ASM Semantics of SDL: Concepts, Methods, Tools

Uwe Glässer
Heinz Nixdorf Institut
Universität-GH Paderborn, Fürstenallee 11, 33102 Paderborn, Germany
Phone: +49 5251 606516, Fax: 606502, Email: glaesser@uni-paderborn.de

Abstract

Based on the ITU-T Recommendation Z.100 [1], also known as SDL-92, we define in [2] a formal semantic model of the dynamic properties of Basic SDL in terms of an *Abstract State Machine (ASM).* More precisely, we use the concept of *multi-agent real-time ASM* [3] as a semantic platform on top of which we construct our abstract mathematical definition. The resulting interpretation model is not only mathematically precise but also reflects the common understanding of SDL (e.g. as presented in the literature [4, 5, 6]) in a direct and intuitive manner; it provides a *concise* and *understandable* representation of the complete dynamic semantics of Basic SDL. Moreover, the model can easily be extended and modified—a particularly important aspect for an evolving technical standard.

Keywords
Abstract State Machines, Basic SDL, formal semantics, operational semantics

1 INTRODUCTION

Abstract State Machines (ASMs) [7]—formerly called *Evolving Algebras* [8]—combine declarative concepts of first-order logic with the abstract operational view of transition systems in a unifying framework for mathematical modeling of discrete dynamic systems. The underlying computation model constitutes a simple yet powerful semantic basis to deal with *concurrent* and *reactive* behaviour in a direct and intuitive way; it *naturally* enables operational interpretations of high-level system specifications and thereby facilitates machine-supported analysis and validation (e.g. through simulation) of the resulting models.

Numerous ASM applications (see for instance the annotated ASM bibliography [9]) to real-life systems engineering problems—in particular, formal semantics of programming languages, modeling languages and protocols—contributed to establish general abstraction principles needed to cope with the complexity of large systems [10].[1] In contrast to many traditional formal methods with a monolithic character, the ASM method is open to be combined with other (e.g. application domain specific) modeling techniques, thus providing additional flexibility.

Based on [2], we present here a formal semantic model of Basic SDL according to the *ITU-T Recommendation Z.100* [1], also known as SDL-92. More precisely, we focus on the dynamic semantics of Basic SDL, which we describe in terms of an abstract interpretation model using the concept of *multi-agent real-time ASM* as a formal basis. The resulting description is intended to be a first step towards a formal documentation of complete SDL which is not only *mathematically precise* but does also reflect the common understanding of SDL (as presented in the literature [4, 5, 6]) in a direct and intuitive manner.

Aiming at a coherent and consistent embedding of a formal semantic model into Z.100, it is important to clearly separate the concerns of *specification* and *verification.* There are good pragmatic reasons (see Sect. 4 of [10]) to avoid in a standardization context the additional overhead of formalisms which are mainly intended to support mechanization of verifications.

[1] For a comprehensive overview on ASM applications, introductory material and supporting tools see also the following two URLs: `http://www.uni-paderborn.de/cs/asm/` and `http://www.eecs.umich.edu/gasm/`.

It is probably worth noticing that the meta-modeling concept we propose for SDL was already employed in our previous work [11] on a formal semantics of the IEEE hardware description language VHDL [12]. The resulting semantic model of complete VHDL'93 is not only the most comprehensive model of VHDL known so far, but is at the same time also considered to be a particular concise and understandable description of this fairly complex modeling language.

Formal Semantics of SDL Z.100 does already come together with a complete formal model of SDL based on a combination of *Meta-IV* and *CSP* (Annex F to [1]). However, the result is not satisfying as the natural language descriptions of Z.100 take precedence over the formal semantics. One reason why Z.100 does not rely on its own formal model is probably the fact that this model is hardly usable because of its size: the entire formal definition is more than 500 pages.

