of an expression, this function identifies the currently executed action, thus taking the role of the agent's program counter;
- *inport*: input queue of the agent containing the sequence of signals that have been sent to this agent and are waiting for consumption;
- *signalChecked*: the signal of the input queue that is currently examined during the transition selection process;
- *transitionChecked*: transition that is currently examined during the transition selection process;
- *self*: agent id, a unique identification of this agent as defined in SDL; this agent id represents an SDL function and therefore should be distinguished from the ASM function *Self*;
- *sender*: agent id of the sender of the last consumed signal.

The behavior of SDL agents is defined by the program AGENT-PROGRAM (see below). Depending on the current top level control state represented by the controlled function *agentMode1*, a macro defining the corresponding activities is selected. Macros are hierarchically structured and thus provide useful abstractions. By using the ASM function *Self*, the agent control block of the SDL agent that is running this program is accessed. In certain situations of the execution phase, SDL requires the sequentialization of a group of SDL agents. This is formalized by defining an execution right that can be owned by at most one agent of a group.

```
AGENT-PROGRAM
if Self.agentMode1 = initialization then
    INITAGENT
if Self.agentMode1 = execution then
    if Self.ExecRightPresent then
        EXECAGENT
    else
        GETEXECRIGHT
```

Execution of agents is modeled by alternating phases, namely transition selection and transition firing, preceded by a start phase. To distinguish between these phases, corresponding control states are defined (see Figure 9). When an agent is in sub phase *selectingTransition* (*agentMode2*), it attempts to select a transition, obeying a number of constraints. In sub phase *firingTransition*, a previously selected transition is fired. Formally, this is defined by the macro EXECAGENT:

```
EXECAGENT ≡
        if Self.agentMode2 = startPhase then
            EXECUTIONSTARTPHASE
        if Self.agentMode2 = selectingTransition then
            SELECTTRANSITION
        if Self.agentMode2 = firingTransition then
            FIRETRANSITION
        if Self.agentMode2 = stopping then
            STOPPHASE
```

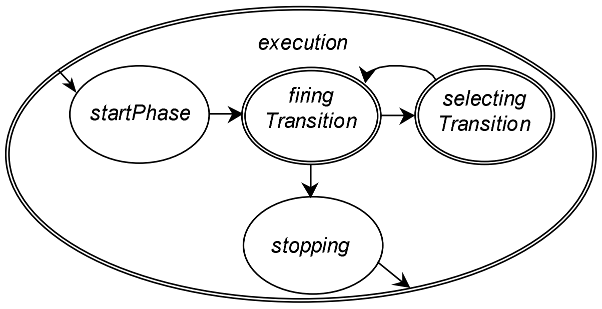

Figure 9: Control state graph of SDL agents: execution (level 2)

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). With the incorporation of inheritance in SDL-92, this became slightly more complicated (Z.100, Section 8.3.3), but was resolved by a transformation step 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). 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. Figure 10 gives a flavour of this complexity, as the refinement of the control state *selectingTransition* is shown. This is not the end, as the refinement may go 2 levels further.

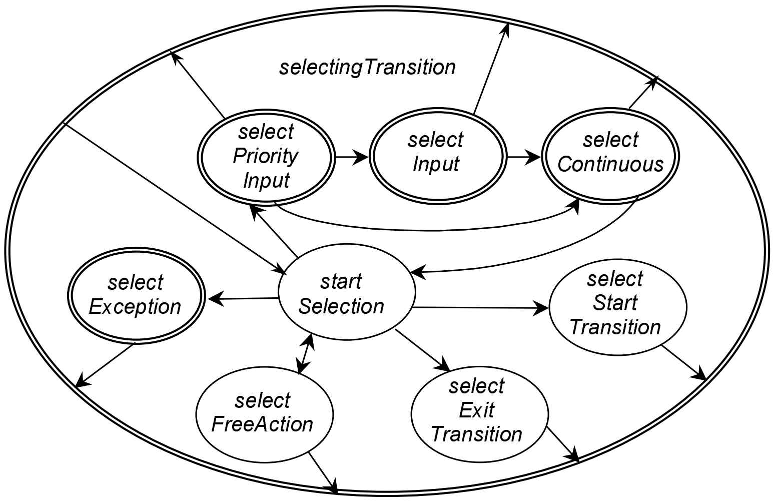

Figure 10: Control state graph of SDL agents: transition selection (level 3)

In control state *selectingTransition* (see Figure 10), an SDL agent searches for a fireable transition. Z.100 imposes certain rules on the search order. For instance, priority input signals have to be checked before ordinary input signals, and these have in turn to be checked before continuous signals can be consumed. Furthermore, a transition emanating from a substate has higher priority than a conflicting transition emanating from any of the containing states. Finally, redefined transitions take precedence over conflicting inherited transitions. These and further constraints have to be observed when formalising the transition selection.

