Journal of Universal Computer Science, vol. 3, no. 12 (1997), 1382-1414
submitted: 25/6/97, accepted: 26/11/97, appeared: 28/12/97 © Springer Pub. Co.

Abstract State Machine Semantics of SDL

Uwe Glässer
(Heinz Nixdorf Institut
Universität-GH Paderborn, Germany
glaesser@uni-paderborn.de)

René Karges
(metronet GmbH Köln, Germany
Rene.Karges@metronet.de)

Abstract: Based on the *ITU-T Recommendation Z.100* [27]—also known as SDL-92—we define a formal semantic model of the dynamic properties of Basic SDL in terms of an *abstract SDL machine*. More precisely, we use the concept of *multi-agent real-time ASM* [17] as a semantic platform on top of which we construct our mathematical description. The resulting interpretation model is not only mathematically precise but also reflects the common understanding of SDL 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 issue for an evolving technical standard.
In this article, we consider all relevant aspects concerning the behavior of channels, processes and timers with respect to signal transfer operations and timer operations. The model we obtain is intended as a basis for formal documentation as well as for executable high-level SDL specifications.

Key Words: SDL, Basic SDL, Semantic Foundations, Telecommunication Systems, Formal Documentation, Executable Specifications, Abstract State Machines
Category: C.2.1, C.2.4, D.2.1, D.3.1, D.3.3, F.3.2

1 Introduction

Abstract State Machines (ASMs)—formerly called *Evolving Algebras* [18, 19]—combine declarative concepts of *first-order logic* with the operational view of *transition systems* in a common framework for mathematical modeling of discrete dynamic systems. The semantic definition of the underlying machine model constitutes a simple yet powerful formal basis to deal with *concurrent* and *reactive* behavior in a direct and intuitive way; the fact that ASM-based system models naturally enable operational interpretations is often considered as an advantage when dealing with complex technical systems (e.g., as in [1, 5, 4])[1]

We present here a formal semantic model of Basic SDL as defined in (Chap. 2 of) the *ITU-T Recommendation Z.100* [27]—also referred to as SDL-92. More precisely, we focus on the dynamic semantics of Basic SDL, which we describe in terms of an abstract interpretation model based on the concept of *multi-agent real-time ASM* (see Sect. 2). The resulting description is intended to be a first step towards a formal documentation of SDL which is not only *mathematically precise* but does also reflect the common understanding of SDL (e.g., as presented in the

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

literature [25, 13, 2]) in a direct and intuitive manner. Such a documentation is of course meant to complement the natural language description of Z.100 rather than to substitute it; aiming at a a coherent (and consistent) view, the goal is to combine both descriptions by embedding the former into the latter.

Our formal definition provides a *concise* yet *understandable* model, which can easily be *extended* and *modified*—this flexibility is particularly important for an evolving technical standard. In that respect the work presented here is comparable to our semantic model of the hardware description language VHDL [6, 5]. Furthermore, the model we obtain provides an excellent basis for mechanizing SDL specifications, e.g. as required for machine supported analysis and transformation or the execution and animation of high-level SDL specifications.

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 [27]). However, the current situation is not really satisfying:

> *"This annex constitutes a formal definition of SDL. If any properties of an SDL concept defined in this document, contradicts the properties defined in Z.100 and the concept is consistently defined in Z.100, then the definition in Z.100 takes precedence and this formal definition requires correction."* (Annex F1 to [27], page 1)

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 [3], Bergstra and Middleburg define a *process algebra* semantics of a restricted version of SDL, which they call φSDL. Broy [10], Holz and Stølen [21] 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 [14]. Rinderspacher employs a *term-rewriting system* based modeling concept [26]. Some of these approaches consider only a relatively small subset of SDL ignoring certain essential features (e.g., dynamic process creation or basic structuring concepts). An approach aiming at a more comprehensive semantic model of SDL is provided by Fischer, Lau and Prinz through their definition of BSDL (*Base SDL*) using *Object-Z* [15, 24].

This work is organized as follows. *Section 2* briefly introduces the basic concepts and notions of Abstract State Machines, including the employed model of real time, as far as these are required here. *Section 3* defines the ASM representation of SDL systems and the overall organization of our abstract interpretation model in terms of an *abstract SDL machine*. *Section 4* addresses the behavior of channels. The behavior of processes (especially, the effect of signal transfer and timer operations) is represented in *Sect. 5*, while *Sect. 6* models the behavior of timers. Finally, *Sect. 7* contains some brief conclusions.

2 Abstract State Machines

We address here (for the convenience of the reader) some essential aspects of the ASM concepts that are relevant in the sequel of this article in an informal style. The particular focus is on the specific ASM model that will be employed for the construction of our SDL model. For a rigorous mathematical definition of the semantic foundations of ASMs, we however refer to [19] and [20]. A comprehensive treatment of the methodological background on ASM-based modeling, validation and verification of complex systems can be found in [9].

2.1 The Basic ASM Model

An ASM $\mathcal{A}$ is defined by its *program Prog* consisting of a finite number of *transition rules* and its *initial state* S_0. For the definition of S_0 we assume a given *vocabulary* Υ, where Υ is a finite collection of function names and predicate names, each of a fixed arity. To indicate that a name has a fixed interpretation, names in Υ may be marked as *static* (whereas non-static names may have different interpretations depending on the state of $\mathcal{A}$). Υ contains an a priori given set of predefined static names including the equality sign, the nullary predicate names *True*, *False*, the nullary function name *undef*, the universe $BOOL$, and the standard Boolean operation names.

States. Mathematically speaking, states of $\mathcal{A}$ are first-order *structures*. They define interpretations of the names in Υ over a nonempty set X, called the *base set* of $\mathcal{A}$. Structures, as considered here, do not contain relations but express relations through characteristic functions (to which we refer as predicates).

Formally, all functions are total functions on X. Though, it is possible to imitate *partial* functions by marking "undefined" values with the designated element *undef*, except for predicates. By definition, the only possible values of predicates are *True*, *False*. Unary predicates have a special role: they are used to form variants of *many-sorted* structures on top of ordinary structures[2].

Updates. Non-static functions are subject to update operations as follows. A *location* of a state S of $\mathcal{A}$ is a pair $loc = (f, \overline{x})$, where f is a non-static name in Υ and $\overline{x}$ denotes a sequence of elements of X according to the arity of f. An *update* of S is a pair $\delta = (loc, y)$, where $y \in X$ is the new value to be associated with the location *loc* of S.

To *fire* δ at S means to transform S into a state S' of $\mathcal{A}$ such that[3] $f_{S'}(\overline{x}) = y$, and all other locations loc' of S, $loc' \neq loc$, remain unaffected. An *update set* Δ over S is a set of updates of S. Δ is *consistent*, iff it does not contain any two updates δ, δ' such that $\delta = (loc, y)$ and $\delta' = (loc, y')$ and $y \neq y'$. To *fire* a consistent update set Δ at a state S means to fire all its members simultaneously at S, i.e. to produce a new state S' such that

$$f_{S'}(\overline{x}) = \begin{cases} y, & \textit{if } ((f,\overline{x}), y) \in \Delta \\ f_S(\overline{x}), & \textit{otherwise.} \end{cases}$$

[2] For instance, the universe $BOOL$ denotes the set of all elements x within the base set X such that $BOOL(x)$ holds (these are of course only the elements denoted by *True*, *False*).