In fact, there is a considerable variety of formal semantic models of SDL which have been developed using various formal methods. Among the approaches which are mainly concerned with analysis and verification of SDL specifications are the following. In [13], Bergstra and Middleburg define a *process algebra* semantics of a restricted version of Basic SDL, which they call φSDL. Broy [14], Holz and Stølen [15] use the *stream processing functions* of *FOCUS* to model subsets of SDL. Fischer and Dimitrov propose *extended Petri nets* as a formal basis to verify SDL protocol specifications [16]. Rinderspacher employs a *term-rewriting system* based modeling concept [17]. Some of these approaches consider only a relatively small subset of SDL ignoring certain essential features (like dynamic process creation or basic structuring concepts). An approach aiming at a more comprehensive semantic model of SDL was proposed by Fischer, Lau and Prinz through their definition of BSDL (*Base SDL*) using *Object-Z* [18, 19].

This paper is structured as follows. *Section 2* briefly explains the underlying ASM concepts and related notions as far as these are required here. *Section 3* introduces our formal semantic model of Basic SDL in terms of an abstract SDL machine. Besides the encoding of SDL objects in the abstract machine model, the particular focus here is on the behaviour of channels, operations on signals and timer operations. To stress the practical relevance of the approach, *Sect. 4* outlines the role of tools for machine-based analysis and execution of ASM models. Finally, *Sect. 5* contains the conclusions.

2 ABSTRACT STATE MACHINES

In order to make the paper self-contained as far as possible, we briefly address in an informal style some basic ASM concepts (Sect. 2.1), general abstraction principles (Sect. 2.2) and the particular ASM class (Sect. 2.3) employed here. For a rigorous definition of the mathematical foundations of ASMs, we refer however to [8, 7]—though it should be possible to get a sufficiently detailed understanding without consulting the formal definition.

2.1 The Basic Model

An ASM $\mathcal{M}$ is defined over a given *vocabulary* Υ by its *program* P and its *initial state* S_0. The vocabulary (or signature) Υ is a finite collection of function names and predicate names, each of a fixed arity. Names in Υ may be marked as *static* indicating that they have a fixed interpretation regardless of the state of $\mathcal{M}$ (whereas non-static names may have different interpretations in different states of $\mathcal{M}$).

States States of $\mathcal{M}$ are variants of first-order *structures*. They contain functions but *no* relations (rather they express relations as characteristic functions[2]). A state defines an interpretation of the names in Υ on the underlying base set of $\mathcal{M}$.

Formally, all functions associated with a given state S of $\mathcal{M}$ are total functions on the base set. It is nevertheless possible to imitate *partial* functions by marking "undefined" values using a distinguished element *undef*, except for predicates. By definition, predicates may only have the values *true* or *false*.

Unary predicates have a special role: they represent sets, also called *universes* or *domains*[3], and thereby form *many-sorted* structures on top of elementary structures.

[2]The treatment of relations as Boolean-valued functions considerably simplifies the underlying computation model (without loss of generality) by employing a uniform notion of *local* updates on structures.

[3]A domain introduces a certain category of objects and may be completely abstract (not imposing any restrictions on theses objects). Alternatively, objects may be assumed to have certain properties or to be in certain relations with other objects. In the latter case, the properties and relations have to be stated separately through additional *integrity constraints* to be associated with the domains.

Every state S of $\mathcal{M}$ has an infinite *reserve* representing additional resources as required to extend domains dynamically. Intuitively, the reserve represents the outside world.

Programs P is defined in terms of *transition rules* specifying local *updates* on states. A transition rule R consists of *update instructions* of the basic form

$$f(t_1, \ldots, t_n) := t_0 \quad (n \geq 0),$$

where $f(t_1, \ldots, t_n)$, t_0 are terms over Υ identifying the location to be changed and the new value to be assigned. The construction of complex transition rules out of basic update instructions is inductively defined by means of ASM rule constructors as will be explained in the ASM model of SDL.