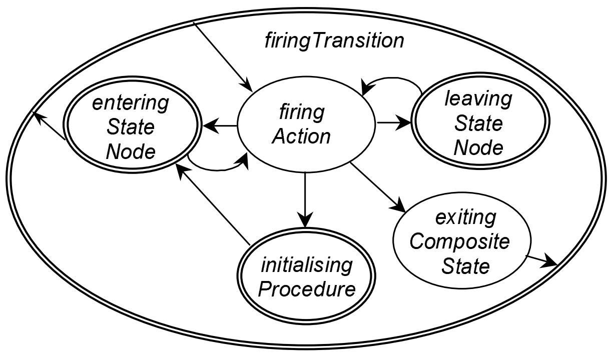

Figure 11: Control state graph of SDL agents: transition firing (level 3)

Firing of a transition is decomposed into the firing of individual actions, which 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.

```
FIRETRANSITION ≡
        if Self.agentMode3 = firingAction then
            FIREACTION
        if Self.agentMode3 = leavingStateNode then
            LEAVESTATENODES
        if Self.agentMode3 = enteringStateNode then
            ENTERSTATENODES
        ...
```

Firing of actions including the execution of behaviour primitives has been addressed in Section 5.1.4. This ends the sketch of the Java Virtual Machine.

5.3 Data

Integration of the semantics of data into the dynamic semantics is done by means of a functional interface. With such an interface, evaluations in the data part will have no side effects. This allows to express the data semantics with any formalism suitable for expressing functions, and therefore, it potentially allows to replace the data semantics with a different model.

The data semantics of SDL consists of the following components:

- Association of variables with values,
- Evaluation of expressions,
- Definition of predefined data, and

- Definition of user-defined data.

To provide the notion of variables that are bound to expressions, the data semantics provides a domain *STATE*, whose elements represent associations of variables and values. Values, in turn, are elements of a domain *VALUE*.

In SDL, two different kinds of values are defined: those based on an object type have reference semantics; those based on a value type have value semantics. Since objects can be shared across SDL process agents, all agents within a process agent share the same *STATE*. Whenever a new agent or procedure is created, the state of the creator is extended with the variable bindings of the new agent. Since each multiple processes or procedures may use the same variable names within a state, they need to identify their portion of the state using a *STATEID*. To access and modify variables, the following functions are provided:

$$\textit{assign}: \textit{Variable-identifier} \times \textit{VALUE} \times \textit{STATE} \times \textit{STATEID} \rightarrow \textit{STATEOREXCEPTION}$$

$$\textit{eval}: \textit{Variable-identifier} \times \textit{STATE} \times \textit{STATEID} \rightarrow \textit{VALUE}$$

When assigning a value to a variable, the variable name, the value, the current state and the state id of the process or procedure must be provided. The function *assign* produces a new state, which is then assigned to a controlled function. In some cases, assignment may produce an exception instead of a state. To access a variable, the variable identifier, the state, and the state id are required.

Evaluation of expressions normally involves application of an operator. The operator can be either predefined or user-defined. If the operator is predefined, the function

$$\textit{compute}: \textit{PROCEDURE} \times \textit{VALUE}^* \rightarrow \textit{VALUEOREXCEPTION}$$

can be used to evaluate the operator application. This function is defined in terms of a number of auxiliary functions which provide the operator definitions for the predefined types.

If the operator is user-defined, a procedure definition must be located for the operator call. Since SDL supports polymorphism and late binding, the selection of the procedure occurs dynamically. This selection is provided with the function

$$\textit{dispatch}: \textit{PROCEDURE} \times \textit{VALUE}^* \rightarrow \textit{Identifier}$$

The actual execution of the procedure being called is controlled by the SVM.

Using a functional interface for the data semantics is feasible for most aspects, with one notable exception: Using object types involves operations that cannot be fully described by means of functions alone. In particular, creation of an object creates a new identity for the object, which is different from any other object identity. This identity is modeled by the domain *OBJECTID*. ASMs support domains that can be extended dynamically. However, extending a domain is an operation with side effects. Therefore, the data semantics requires a function *mkObjectId*, which returns a different value in each state of system execution.

6 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. Different from 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.

The following aspects of SDL have been directly influenced by the formalization:

- Formalization of inheritance and object-orientation

  In 1992, object-orientation was introduced into SDL. However, the first attempt to describe object-orientation was to give transformations how to map the new type-based concepts onto the old concepts. This is of course a lot of transformations and therefore the essence of the SDL types is not easily understood. For the SDL-2000 formalization, we insisted on giving a direct semantics for object-orientation in order to have those concepts readily available for description. This was a major change in the language description and also revealed several design flaws of the whole object-orientation that could not have been discovered in the transformation approach. We even inserted new transformations making the non-typebased concepts derived concepts, where an implicit type was introduced.