[3] The notation $f_{S'}$ is used to denote the interpretation of the function f in state S'.

To fire an inconsistent update set means to do nothing (i.e., to produce a state S' such that $S' = S$). In this way, computations are modeled through (finite or infinite) *runs* of $\mathcal{A}$ as sequences of ASM states $S_0\, S_1\, S_2 \ldots$, such that S_{i+1} is obtained from S_i by firing Δ_{S_i} on S_i $(i \geq 0)$.

Instructions. Updates are specified by the transition rules of *Prog* in terms 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. For brevity, we explain the meaning of ASM rules as employed in the abstract machine model of SDL (see Sect. 3) only informally and refer to [19] resp. [20] for a rigorous semantic definition. Nevertheless, it should easily be possible to get a sufficiently detailed understanding without consulting the formal semantic definition.

2.2 Multi-Agent Real-Time ASMs

In the sequel, we concentrate on *concurrent* and *reactive* systems that are embedded into some given physical environment to which we refer as the *(external) environment*. While performing permanent interactions with the environment, the operations of these systems are subject to external timing constraints (see Sect. 2.2.2). The mathematical model we use to describe the behavior of the system class considered here is based on *multi-agent real-time ASMs* as detailed below.

A particular important modeling aspect is the dependency of the system behavior on externally controlled conditions and events. To clearly identify the embedding of a system model into a given environment, it is expedient to classify ASM functions depending on whether and how they may (or may not) change during a run.[4]

- A *static* function never changes; the name of a static function has the same fixed interpretation independent of a particular ASM state.
- A *controlled* function can be updated as specified by the ASM program; the name of a controlled function may have different interpretations in different ASM states.
- A *monitored* function represents a *read-only* function of the ASM program; though it must not be updated by the ASM itself, it may be altered by the external environment. Accordingly, the name of a monitored function can have different interpretations in different ASM states.

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

[4] We use here the terminology introduced in [7], which is essentially based on the classification scheme defined in [9] (though the naming is different).

is sometimes required to have functions which are partly controlled and partly monitored at the same time (as will be exemplified in Sect. 3.1.3). These are called *interaction functions*. A reasonable integrity constraint for interface functions is that no interference with respect to mutually updated locations must occur.

2.2.1 Multi-Agent ASMs

A *multi-agent*[5] ASM $\mathcal{A}$ consists of multiple autonomous *agents* cooperatively performing concurrent computations of $\mathcal{A}$. Agents communicate *asynchronously* through globally shared ASM states. The behavior of an agent a is defined through $Prog(a)$, the program *module* associated with a. Assuming a statically defined set of modules, a unary dynamic function *Mod* assigns to each of the agents one of these modules.[6]

A special nullary function *Self* is used by the agents as a *self reference* (*Self* returns different values when called by different agents). Each agent a has its own partial view $View_a(S)$ on a given global state S of $\mathcal{A}$ on which it fires the rules in $Prog(a)$ – see Fig. 1. The underlying semantic model ensures (by restricting the class of admissible runs of $\mathcal{A}$) 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 [19]).

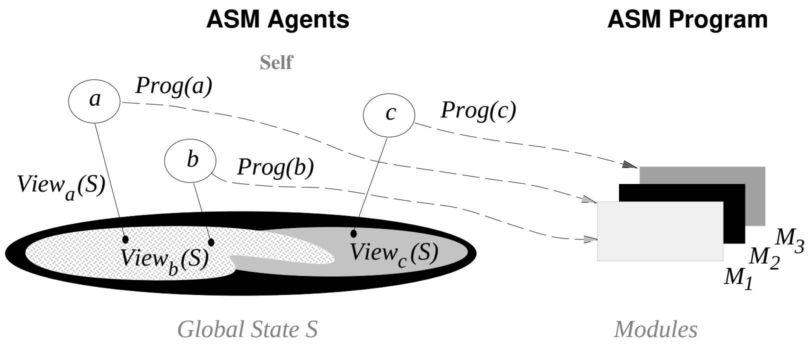

Figure 1: Multi-Agent ASM scheme with three agents and three modules

---

[5] Formally, the meaning of the notion of *multi-agent ASM*, as it is used here, is identical with the meaning of *distributed ASM* as defined in [19] (where the term *distributed* actually refers to the distribution of control rather than to the distribution of data).

[6] Note that *Mod* allows to define (or to redefine) the behavior of agents dynamically; it is thereby possible to create new agents at run time. In a given state S of $\mathcal{A}$ the agents of $\mathcal{A}$ are those elements a from the underlying base set such that $Mod_S(a) \neq undef$.

2.2.2 Real-Time ASMs

Telecommunication systems are in principle real-time systems as they must respond within certain time limits[7]. To model the dynamic properties of SDL, including timing behavior, we apply the concept of real-time ASMs as defined by Gurevich and Huggins in [17]. Real-time ASMs impose additional constraints on the notion of run and thereby provide a restricted class of ASMs with agents performing *instantaneous* actions in *continuous* time. For the purpose considered here, it is important that agents fire their rules at the moment they are enabled. Additionally, we assume that changes in the environment take also place instantaneously. We outline here only the basic idea and refer to the original definition for further details.

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*[8]. We therefore introduce a nullary monitored function *now* taking values in a corresponding domain *TIME*

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

For the construction of our model assume a given vocabulary Υ containing *TIME* (but not *now*) and let Υ^+ be the extension of Υ with the function symbol *now*. Restrict attention to Υ^+-states where *TIME* is identified with $\mathbb{R}$ and *now* evaluates to a real number. One can then define a run R of the resulting machine model as a mapping from the interval $[0, \infty)$ to states of vocabulary Υ^+ satisfying the following *discreteness* requirements, where $\rho(t)$ is the reduct[9] of $R(t)$ to Υ:

1. for every $t \geq 0$, *now* evaluates to t at state $R(t)$;
2. for every $\tau > 0$, there is a *finite sequence* $0 = t_0 < t_1 < \ldots < t_n = \tau$ such that if $t_i < \alpha < \beta < t_{i+1}$ then $\rho(\alpha) = \rho(\beta)$.

Based on this notion of run, one can then define a computation model in which agents fire their rules immediately. Intuitively, that means that an agent fires a rule as soon as the enabling condition expressed by the guard of the rule becomes true. Strictly speaking, one has to be more careful about the precise meaning of 'immediate' (as explained in [17]). 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). By adding further constraints on how basic functions evolve, the model in [17] is such that internal updates (as performed by the agents) and external updates (as performed by the environment) do not interfere.

---

[7] According to [25], time supervision in telecommunication systems is typically used for purposes such as the following: to control the release of limited resources, to control answers from unreliable resources, or to issue actions on a regular basis; whether the time constructs of SDL are sufficient for specific real-time requirements depends very much on the particular application.

[8] *Time* values are actually real numbers with restricted operations (see Appendix D to Z.100); SDL does in particular not define a notion for scaling time (it is merely assumed that the same scale of time is used throughout an entire system description).

[9] That is, for a given value t, we obtain $\rho(t)$ from $R(t)$ by ignoring the interpretation of the function name *now*.

2.3 Notational Conventions

The construction and the understanding of the 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.

To further increase the readability of our formal model, we use the following notational conventions. Macro names are written in Small Caps (e.g., DELIVERTOPROCESS(..)), domain names in capitalized italics (e.g., *BOOL*), function names in small italics, and predicate names are written in italics in the form *CondName*. The names of ASM modules are written in Sans Serif (e.g., Timer_Module).