Computations Computations are modeled through (finite or infinite) *runs* of $\mathcal{M}$ as sequences of state transitions of the form

$$S_0 \overset{\Delta_{S_0}(P)}{\longrightarrow} S_1 \overset{\Delta_{S_1}(P)}{\longrightarrow} S_2 \overset{\Delta_{S_2}(P)}{\longrightarrow} \cdots$$

such that S_{i+1} is obtained from S_i, for $i \geq 0$, by firing $\Delta_{S_i}(P)$ on state S_i, where $\Delta_{S_i}(P)$ denotes the update set computed by the program P of $\mathcal{M}$ on S_i.

2.2 General Abstraction Principles

A key aspect in mathematical modeling of large systems is the ability to cope with complexity. The concept of ASM thus comes together with general abstraction principles allowing a systematic analysis and validation of relevant system properties at any desired abstraction level. This freedom of abstraction in combination with the consequent use of information hiding and interface mechanisms considerably simplifies the construction and the interpretation of abstract operational models. There are many illustrative examples (see the ASM applications cited in [9]) how the concept of stepwise refinement can be utilized to obtain through a hierarchical organization and systematic structuring a significant reduction in the complexity of ASM models.[4]

To obtain a reliable basis for establishing correctness of mathematical models, especially whether a model is faithful to reality, the given context in which a model is to be defined is of particular interest. With respect to the embedding of a system into its physical environment, it is therefore important to clearly identify and properly state

- the underlying assumptions and constraints on the expected behaviour of the external environment;
- how external conditions and events (as associated with the environment) affect the transition behaviour of a system.

The ASM method provides various conceptual and expressive means effectively supporting an *open system view* (in contrast to a closed world assumption where everything is included in the model). To incorporate the semantic issues addressed above into the notion of ASM state and the conceptual framework of expressing state changes as local transformations on structures, the following classification scheme on ASM functions is employed.

Classification of ASM Functions Assume a given ASM $\mathcal{M}$ defined over a vocabulary Υ. Names in Υ denote abstract mathematical representations of real-world objects, their properties and the relations between these objects, as associated with a certain application domain. Depending on application domain specific knowledge about the role or status of these objects (e.g. whether an object belongs to the system or to the environment, how properties and relations may be affected by actions taking place within the system or the environment etc.) one can classify the functions of $\mathcal{M}$ according to a general scheme.

- A *static* function never changes; the name of a static function does always have the same fixed interpretation independent of the given state of $\mathcal{M}$.
- A *controlled* function can be updated as specified by the ASM program; the name of a controlled function may thus have different interpretations in different states of $\mathcal{M}$.

[4]A comprehensive treatment of the methodological background on ASM-based modeling, validation and verification of complex systems can be found in [10].

- A *monitored* function represents a *read-only* function of the ASM program; though it *must not* be updated by $\mathcal{M}$ itself, it may be altered by the external environment. Accordingly, the name of a monitored function may have different interpretations in different states of $\mathcal{M}$.

Controlled functions and monitored functions represent non-static mathematical objects and are therefore also called *dynamic* functions.

Finally, there is a more subtle class of functions in addition to the ones described above. To model interactions between a system and its environment, it is sometimes required to have functions which are *shared* between $\mathcal{M}$ and the environment, i.e. they are partly controlled and partly monitored at the same time. Those functions are called *interaction functions*. A reasonable integrity constraint for interaction functions is that no interference with respect to mutually updated locations must occur.

2.3 Multi-Agent Real-Time ASMs

Telecommunication systems are characterized by their concurrent and reactive nature; their operation increasingly depends on embedded control functions and as such they are subject to external timing constraints [4]. A suitable mathematical basis for modeling the operational semantics of SDL, including timing behaviour, is provided by the concept of multi-agent real-time ASM outlined below.

Multi-Agent ASMs A *multi-agent* ASM $\mathcal{M}$ consists of multiple autonomous *agents* cooperatively performing concurrent computations of $\mathcal{M}$. Agents communicate asynchronously through globally shared states of $\mathcal{M}$. The behaviour of an agent a is defined by its program as represented by the *module* $P(a)$. Assuming a statically defined set of modules, a unary dynamic function *Mod* assigns to each of the agents one of these modules.[5]

A special nullary function *Self* is used as a self reference (i.e. *Self* returns different values when called by different agents). Each agent a has its own partial view $View_a(S)$ on a global state S of $\mathcal{M}$ on which it fires the transition rules in $P(a)$ – see Fig. 1. The underlying semantic model ensures (by restricting admissible non-determinism in runs of $\mathcal{M}$) that the order in which the agents perform their operations is always such that no conflicts arise (for details see the definition of *partially ordered runs* in [8]).