- Implicit transitions

  In SDL, there is the understanding that signals that are not explicitly handled are implicitly discarded. This situation was covered in SDL-92 using a transformation, which inserted an implicit transition to discard the signal. However, this approach cannot be used in SDL-2000, because of the introduction of composite states. This led to a direct formal semantics for signal discarding.

- Identifier resolution

  SDL has a very complex identifier resolution scheme, because sometimes identifiers are resolved with regard to the context. It is possible in SDL to have the same name for different entities of different kinds, e.g. a signal can have the same name as a data type. Moreover, operators can have the same names and differ only in their signatures, e.g. different versions of the operator "+". When formalizing the Z.100 text about resolution, some problems occurred, finally leading to the resolution rules being redrafted and stated more clearly.

- Input triggers

  In SDL, it is possible to not only have inputs of signals but also to have conditions for the reception of signals, to have prioritized inputs and to have conditions to trigger a transition. The semantics of these concepts has been defined by transformations in previous versions of SDL: these inputs were transformed into the usual signal inputs using several implicitly defined local signals. However, it was never clear what it meant to have all of these concepts within one specification. So we decided to have a direct semantics for these concepts as well. This reduced the informal description and made the formal description very concise.

- Evaluation of decisions

  When evaluating a decision, user-defined operators and methods can be invoked to select the matching decision answer. Since such methods may have side-effects, or may raise an exception, the evaluation order of those operator applications may matter for the outcome of the decision. Originally, the informal language definition was silent on the issue of evaluation order. When this underspecification was detected during definition of the formal semantics, an explicit description of the evaluation procedure for decisions was added to the informal definition.

7 Experience and Outlook

Regarding the very 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 for robustness of the formalization approach as a prerequisite for practicability. Conciseness and flexibility therefore were of primary importance for the choice of the formal modeling framework.

Despite of the richness of SDL, the formal model is intelligible and maintainable. This is essentially achieved through three properties, namely: the compiler-based approach, the organization of the abstract machine model, and the consequent use of parameterized ASM rule macros.

Comparing the two parts of the SDL semantics (static and dynamic), the static semantics is twice as large (in pages) as the dynamic semantics. This is already the result of an improved balancing between these two parts, as concepts that previously had been defined by means of in-language transformations now are part of the dynamic semantics.

The notations and concepts used in the formal definition of SDL were chosen to allow automatic processing by means of computer programs. For a subset of SDL, whose definition also uses a subset of the notations, tools have been created that allow the execution of an SDL system in the ASM workbench. Work is still in progress on applying these tools to the complete definition of SDL.

Finally, it should be stressed that the definition of the formal semantics has not just been an academic exercise, but took 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 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 here is an important step in this direction.

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, and Rick Reed and Thomas Weigert as responsible SDL Rapporteurs of ITU-T Study Group 10/Q6. Furthermore, we thank the anonymous referees for their especially valuable comments. Finally, we thank Telelogic, Ericsson, and Motorola for their financial support of part of this work.

References

[Eschbach et al. 2000] R. Eschbach, U. Glässer, R. Gotzhein and A. Prinz. “On the formal semantics of SDL-2000: a compilation approach based on an Abstract SDL Machine”. In Abstract State Machines - Theory and Applications. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele (Eds.), Lecture Notes in Computer Science, Vol. 1912, Springer-Verlag, 2000

[Glässer et al. 1999] U. Glässer, R. Gotzhein, A. Prinz: “Towards a New Formal SDL Semantics Based on Abstract State Machines”, in: R. Dssouli, G.v. Bochmann, Y. Lahav (Eds.), SDL’99 – The Next Millennium, Proc. Of the 9th SDL Forum, Elsevier Sciences B.V., July 1999

[Gurevich 1993] Y. Gurevich. “Evolving Algebras 1993: Lipari Guide”. In E. Börger, editor, Specification and Validation Methods, pages 9-36, Oxford University Press, 1995

[Gurevich and Huggins 1996] Y. Gurevich and J. Huggins: “The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions“. In *Proc. of CSL’95*, volume 1092 of LNCS, pages 266-290, 1996

[ITU 1999a] ITU-T Rapporteur, “Question 6/10. Introduction of Time Semantics in SDL“. Temporary Document 41, Geneva, November 1999

[ITU 2000] ITU-T Recommendation Z.100 Annex F: SDL Formal Semantics Definition, International Telecommunication Union, Geneva, 2000

[ITU 1999b] ITU-T Recommendation Z.100: Languages for Telecommunications Applications - Specification and Description Language (SDL), International Telecommunication Union, Geneva, 1999

[ITU 2001] SDL Formal Semantics Project. ITU-T Study Group 10: SDL Semantics Group. URL: http://rn.informatik.uni-kl.de/projects/sdl/