Closing constructs of rules (like the `endif` of an `if` construct) are omitted if the meaning is clear from the context. For layout reasons we write `elif` instead of `elseif`. Finally, SDL keywords are written in bold font.

3 An Abstract SDL Machine

We define our mathematical model of the dynamic properties of SDL in terms of an *abstract SDL machine* using the concept of multi-agent real-time ASM as a formal basis. To ensure that the resulting description is easily readable and understandable, our model reflects the common view on SDL systems and also adopts the standard terminology of SDL.

For the construction of the abstract machine model, we 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 – see Sect. 2.1), where we assume the default value of those locations which are not explicitly defined to be *undef* resp. *False*.

Our abstract machine model describes the functional behavior and the timing behavior of an SDL system in terms of the behavior of its *active* components, namely: *processes, timers* and *channels*. Accordingly, we have three basic ASM modules—called Process_Module, Timer_Module and Channel_Module—to be executed by a set of concurrently operating ASM agents, where we identify a separate agent[10] with each instance of a *system process*, each *timer instance*, and each *delaying channel*. We start by defining our ASM representation of the main SDL objects out of which an SDL system is composed.

3.1 ASM Representation of SDL Objects

An SDL system consists of a fixed number of statically interconnected *blocks*. Blocks are connected to each other and to the system environment by means of *channels*. The concept of *block partitioning* allows to recursively define the system structure: blocks may further be decomposed into subblocks connected (to each other and/or to the enclosing block) through channels; at the bottom level, a block contains a flat collection of processes connected (to each other and/or

[10] Recall that the association of ASM agents to the modules they execute is formally defined by the function *Mod* (see Sect. 2.1).

to the enclosing block) through *signal routes*.[11] In Basic SDL the structuring concept is however restricted to a two-level hierarchy: a system model consists of blocks containing processes but no further blocks. Figure 2 shows an example.

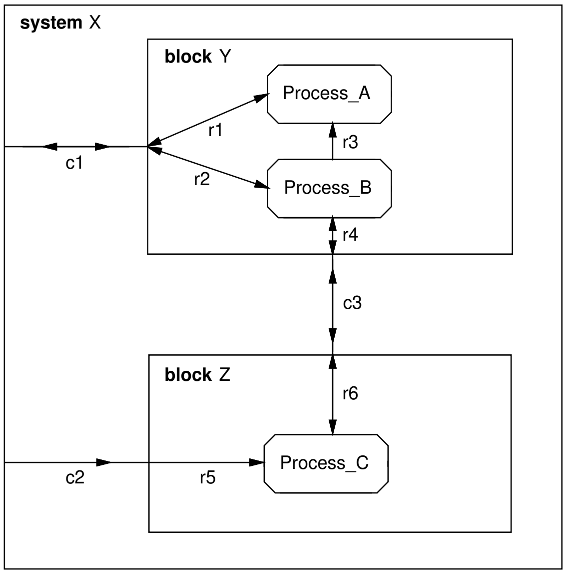

Figure 2: Structural organization of an SDL system[12]

SDL defines a distributed computation model based on concurrently operating and asynchronously communicating processes. Processes interact with each other as well as with the common environment by sending and receiving signals via channels and signal routes. Each process has its own *input buffer* keeping signals until the process is ready to actively receive them.

Exchange of signals is in fact the only form of interaction between a system and its environment. Even though the behavior of the environment is basically unpredictable, it is expected that the environment behaves in an SDL-like fashion through a corresponding set of *environment processes*. In particular, one can assume that the environment knows the system specification and acts in compliance with the constraints and requirements given by it.

Types, Instances and Sorts. A fundamental concept of SDL is the distinction between object *types* and object *instances*. Accordingly, a *system instance* is organized as a collection of named *instance sets* (e.g., *block instance sets*, *process instance sets* and *Signal instance sets*), each of which represents the instances of

[11] Note that SDL does in particular not allow to have blocks containing both subblocks as well as processes.

[12] We apply here the graphical representation format of SDL.

a particular type; i.e., instance names (e.g., process names and signal names) refer to the type. If the context is clear, we sometimes use a less strict terminology avoiding to permanently mention the distinction between type and instances; in particular, we often use 'process' as a synonym for 'process type' and 'signal' as a synonym for 'signal instance'.

In the abstract machine model the various instance sets of an SDL system are given through a number of static domains—e.g., *BLOCK*, *CHANNEL*, *SIGROUTE*, *PROCESS* and *SIGNAL*—such that each element of these domains denotes a particular instance set. The instances itself are represented as follows.

For blocks, channels and signal routes the situation is particularly simple as there is a one-to-one correspondence between types and instances so that the elements of *BLOCK*, *CHANNEL* and *SIGROUTE* can directly be identified with their respective instances. Process instances and signal instances need however to be represented in a different way, as will be explained in Sect. 3.1.1.

For the representation of (predefined and user-defined) SDL sorts we introduce another static domain *SORT* (i.e., the elements of *SORT* are identified with the sort names). In our model we need however not to distinguish the sorts of individual values and therefore assume that all values are uniformly represented as elements of a domain *VALUE*. Additionally, we assume to have some abstract representation for expressions in terms of a domain *EXPRESSION* in combination with a corresponding evaluation function:

$$eval:\ EXPRESSION \rightarrow VALUE.$$

3.1.1 Processes and Signals

Process instances are uniquely identified through *process instance identifiers (PIds)* as represented by a dynamic domain *PID*. For special purposes (e.g., such as initialization) *PID* contains a designated element *null*, which does not refer to a valid process instance. In order to distinguish between system process instances and process instances within the system environment[13], we assume that *PID* consists of two disjoint subsets, $PID = PID_{sys} \cup PID_{env}$, where PID_{sys} and PID_{env} respectively denote the instances of system processes and those of environment processes.

SDL provides four built-in operations on process identifiers: **self**, **sender**, **offspring** and **parent** (see Sect. 2.4.4 of [27]). We define analogous functions on *PID*, namely *Self*, *sender*, *offspring* and *parent*, each of which yields a corresponding PId value. For a given process instance $p \in PID$ the meaning is as follows[14]: *Self* is equal to p; $sender(p)$ identifies the process instance from which p has most recently received a signal (if defined); $offspring(p)$ identifies the process instance most recently created by p (if any); $parent(p)$ yields the process instance that has created p.[15]

[13] The PId values of environment process instances must be distinguishable from any of the PId values within the system (see Sect. 1.3.2 of [27]).

[14] The default meaning of the function *Self* in the multi-agent ASM model (see Sect. 2.2.1) actually coincides with the meaning of the SDL function **self**.

[15] For all process instances present at system initialization, the predefined value of *parent* is *null*, and for all newly created process instances, the predefined value of *sender* and *offspring* is *null* (see Sect. 2.4.4 of [27]).