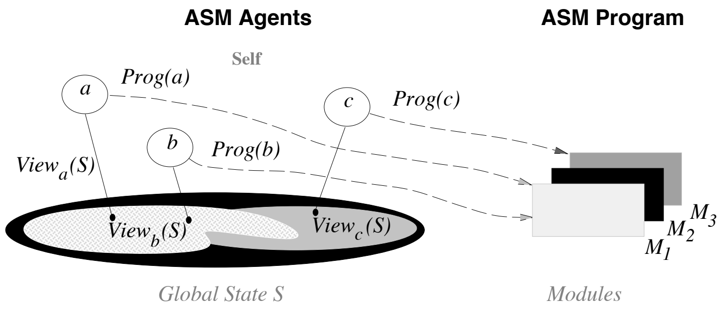

Figure 1: ASM scheme with three agents and three modules

Real-Time ASMs Multi-agent ASMs with real-time behaviour are defined in [3]. This model is adopted here with a corresponding notion of time. SDL uses the expression **now** to represent the global system time, where the possible values of **now** are given by the predefined SDL sort *Time*[6]. Accordingly, we introduce a nullary monitored function *now* taking values in a corresponding domain *TIME*

$$now: \quad TIME, \qquad TIME \subseteq \mathbb{R}.$$

By imposing additional constraints on the notion of run, namely the *discreteness* requirement defined in [3] together with simple restrictions on how functions evolve, one obtains a restricted class of multi-agent ASMs with agents performing *instantaneous* actions in *continuous* time (see also [2]). Intuitively, that means that an agent fires a rule as soon as the enabling condition expressed by the guard of the rule

[5] *Mod* is called a *dynamic* function because it may have different interpretations in different states of $\mathcal{M}$; the behaviour of newly created agents can thus be defined at run time by updating *Mod* accordingly.

[6] *Time* values are actually real numbers with restricted operations (see Appendix D to Z.100).

becomes true. Strictly speaking, one has to be more careful about the precise meaning of "immediate" (as explained in [3]). Nevertheless, we can assume here that an agent which is enabled at time t to fire a certain rule actually fires the rule not later than $t + \epsilon$ (for some infinitely small ϵ).

3 AN ABSTRACT SDL MACHINE

Our mathematical model of the operational semantics of Basic SDL is defined in terms of a multi-agent real-time ASM. We obtain an abstract interpretation model for Basic SDL at the level of basic objects (like processes, signals, channels etc.) and elementary operations (such as signal transfer operations and timer operations). To ensure that the resulting description is easily readable and understandable, our *abstract SDL machine* directly reflects the common view of SDL systems and also adopts the standard terminology of SDL.[7]

For the construction of the abstract SDL machine assume to have states of a fixed vocabulary Υ_{SDL}. The names in Υ_{SDL} denote various static/dynamic domains together with various static/dynamic functions and predicates defined on them. Functions are regarded as partial functions, whereas predicates are total. The default value of functions and predicates at locations not defined by the initial state of the abstract machine model is *undef* respectively *false*.

The behaviour of SDL systems, including both functional aspects and timing aspects, is described in terms of the behaviour of their *active* components, namely: *processes, timers* and *delaying channels.* Thus there are three ASM modules—called Process_Module, Timer_Module and Channel_Module—to be executed by a set of concurrently operating ASM agents, where one can identify a separate agent[8] with each instance of a system process, each timer instance, and each delaying channel.

Hierarchical Structure The construction and the understanding of the formal model is considerably simplified through its modular structure. In particular, the use of *macros* for defining subrules naturally enables stepwise refinements leading through a hierarchy of abstraction levels.