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

$procname : PID \rightarrow PROCESS.$

The use of formal parameters in the definition of processes allows to instantiate certain variables when creating process instances. Assuming a corresponding domain $VARIABLE$, the association of processes with their formal parameters is expressed by the following function

$fpar : PROCESS \rightarrow VARIABLE^*.$

Process Instance Sets. Process instance sets may grow and shrink dynamically during the execution of an SDL system. For each process type the *initial* number and the *maximum* number of process instances of that type can be specified by means of two unary static functions

$initial\_instances, maximum\_instances : PROCESS \rightarrow \mathbb{N}.$ [16]

Representation of Signal Instances. Signal instances form the elementary units of communication. Our model represents signal instances as elements of a dynamic domain $SIGINST$. To access the relevant information associated with signal instances, we introduce a number of operations on $SIGINST$, as follows: *signame* yields the signal type as an element of $SIGNAL$; *values* yields an optional list of signal values from the domain $VALUE^*$; *senderid* and *receiverid* refer to elements of PID; and *path* yields a value from a domain $PATH$ (as will be explained in Sect. 3.2).

In addition, we define *receivername* on $SIGINST$, where *env* denotes the external system environment (see Sect. 3.1.3). When applied to some $si \in SIGINST$, the expression $receivername(si)$ has the following meaning:

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

Representation of Input Buffers. With each element of PID_{sys} we associate a uniquely determined *input buffer* (also called *input queue* in Z.100) as expressed by means of a dynamic function

$buffer : PID \rightarrow SIGINST^*,$

where the elements of $SIGINST^*$ denote (possibly empty) sequences of signal instances on which we apply standard operations *head*, *tail* with their usual meaning. *empty_buffer* denotes the empty input buffer.

[16] If for some given $p \in PROCESS$ the definition of p leaves the initial number of instances unspecified, the default value of $initial\_instances(p)$ is one, i.e. in the initial state p contains exactly one process instance; an unspecified maximum value means that p may contain arbitrary many process instances. (If $initial\_instances(p)$ and $maximum\_instances(p)$ are both defined, it is of course required that $initial\_instances(p) \leq maximum\_instances(p)$.)

3.1.2 Channels and Signal Routes

Channels and signal routes form *unidirectional* or *bidirectional* connections for the transportation of signals. Accordingly, they consist of one or two directed paths, called *channel paths* respectively *signal route paths*. A channel may *delay* the transmission of a signal for an indeterminate and non-constant time interval. In spite of this behavior, signals are always conveyed in a FIFO order.

In SDL-92, it is possible to mark a channel as *non-delaying* (as indicated by the SDL keyword **nodelay**), which means that the channel does not introduce any delay in conveying signals (while the default behavior is delaying). Signal routes do however always behave like non-delaying channels.

An SDL channel specification has basically the following form

$$
\begin{array}{l}
\textbf{\textit{channel}}\ \textit{channelname} \\
\quad [\textbf{\textit{nodelay}}] \\
\quad \textbf{\textit{from}}\ \{\textit{blockname} | \textbf{\textit{env}}\}\ \textbf{\textit{to}}\ \{\textit{blockname} | \textbf{\textit{env}}\} \\
\quad \textbf{\textit{with}}\ \textit{signalname}\{,\ \textit{signalname}\}^*; \\
\quad [\textbf{\textit{from}}\ \{\textit{blockname} | \textbf{\textit{env}}\}\ \textbf{\textit{to}}\ \{\textit{blockname} | \textbf{\textit{env}}\} \\
\quad \textbf{\textit{with}}\ \textit{signalname}\{,\ \textit{signalname}\}^*;] \\
\textbf{\textit{endchannel}};
\end{array}
$$

where the **from/to**-parts denote directed connections between a block and another block or the environment *env*. The **with**-part identifies the signal types that the channel is able to convey; i.e., a channel can only be used by signal instances of the declared types. Signal routes are analogously specified as directed connections between processes within the same block or between a process and its block environment.

$$
\begin{array}{l}
\textbf{signalroute}\ \textit{signalroutename} \\
\quad \textbf{from}\ \{\textit{processname} | \textbf{env}\}\ \textbf{to}\ \{\textit{processname} | \textbf{env}\} \\
\quad \textbf{with}\ \textit{signalname}\{,\ \textit{signalname}\}^*; \\
\quad [\textbf{from}\ \{\textit{processname} | \textbf{env}\}\ \textbf{to}\ \{\textit{processname} | \textbf{env}\} \\
\quad \textbf{with}\ \textit{signalname}\{,\ \textit{signalname}\}^*;]
\end{array}
$$

To model channels and signal routes according to the view of SDL, we introduce static domains *CH_PATH* and *SR_PATH* in combination with static unary functions *from*, *to* and *with* defined on $CH\_PATH \cup SR\_PATH$. *from*, *to* yield elements from $BLOCK \cup PROCESS \cup \{env\}$, while *with* yields subsets of $SIGNAL$. To associate the elements of *CH_PATH* and *SR_PATH*—i.e., the underlying directed connections of channels and signal routes—to the corresponding elements of *CHANNEL* and *SIGROUTE*, we use two unary static functions:

$$channel : CH\_PATH \rightarrow CHANNEL, \qquad sigroute : SR\_PATH \rightarrow SIGROUTE.$$

SDL reasonably restricts the definition of channels and signal routes through the following constraints:

1. $\forall x, y \in CH\_PATH :$
   $x \neq y \wedge channel(x) = channel(y) \;\Rightarrow\; from(x) = to(y) \wedge from(y) = to(x)$
2. $\forall x, y \in SR\_PATH :$
   $x \neq y \wedge sigroute(x) = sigroute(y) \;\Rightarrow\; from(x) = to(y) \wedge from(y) = to(x)$
3. $\forall x \in CH\_PATH \cup SR\_PATH : from(x) \neq to(x)$

According to the definition of the abstract SDL machine, as given so far, a delaying channel is identified with a channel agent, i.e. with an element $ch \in CHANNEL$ such that the following predicate holds

$$Delaying(ch) \equiv \; Mod(ch) = \mathsf{Channel\_Module}$$

(whereas a non-delaying channel ch has no behavior, i.e. $Mod(ch) = undef$).

Representation of Channel Queues. With each direction of a delaying channel SDL associates a *channel queue*[17] holding signals which are presently *in transit* on the channel. We model channel queues through a dynamic function *queue* as finite sequences of signal instances.

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

Figure 3 illustrates the representation of delaying channels in the abstract SDL machine.

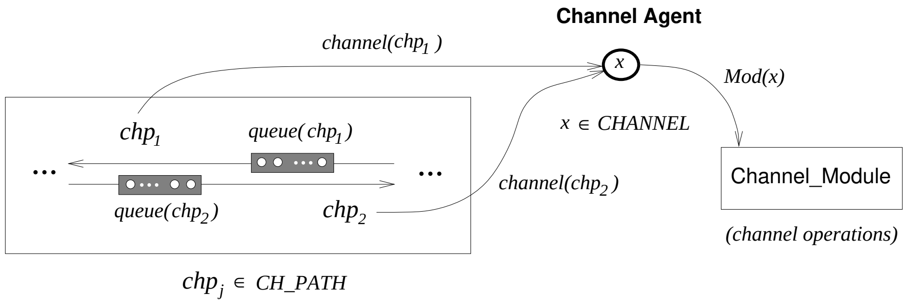

Figure 3: ASM representation of a bidirectional delaying channel

3.1.3 The External Environment

Consider the dependency of an SDL system from its external environment, especially, the fact that the functional behavior and the timing behavior of the system is manipulated in various ways by external conditions and events. Typical examples are the signal transfer on delaying channels or the representation of the global system time. Even though such external dependencies are actually outside the scope of an SDL system description one can often make reasonable assumptions about the behavior of the environment. For instance, one can assume that time values increase monotonically. Concerning the behavior of channels, the following constraints are assumed (see Sect. 2.5.1 of [27]): (i) signal transfer via channels is *reliable*; (ii) the propagation of signals with respect to a given

[17] There are no a priori given restrictions on the length (or size) of channel queues.

channel is *order-preserving*; (iii) the time interval delaying the transfer of a signal is indeterminate and non-constant but *finite*.

To cope with that kind of interface problems, the mathematical concept of *externally alterable* functions, viz. monitored functions and interaction functions (see Sect. 2.2), offers a convenient abstraction mechanism; i.e., an instrument to model the system behavior at a level of detail and precision that reflects our intuitive understanding as close as possible. Even with a vague or uncertain understanding of the external environment (i.e., without having complete definitions) one can often restrict the admissible class of functions through *integrity constraints* stating assumptions on the expected behavior, as will be illustrated below.

System Time and Signal Delays. Recall that the monitored function *now* (see Sect. 2.2.2) represents the global system time. To model the delaying behavior of channels, we introduce a monitored predicate *InTransit*:

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

The meaning of *InTransit* is as one expects: $InTransit(s, chp)$ holds for some signal instance $s \in SIGINST$ and delaying channel path $chp \in CH\_PATH$ iff the following two conditions do so:

(1) s is contained in $queue(chp)$,
(2) the transmission of s on chp has not yet been completed.[18]

With respect to the SDL requirements on signal transfer via delaying channels, as stated above, (i), (ii) are formalized by the representation of channel queues (and the operations on channel queues). Requirement (iii), however, is a necessary integrity constraint on the predicate *InTransit*.

Signals from the Environment. In order to receive signals from the environment[19], we allow that the respective channel queues and input buffers (as detailed below) may be updated by the environment, as well. Accordingly, the domain *SIGINST* together with the operations on signal instances (see Sect. 3.1.1) and the functions *buffer*, *queue* are internally as well as externally alterable under the integrity constraint that the system and the environment do not interfere. More precisely, the functions *buffer* and *queue* are externally alterable only at the locations identified by the two sets Pid_{ext} and Chp_{ext} (where Pid_{ext} will be defined in Sect. 3.2.1):

$$Chp_{ext} = \{p \mid CH\_PATH(p) \wedge Delaying(channel(p)) \wedge \ from(p) = env\}.$$

Environment Processes. The creation and termination of process instances in the environment affects the domain *PID* (through PID_{env}) as well as the operations on *PID* (see Sect. 3.1.1). That means, *PID* and the operations on *PID* have to be considered as interaction functions. Concurrent updates of *PID*, as performed by the system and the environment, cannot cause any conflicts since PID_{env} and PID_{sys} resp. refer to disjoint sets.

[18] The second condition typically depends on external behavior (e.g., of the communication network); as such it is controlled by the environment and can only be observed within an SDL system.

[19] The transfer of signals in the opposite direction, i.e. from the system to the environment, will be defined in Sect. 5.2.

Table 1 gives an overview of the domains, functions and predicates linking the abstract SDL machine model to the external world.

Identifier	Function Type	Intended Meaning
	Domains	
PID_{env}	monitored	Environment process instances
SIGINST	interaction	Signal instances
	Functions	
now	monitored	Global system time
offspring	interaction	
...	...	Operations on *PID* (see Sect. 3.1.1)
sender	interaction	
signame	interaction	
...	...	Operations on *SIGINST* (see Sect. 3.1.1)
path	interaction	
buffer	interaction	input buffers
queue	interaction	channel queues
	Predicates	
InTransit	monitored	Delaying behavior of channels
Spontaneous	monitored	Transition behavior (see Sect. 5.3)

Table 1: Interfaces between the model into the external environment

3.1.4 The Initial State

A complete definition of the ASM interpretation model for a given SDL system is obtained by encoding the SDL system representation[20] into the internal ASM representation scheme. In other words, the mathematical objects forming the initial state of the machine model (i.e., the domains, functions and predicates defined on the underlying base set—except for the predefined components) have to be extracted from the SDL description by means of a preprocessing step. Since that is primarily a question of mechanization (using standard compiler techniques and tools) and concerns the static semantics of SDL, we avoid the details here. Instead, we introduce the employed *encoding scheme* abstractly by means of a mapping '$[\![\ ]\!]$'. The basic idea is exemplified below (for a comprehensive treatment see also [22]).

As a primitive example for the representation of syntactical objects consider the encoding of an SDL process definition. Assume that a process type named *Server* is to be defined. The initial number and the maximum number of process instances of type *Server* is 2 resp. 16. Figure 4 sketches the resulting structure

[20] We restrict here on textual representations of SDL systems which are syntactically correct and consistent with the static and dynamic semantics as defined in the Z.100 Recommendation for Basic SDL. Additionally, we assume that shorthand notations, e.g. as offered by the so-called *additional concepts* of Basic SDL (like *RPC* or *continuous signal* – see Sect. 4 of [27]), have been eliminated by transforming them into equivalent primitive concepts.

(where NAT denotes the domain of natural numbers). For brevity, the main part of the definition, as represented by the body of the process *Server*, is left out. As will be explained in Sect. 5.1, the control flow and the SDL statements is to be encoded into a set of *flowcharts* associated with $[\![Server]\!]$.

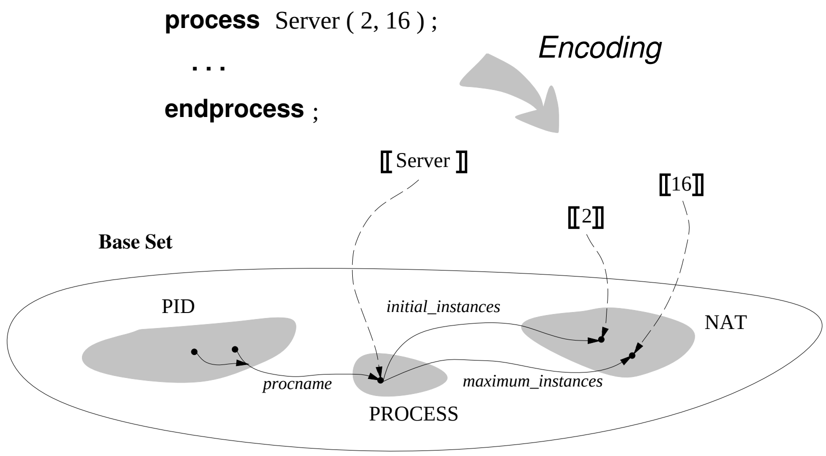

Figure 4: Encoding of the process *Server*

3.2 Global Communication via Channels and Signal Routes

The static interconnection structure of an SDL system is defined through a finite set of *communication paths*. A communication path consists of a concatenation of channel paths and signal route paths and forms a directed global connection for the transportation of signals; as such it is part of the information to be associated with an SDL signal. To each signal instance a uniquely identified communication path leading from the sender to the destination is assigned at the moment the signal instance is created.

In order to simplify the modeling of signal transfer operations (as presented in Sect. 5), we address here some essential properties of the underlying communication mechanism. Section 3.2.1 defines the abstract machine representation of communication paths. The representation of a given interconnection structure is then explained in Sect. 3.2.2 in terms of the resulting reachability constraints.