For brevity, we focus here on the formalization of some typical SDL features exemplifying our modeling approach and refer to [2] for further details.

3.1 ASM Representation of SDL Objects

For the purpose of illustrating how SDL objects and relations between these objects are encoded into ASM states, the formalization of process instances, signal instances and delaying channels is detailed below. To simplify matters, let *CHANNEL*, *PROCESS* and *SIGNAL* be given sets abstractly representing channels, process names and signal types as associated with an underlying SDL system specification.

Process Instances PId values of environment process instances must be distinguishable from PId values of system process instances. It is therefore assumed to have a set *PID*, the *process instance identifiers*, consisting of two disjoint subsets, $PID = PID_{sys} \cup PID_{env}$, where PID_{sys} and PID_{env} respectively identify the process instances of the system and the environment.

The relation between process instances and process types is defined through the following function

$$procname: \quad PID \rightarrow PROCESS.$$

Signal Instances Signal instances are elements of a dynamic domain *SIGINST* on which various operations are defined: *signame* yields an element of *SIGNAL*; *values* yields an optional list of signal values from a domain *VALUE*; *senderid* and *receiverid* refer to elements of *PID*; and *path* yields the path information as an element of a static domain *PATH.*

[7] One may of course argue that there are alternative levels of abstraction which might be more adequate for dealing with certain behavioural or structural aspects of SDL. Note that such questions are outside the scope of this paper; the purpose here is to convince the SDL experts of the ability of the ASM approach to directly convert their intuitive understanding of SDL into a formal semantic model which can be defined at any desired abstraction level with reasonable detail and precision.

[8] Recall that the association of ASM agents to the modules they execute is defined by the function *Mod* (see Sect. 2.3).

Additionally, an auxiliary function *receivername* defined on *SIGINST* is used to refer to receiver names; when applied to some $si \in SIGINST$, the expression $receivername(si)$ has the following meaning:

$$receivername(si) = \begin{cases} procname(receiverid(si)), & \text{if } receiverid(si) \in PID_{sys} \\ env, & \text{if } receiverid(si) \in PID_{env} \\ undef, & \text{otherwise.} \end{cases}$$

Input Buffers To each element of PID_{sys} a uniquely determined *input buffer* is assigned using a unary dynamic function *buffer*,

$$buffer: \quad PID \rightarrow SIGINST^*,$$

where $SIGINST^*$ denotes the set of all finite *sequences* of signal instances.

Delaying Channels For each path of a delaying SDL channel there is a *channel queue* holding signals which are presently *in transit* on this channel path. Channel queues are finite sequences of signal instances as defined by a unary dynamic function

$$queue: \quad CH\_PATH \rightarrow SIGINST^*,$$

where *CH_PATH* refers to the set of channel paths. The function *queue* yields the result "*undef*" when applied to a non-delaying channel.

Similarly, the elements of *CH_PATH* are associated with the delaying channels to which they belong as expressed by a unary static function *channel* from *CH_PATH* to *CHANNEL*. (Recall that delaying channels are identified with channel agents.)

Figure 2 illustrates the representation scheme of delaying channels in the abstract SDL machine.

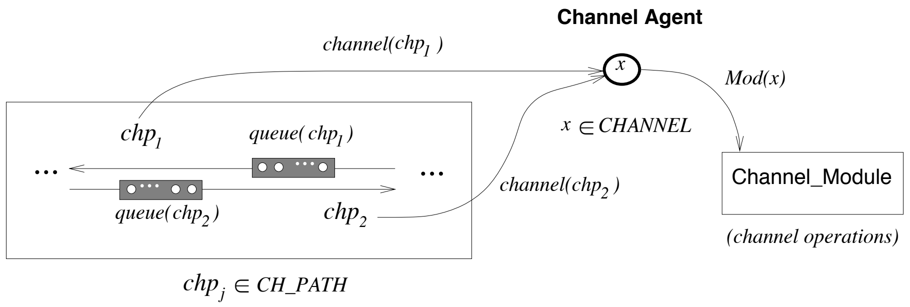

Figure 2: ASM representation of a bidirectional delaying channel

Signal Transfer The delaying behaviour of channels actually depends on the system context, e.g. on the underlying communication network; as such it is outside the scope of an SDL system model. Consequently, this behaviour is modeled in terms of an abstract predicate *InTransit*,