3.2.1 Representation of Communication Paths

Communication paths are specified by means of **connect**-statements. A **connect**-statement attaches a channel to one or more signal routes of a given scope unit, where the scope unit for a signal route is defined to be the enclosing block. The resulting topological information can directly be represented by means of a binary static predicate *Connected*. Additionally, we use a unary static function *scope* to associate the elements of *SIGROUTE* with those of *BLOCK* according to the scope definitions of an SDL description. For given elements $ch \in CHANNEL$, $sr \in SIGROUTE$ the possible values of *Connected* are restricted by the following constraint:

$$\begin{aligned}
&Connected(ch, sr) \Leftrightarrow \\
&\quad \exists\ block \in BLOCK : scope(sr) = block \\
&\quad \wedge\ \exists\ srp \in SR_PATH : sigroute(srp) = sr \\
&\quad \wedge\ \exists\ chp \in CH_PATH : channel(chp) = ch \\
&\quad \wedge\ ((to(srp) = env \wedge from(chp) = block) \\
&\qquad \vee (to(chp) = block \wedge from(srp) = env)).
\end{aligned}$$

To represent the communication paths that can be derived from a given SDL description (by means of simple static analysis) in a uniform way, we introduce a static domain *PATH* in combination with a number of path constructors. As far as Basic SDL is concerned, the situation is particularly simple because of the restriction to a two-level hierarchy (see Sect. 3.1). We can thereby restrict on four basic path constructors to define the elements of *PATH*—namely, *path_within_block*, *path_between_blocks*, *path_to_env* and *path_from_env*—having the following meaning: [21]

1. $path\_within\_block : SR\_PATH \rightarrow PATH$ defines the encoding of signal routes between processes of the same block as elements of *PATH*;
2. $path\_between\_blocks : SR\_PATH \times CH\_PATH \times SR\_PATH \rightarrow PATH$ represents paths between processes that are assigned to different blocks;
3. $path\_to\_env : SR\_PATH \times CH\_PATH \rightarrow PATH$ represents paths leading from a system process to the environment;
4. $path\_from\_env : CH\_PATH \times SR\_PATH \rightarrow PATH$ represents paths leading from the environment to a system process.

Using the above definitions we can now provide the exact meaning of Pid_{ext} (introduced in Sect. 3.1.3):

$$\begin{aligned}
Pid_{ext} = \{pid \mid\ & PID(pid) \wedge \exists chp, srp, p : CH_PATH(chp) \wedge SR_PATH(srp) \\
& \wedge\ PATH(p) \wedge p = path_from_env(chp, srp) \\
& \wedge\ \neg Delaying(channel(chp)) \wedge to(srp) = procname(pid)\}.
\end{aligned}$$

3.2.2 Static Reachability Constraints

Consider the complete set of communication paths of a given SDL system. For each individual process instance set of the system one can now derive the subset of all communication paths leading from the respective process instance set to some other process instance set or to the external environment. Additionally, one can derive the set of signals that each path is able to convey. As result one

[21] Note that we can easily extend this representation to the more general SDL model with block partitioning. Even though *path_from_env* is actually not needed for the model as presented here (as the behavior of processes in the system environment is not explicitly addressed), we merely mention this path constructor for the sake of completeness and symmetry.

obtains the so-called *Reachability* set[22] of a process instance set (see Sect. 6.4 of Annex F3 to [27]), which is required to define the signal output behavior (see Sect. 5.2).

In general, a *Reachability* set may contain more than one communication path that is able to convey a certain signal to a specified destination. To further restrict the set of applicable paths, an SDL output statement (as will be explained in Sect. 5.2) may specify a number of path components (i.e., channel paths and signal route paths) that a matching path must contain. Nevertheless, the destination address and the information on the communication path may be ambiguous and incomplete; in particular, there may exist several valid destinations as well as more than one path to reach a certain destination.[23]

To solve the possible ambiguities, SDL chooses *nondeterministically* among the existing alternatives as represented by the respective *Reachability* set. With respect to a given output statement (see Sect. 5.2) that means, a matching receiver/path combination is to be selected depending on the *signal type*, the *destination address* (i.e., the **to**-argument) and the given *path constraints* (i.e., the **via**-argument). If no such combination exists, a signal will not be generated.

Selection of Receiver and Path. We assume here that all possible receiver/path combinations, as identified by the *Reachability* set associated with a process instance set, are represented through a (possibly empty) set of tuples $\langle p, r \rangle$, each of which specifies a receiver process $r \in PROCESS \cup \{env\}$ and a communication path $p \in PATH$.

In order to obtain a concise representation, it is convenient to determine the required reachability information by means of an abstract function

$$\begin{aligned} choose\_reachability : \; & PID \times SIGNAL \\ & \times (PROCESS \cup PID \cup \{undef\}) \\ & \times (SIGROUTE^* \cup CHANNEL^* \cup \{undef\}) \\ & \rightarrow PATH \times (PROCESS \cup \{env\}) \end{aligned}$$

such that *choose_reachability* when applied to the arguments as provided by an SDL output statement—namely: Pid (the sender process instance), *SName* (the signal type), *ToArg* (specifying the receiver), *ViaArg* (specifying the path constraints)—yields a tuple $\langle p, r \rangle$ from the corresponding *Reachability* set, provided that this set is nonempty, and $\langle undef, undef \rangle$, otherwise. For given arguments *Pid*, *SName*, *ToArg*, *ViaArg* the meaning of *choose_reachability* is as

[22] The computation of the *Reachability* sets is completely deterministic and can be carried out by means of a preprocessing step. Sect. 6.4 of Annex F3 to [27] describes an incremental construction procedure determining for each channel path and each signal route path of a system the corresponding *Reachability* set. The basic idea is to compute the required reachability information on the basis of inductively defined ingoing and outgoing partial *Reachability* sets, as associated with the adjacent paths segments of a given channel path or signal route path. With Basic SDL the situation is particularly simple because of the two-level hierarchy.

[23] If an output statement does neither specify a receiver nor a communication path, any process instance of the system for which there exists a suitable communication path may actually receive the resulting signal (see Sect. 2.7.4 of [27]).

follows:

$$choose\_reachability(Pid, SName, ToArg, ViaArg) =$$

$$\begin{cases} \langle p, r \rangle, & \text{if } \exists\, p \in PATH\ \exists\, r \in PROCESS \cup \{env\} : \\ & \qquad Reachable(p, r, Pid, SName, ToArg, ViaArg) \\ \langle undef, undef \rangle, & \text{otherwise;} \end{cases}$$

where *Reachable* represents a static predicate that holds on the arguments *Pid*, *SName*, *ToArg*, *ViaArg* iff the choice of $\langle p, r \rangle$ is compatible with the combination of values of *ToArg*, *ViaArg* (see Sect. 5.2) and p is able to convey a signal of type *SName* from the sender *Pid* to the destination r.

With respect to a possibly nondeterministic choice to be made in order to select $\langle p, r \rangle$, we additionally assume that the following constraint holds. In case that the underlying *Reachability* set contains another matching tuple $\langle q, s \rangle$, in addition to $\langle p, r \rangle$, and in the given ASM state the process instance set denoted by s is non-empty than the process instance set denoted by r is non-empty as well.

4 Behavior of Channels

The module Channel_Module consists of a single rule expressing how a channel agent delivers signal instances to specified receivers. In each computation step a channel agent c checks for each path *chp* such that *chp* is a channel path of c (i.e., $channel(chp) = c$) whether there is a signal instance s which has been conveyed on *chp* and can now be delivered to its destination.