$$InTransit: \quad SIGINST \times CH\_PATH \rightarrow BOOL.$$

The abstract meaning of *InTransit* is as one expects: $InTransit(si, chp)$ holds for some signal instance $si \in SIGINST$ and delaying channel path $chp \in CH\_PATH$ iff

si is currently in transit on chp and has not yet reached its destination.

InTransit is a typical monitored predicate (see Sect. 2.2). Irrespective of its non-deterministic nature, there is a necessary integrity constraint on the predicate *InTransit* to ensure that signal transfer via channels is reliable: the time interval delaying the transfer of a signal is indeterminate and non-constant but *finite*.

3.2 Behaviour of Channels

The module Channel_Module consists of a single rule stating how channel agents deliver signal instances to specified receivers. In each computation step a channel agent c checks for each path chp such that $channel(chp) = c$ whether there is a signal instance si ready to be delivered to its destination.

In the rule below, the selection of channel paths is specified by means of a `do forall`-construct. That is, all chp in CH_PATH such that $channel(chp) = Self$ are examined within one single step. If the auxiliary predicate $ReadyToDeliver$ holds on chp, the update operations stated by the subrule in the body of the `do forall`-construct become effective. In other words, there are separately instantiated copies of this subrule, one for each matching chp.

Depending on the location of the signal destination one can distinguish two cases: si is either appended to the input buffer of some system process instance—as stated by DeliverToProcess(si), or it is delivered to the environment env—as stated by DeliverToEnv(si). Since the propagation of signals within the external environment is outside the scope of the model captured by the definition of SDL, DeliverToEnv(si) is consequently left abstract.

```
DeliverSignals
  ≡ do forall $chp$ : $CH\_PATH(chp)$ and $channel(chp) = Self$
      if $ReadyToDeliver(chp)$ then
         $queue(chp) := tail(queue(chp))$
         let $si = head(queue(chp))$, $r = receivername(si)$ in
            if $r = env$ then
               DeliverToEnv($si$)
            else
               DeliverToProcess($si, r$)
    where
      $ReadyToDeliver(chp)$
        $\equiv \exists si : SIGINST(si) \land si = head(queue(chp)) \land \neg InTransit(si, chp)$
```

DeliverToProcess(..) needs to distinguish whether the signal instance $SInst$ is to be delivered to an arbitrary process instance of the process instance set $PName$ (i.e, if no receiver PId is defined) or to a particular process instance as identified through its PId.

Now, it may of course happen that the specified process instance does not exist anymore when $SInst$ eventually arrives at the end of the communication path. Similarly, the nondeterministic choice does not necessarily yield a definite result[9] (since all instances of $PName$ may already have terminated their execution). It is therefore to be checked prior to delivering $SInst$ whether a valid receiver exists.

Whenever no receiver exists, Z.100 assumes (see Sect. 2.7.4 of [1]) that the signal instance is discarded[10]. In our model there is however no need to discard $SInst$ from $SIGINST$ (what we could easily do) as it will not be referred to any further.

```
DeliverToProcess($SInst, PName$)
  ≡ let $PId = receiverid(SInst)$ in
      if $PId = undef$ then
         choose $p$ : $PID(p)$ and $procname(p) = receivername(SInst)$
            $buffer(p) := buffer(p)^\frown\langle SInst\rangle$
      else
         if $PID_{sys}(PId)$ then
            $buffer(PId) := buffer(PId)^\frown\langle SInst\rangle$
```

3.3 Behaviour of Timers

Recall that timer agents of the abstract SDL machine are identified with timer instances of an underlying SDL system. Their behaviour is defined by the rules of the Timer_Module. Similar to process instances

[9]Note that a `choose`-construct does not affect the ASM state if the underlying set is empty (i.e., in that case the subrule in the body of the `choose`-construct is simply ignored – see [7]).