In the rule below the selection of channel paths to be checked is specified by means of the `do forall`-construct. That is, the guarded update instruction in the body of the `do forall`-construct is applied to all elements *chp* in *CH_PATH* such that $channel(chp) = Self$. Provided that the condition stated by the auxiliary predicate *ReadyToDeliver* holds on *chp*, a corresponding signal instance from the channel queue of *chp* actually reaches its destination.

Depending on the location of the destination process instance, we can distinguish two cases: s is either appended to the input buffer of some system process instance (as stated by DeliverToProcess(s)) or it is delivered to the environment *env* (as stated by DeliverToEnv(s)). Since the propagation of signals within the external environment is outside the scope of the model captured by the definition of SDL, DeliverToEnv(s) 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 s = head(queue(chp)), r = receivername(s) in
              if r = env then
                 DeliverToEnv(s)
              else
                 DeliverToProcess(s, r)
    where
       ReadyToDeliver(chp)
         ≡ ∃s : SIGINST(s) ∧ s = head(queue(chp)) ∧ ¬InTransit(s, 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[24] (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 [27]) that the signal instance is discarded[25]. 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)⁀⟨SInst⟩
        else
            if PID_sys(PId) then
                buffer(PId) := buffer(PId)⁀⟨SInst⟩
```

5 Behavior of Processes

The underlying model of SDL processes is that of an Extended Finite State Machine. When started, a process performs a start transition and enters its first state. While executing a transition, the process may perform certain actions (e.g., sending or receiving signals or assigning values to variables). On completing a transition the process enters its next state.

Regarding the behavior of processes one can identify two basically different modeling aspects, namely: modeling of *control flow* and modeling of *signal transfer operations* and *timer operations*. To exemplify the formalization of typical SDL features, we concentrate here on the latter aspect (Sect. 5.2-Sect. 5.4), while control flow is addressed only very briefly[26] (Sect. 5.1). The resulting description forms the Process_Module of the abstract machine model.

[24] 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 [20]).

[25] 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 behavior?

[26] See [22] for a thorough description of control flow related behavior.

5.1 Modeling of Control Flow

We model control flow through operations on transition diagrams, usually called *flowcharts*[27]. Flowcharts provide a direct and intuitive means to formalize the control flow part as defined by the body of an SDL process. For each process type we derive a collection of flowcharts from the textual representation of an SDL system description. The start transition and each of the state descriptions in a process body are represented by separate flowcharts.

A flowchart is a directed graph with attributed nodes and labeled arcs. In our formal model of flowcharts, we represent nodes as elements of a domain *NODE*; additional domains—such as *STATEMENT*, *ARGUMENT*, *LABEL*, *STATE* and *RANGE*—are used for the representation of node attributes. Arcs are implicitly given through various operations defined on *NODE*. While processing a flowchart, a process agent or timer agent can then access the relevant information as explained below (where we consider here only the basic operations).

Consider a collection of flowcharts to be associated with some process agent denoted through *Self*. The actual flowchart node to be examined in a given abstract machine state is identified by *node*(*Self*). *start*(*procname*(*Self*)) yields the node to start with. For an arbitrary node n the two functions *stmt*(n) and *args*(n) yield an SDL statement together with the corresponding arguments (see Fig. 5).

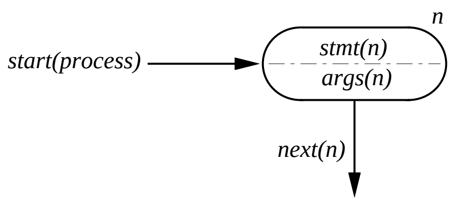

Figure 5: Representation of flowchart nodes

For the representation of edges we use three different functions. If there is no branching, *next*(n) points to the unique successor of node n. Otherwise, there are two kinds of branching to be distinguished depending on whether an **input** statement or a **decision** statement is to be processed. Below we illustrate the encoding of **input** statements in flowcharts.

On processing an **input** statement (see Sect. 5.3) with several alternatives (i.e., various types of signals that can be consumed or the option of performing a spontaneous transition) the choice of the successor node is expressed by a function *inputbranch*. Fig. 6 shows an example of a flowchart fragment as generated from an SDL state description with two possible inputs. The set of valid input signals and the set of signals to be saved in state S are given through corresponding functions *inputset* and *saveset*.

[27] The idea of modeling control flow by means of flowcharts is of course not new; similar concepts have for instance been applied in ASM-based semantic models of various other languages (e.g., Occam [8] and C [16]).