[10]Here arises an interesting question, which is not completely answered by the definitions of Z.100: does the fact that the environment may continue to send signals to an SDL system even when no process instances are left mean that such a system still has a behaviour?

and signal instances, timer instances are represented as elements of dynamic domain *TIMERINST*. For a given timer instance t its expiration time is obtained as the value of a unary dynamic function *expire*,

$$expire : TIMERINST \to TIME,$$

where $expire(t)$ is set to *undef* each time t expires.

The relationship between timer instances and the process instances to which they belong is expressed by a corresponding mapping *owner* from *TIMERINST* to *PID*. To associate the timer signals in *SIGINST* with the timer instances they originate from there is a mapping *timer* from *SIGINST* to *TIMERINST*.

A timer instance t is said to be *active* (in accordance with the meaning of the SDL expression **active**) if the following predicate holds on t.

$$Active(t) \equiv$$
$$ActiveTime(t) \vee ActiveSignal(t)$$
$$ActiveTime(t) \equiv$$
$$TIMERINST(t) \wedge expire(t) \neq undef$$
$$ActiveSignal(t) \equiv$$
$$TIMERINST(t) \wedge \exists\, s \in SIGINST :\; t = timer(s) \wedge s \; in \; buffer(owner(t))$$

A timer agent t watches the activities of $owner(t)$, the related process agent, and becomes involved only when $owner(t)$ encounters a **set** or **reset** instruction. In the meantime, i.e. when no **set** or **reset** instruction is to be executed, t merely checks whether it is currently active and the value of *now* is already equal or greater than $expire(t)$ in order to generate a timer signal.

In the definition of TimerOperation below, an auxiliary predicate *MyAction* triggers the operations of the timer agent. In case that *MyAction* holds on *Self*, *Action* is either *set* or *reset* (for brevity, the formalization of *MyAction* and *Action* is not given here); otherwise, the value of *now* is checked against the expiration time.

```
TimerOperation
  ≡ if $MyAction(Self)$ then
      if $Action = set$ then
        let $time = fst(Arg)$ in
          SetExpirationTime($time$)
          DiscardTimerSignal
      else
        if $Active(Self)$ then
          $expire(Self) := undef$
          DiscardTimerSignal

    else
      if $ActiveTime(Self) \wedge now \geq expire(Self)$ then
        $expire(Self) := undef$
        CreateTimerSignal
```

The value expressed by the unary function *duration* in the following rule is either "0" or a default value derived from the timer definition[11].

```
SetExpirationTime($Time$)
  ≡ if $Time = undef$ then
      $expire(Self) := now + duration(timername(Self))$
    elif $Time \leq now$ then
      $expire(Self) := undef$
      CreateTimerSignal
    else
      $expire(Self) := Time$
```

For the definition of CreateTimerSignal and DiscardTimerSignal see [2].

[11]SDL allows to set a timer without explicitly specifying the expiration time; the resulting time value is then obtained by adding a *default duration* to the current value of **now**. Furthermore, a timer may be set to a time value which is smaller or equal to the value of **now** having the effect that the timer expires immediately (see Sect. 2.8 of [1]).

4 ASM TOOL SUPPORT

For the purpose of using Abstract State Machines as a practical tool for mathematical modeling of complex systems, especially in an industrial system design context, abstract models need to be implemented on real machines. To provide supporting software tools, the *ASM Workbench*—a tool environment based on the *ASM Specification Language (ASM-SL)* [20]—has been developed at Paderborn University (see also [21]). The ASM Workbench offers various tools for machine-based analysis and simulation of executable ASM models and in particular includes:

- a type checker implementing a typed version of ASMs,
- an interpreter for simulating high-level ASM specifications,
- a graphical user interface for controlling the simulation flow.

The purpose of ASM-SL is to include in the ASM language a set of predefined types (e.g. booleans, integers, strings), generic mathematical structures (like tuples, lists, sets) and a few simple but powerful constructions (such as freely generated types, case distinction, recursion etc.) which proved to be particularly useful in software specification. Such a model based approach, while being less abstract than other techniques (like, for instance, algebraic specification), allows to model in a natural and concise way a wide range of systems, and has the advantage that executable models are obtained by construction. Moreover, the use of familiar notations coming from discrete mathematics and computer science guarantees that specifications are easy to understand and easy to translate into other languages, for the purpose of verification or implementation.

The current tool environment is mainly intended to support interactive development of formal requirement specifications in early system design phases. Additional support for efficient prototyping will soon be available through the compilation of executable ASM specifications into Java VM code. A corresponding compiler is currently being developed in cooperation with the University of Tromsø.

Provided that those details which are left abstract in the ASM model of Basic SDL (this mainly concerns interfaces to the external environment) are handled properly (e.g. by specifying them through user inputs, external processes etc.), an executable version of this model can be produced with a reasonable effort. Though there is not yet a concrete result, present investigations in this direction are promising.

5 CONCLUSIONS

Two significant observations concerning the modeling approach presented here can be summerized as follows.

- The SDL view of distributed systems with real-time constraints and the semantic modeling concept of multi-agent real-time ASM clearly coincide; essential properties of the underlying computation models—namely, the notions of concurrency, reactivity and time as well as the notion of states—are so tightly related that the common understanding of SDL can directly be converted into a formal semantic model avoiding any formalization overhead.
- Even without direct support of object-oriented features the resulting ASM model of the operational semantics of Basic SDL is particularly concise, readable and understandable; furthermore, this model can easily be extended and modified as required for an evolving technical standard. Beyond the purpose addressed here, ASM models can be utilized for defining a bridging semantics in order to combine SDL with other domain specific modeling languages.

An obvious way of extending the current model is to include the full structuring facilities of SDL. Based on our experience with VHDL [11], which offers structuring concepts similar to block partitioning in SDL, we can state that this does not cause any real problems.

References

[1] ITU-T Recommendation Z.100. *Specification and Description Language (SDL).* International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.

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

[3] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.

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

[5] O. Færgemand and A. Olsen. Introduction to SDL-92. *Computer Networks and ISDN Systems*, (26):1143–1167, 1994.

[6] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Carl Hanser Verlag / Prentice Hall International, 1991.

[7] Yuri Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan–Ann Arbor, 1997.

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

[9] E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64, February 1998.

[10] Egon Börger. Why use evolving algebras for hardware and software engineering? In *Proc. of SOFSEM'95*, volume 1012 of *LNCS*, pages 236–271. Springer-Verlag, 1995.

[11] E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by EA–machines. In C. Delgado Kloos and P.T. Breuer, editors, *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science.* Kluwer Academic Publishers, 1995.

[12] The Institute of Electrical and Electronics Engineers, New York, NY, USA. *IEEE Standard VHDL Language Reference Manual-IEEE Std 1076-1993*, 1994. Order Code SH16840.

[13] J. A. Bergstra and C. A. Middleburg. Process Algebra Semantics of φSDL. Technical Report UNU/IIST Report No. 68, UNU/IIST, The United Nations University, April 1996.

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

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

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

[17] Markus Rinderspacher. A Verification concept for SDL systems and its application to the Abracadabra Protocol. Interner Bericht 14/94, Institut fr Logik, Komplexitt und Deduktionssysteme, Universitt Karlsruhe, 1994.

[18] J. Fischer, St. Lau, and A. Prinz. A Short Note About BSDL - Semantic Issues for SDL. *SDL Newsletter*, (18), January 1995.

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

[20] Giuseppe Del Castillo. ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial. Technical report, Department of Mathematics and Computer Science, Paderborn University. Technical Report (to appear).

[21] G. Del Castillo and U. Glässer. Machine-Supported Execution and Validation of High-Level Abstract State Machine Models. In *Preliminary Proc. of TOOLS'98 – Int'l Workshop on Tool Support for System Specification, Development and Verification (Malente, Germany, June 1-4, 1998)*, 1998. (to appear).