```
state S;
   input Sig1;
      task x:=5;
      ...
   input Sig2;
      output Sigout;
      ...
   save Sig3;
```

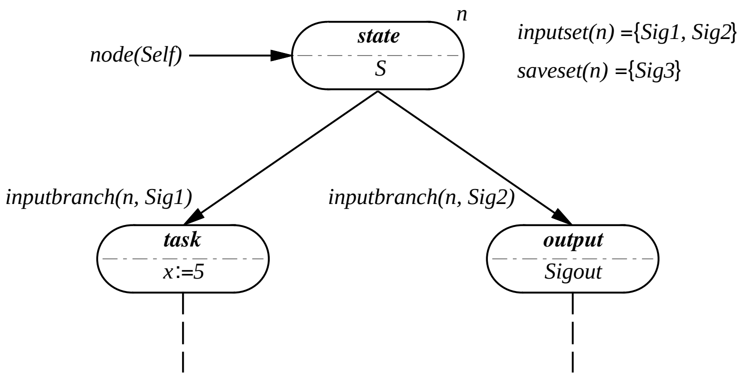

Figure 6: Flowchart generated from an SDL ***state*** description

Finally, there are operations on flowcharts that allow to switch between the individual flowcharts within the collection of flowcharts associated with a process agent. For instance, in the encoding of a **nextstate** statement (terminating an SDL transition) a function *stateloc* yields the node and the flowchart to proceed with as illustrated in Fig. 7.

As a primitive example of the interpretation scheme controlling the processing of flowcharts (as described in [22]) consider the following fragment for an output statement:

`case` $stmt(node(Self))$

$\vdots$

$output:$

$\quad$ $\text{O}\textsc{utput}\text{S}\textsc{ignal}\text{I}\textsc{nstance}(args(node(Self)))$

$\quad$ $node(Self) := next(node(Self))$

$\vdots$

`endcase`

where the rule OUTPUTSIGNALINSTANCE is defined in the Sect. 5.2 on signal output.

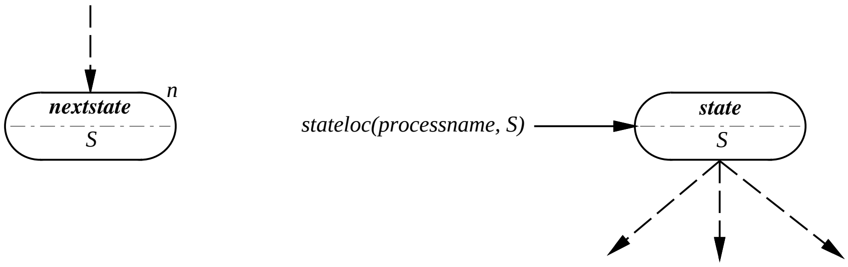

Figure 7: Application of the function *stateloc*

Process Creation and Process Termination. Process instances may either be created by other process instances or they may be generated as part of the initial state of an SDL system. In any case, it has to be ensured that the constraints for the respective process instance set are not violated.

Assuming that *PName* denotes an element of *PROCESS* and *Params* a matching list of process parameters, the corresponding abstract machine instruction $\textsc{CreateProcessInstance}(PName, Params)$ is defined below, where the initialization of variables and the assignment of values is left abstract.

$\textsc{CreateProcessInstance}(PName, Params)$
$\quad \equiv$ `if` $number\_of\_instances(PName) < max\_instances(PName)$ `then`
$\quad\quad$ `extend` PID `with` p
$\quad\quad\quad procname(p) := PName$
$\quad\quad\quad Mod(p) := \mathsf{Process\_Module}$
$\quad\quad\quad parent(p) := Self$
$\quad\quad\quad offspring(Self) := p$
$\quad\quad\quad number\_of\_instances(PName) := number\_of\_instances(PName) + 1$
$\quad\quad\quad node(p) := start(PName)$
$\quad\quad\quad \textsc{InitializeVars}(PName, p)$
$\quad\quad\quad \textsc{AssignActualParams}(PName, Params, p)$
$\quad$ `else`
$\quad\quad offspring(Self) := null$

On processing the SDL statement **stop** a process instance terminates itself[28]. That means, after the interpretation of this statement the process ceases to exist and thereby also the associated timer instances (see Sect. 5.4) become meaningless. In the abstract machine model a stop condition triggers the following simultaneous updates to be performed by a process agent (where the subrule $\textsc{StopTimerInstances}$ is defined in Sect. 5.4).

$\textsc{TerminateProcessInstance}$
$\quad \equiv Mod(Self) := undef$
$\quad\quad PID(Self) := False$
$\quad\quad \textsc{StopTimerInstances}$

[28] There is in fact no other way of terminating process instances of an SDL system.

5.2 Signal Output

An SDL output statement has basically the following form

> **output** $\langle signal \rangle$ $[(\langle value \rangle\{,\ \langle value \rangle\}^*)]$
> $[$**to** $\langle pid \rangle \mid \langle process \rangle]$
> $[$**via** $[$**all**$]$ $\{\langle channel \rangle | \langle signalroute \rangle\ \{,\ \langle channel \rangle | \langle signalroute \rangle\}^*\}]$,

where the signal values, the **to**-argument and the **via**-argument are optional; only a signal name must be specified. The address of a destination process may either be given *explicitly* (by the **to**-argument), *implicitly* through the system structure (if there is only one possible destination), or by naming some signal route or channel in the **via**-argument.

If the **to**-argument is defined, it either specifies a PId (i.e., some element from PID_{sys} or PID_{env}) or it just denotes a process instance set out of which a receiver process instance is to be selected. Otherwise, if the **to**-argument is undefined, a destination process instance is to be determined by means of the associated *Reachability* set.

The **via**-argument may state additional constraints restricting the possible choices of receiver/path combinations. **via all** represents *multicast* and is actually an abbreviation used to send identical copies of a specified signal to all destinations[29] that can be reached (depending on the available paths) from a given process instance (see Sect. 2.7.4 of [27]); as such it can easily be transformed into a number of output statements in standard form as part of a preprocessing step and is therefore not further addressed here.

On processing an SDL **output** statement a process initiates a send operation (provided that a valid receiver exists) by creating a new signal instance. In the corresponding abstract machine instruction[30] OUTPUTSIGNALINSTANCE(..), as defined below, the operations DEFSIGNALINST(..) and SENDSIGNALINST(..) resp. refer to the creation of the signal instance and to the effective output operation. Note that both operations can occur in the same computation step of the abstract machine as the new signal instance is not globally visible (and thus cannot be consumed by another process agent – see Sect. 5.3) prior to the next computation step.

The arguments of OUTPUTSIGNALINSTANCE(..) denote the corresponding parameter values of an output statement, namely the signal type (*SName*), a possibly empty list of signal values (*Val*), the value of the **to**-argument (*ToArg*), and the value of the **via**-argument (*ViaArg*). As explained in Sect. 3.2.2, the function *choose_reachability* is used to determine a matching process instance set r together with a suitable path p.

Recall that an output statement has no effect (i.e., does not generate a signal instance) in each of the following situations (see Sect. 2.7.4 of [27]):

1. a communication path able to convey a signal of the specified type to the matching destination does not exist;

---

[29] In combination with the **via all** option an output statement must not specify any destination (neither a receiver process instance nor a process instance set).

[30] To create a new signal instance s, means to extend the domain *SIGINST* with some 'fresh', element s (from the base set). The intuitive meaning of this operation is obvious and requires actually no further explanation. (For a formal treatment of the semantics of the `extend`-construct see [20].)

2. the specified destination is a process instance which does not exsist;
3. the specified destination is a process instance set which is empty.

$$
\begin{array}{l}
\textsc{OutputSignalInstance}(\mathit{SName}, \mathit{Val}, \mathit{ToArg}, \mathit{ViaArg}) \\
\quad \equiv \texttt{let } \langle p, r\rangle = \mathit{choose\_reachability}(\mathit{Self}, \mathit{SName}, \mathit{ToArg}, \mathit{ViaArg}) \texttt{ in} \\
\qquad\quad \texttt{if } p \neq \mathit{undef} \texttt{ then} \\
\qquad\qquad \texttt{if } r = \mathit{env} \lor \mathit{PID}_{sys}(\mathit{ToArg}) \lor \mathit{ValidReceiver}(r) \texttt{ then} \\
\qquad\qquad\quad \texttt{extend } \mathit{SIGINST} \texttt{ with } s \\
\qquad\qquad\qquad \textsc{DefSignalInst}(s) \\
\qquad\qquad\qquad \texttt{if } \mathit{PID}(\mathit{ToArg}) \texttt{ then} \\
\qquad\qquad\qquad\quad \textsc{SendSignalInst}(s, \mathit{ToArg}, r, p) \\
\qquad\qquad\qquad \texttt{else} \\
\qquad\qquad\qquad\quad \textsc{SendSignalInst}(s, \mathit{undef}, r, p) \\
\qquad \texttt{where} \\
\qquad\quad \mathit{ValidReceiver}(r) \equiv \ \mathit{PROCESS}(\mathit{ToArg}) \land \mathit{NonEmpty}(r) \\
\qquad\quad \mathit{NonEmpty}(r) \equiv \ \exists x : \mathit{PID}_{sys}(x) \land \ \mathit{procname}(x) = r \\
\qquad\quad \textsc{DefSignalInst}(s) \\
\qquad\quad\ \equiv \ \mathit{values}(s) := \mathit{Val} \\
\qquad\qquad \mathit{senderid}(s) := \mathit{Self} \\
\qquad\qquad \mathit{signame}(s) := \mathit{SName} \\
\qquad\qquad \mathit{path}(s) := p \\
\qquad\qquad \texttt{if } \mathit{PID}(\mathit{ToArg}) \texttt{ then} \\
\qquad\qquad\quad \mathit{receiverid}(s) := \mathit{ToArg}
\end{array}
$$

In the definition of $\textsc{SendSignalInst}(..)$ the arguments sr, sr' refer to elements of $\mathit{SR\_PATH}$ while chp refers to an element of $\mathit{CH\_PATH}$.[31]

$$
\begin{array}{l}
\textsc{SendSignalInst}(\mathit{SInst}, \mathit{PId}, \mathit{PName}, \mathit{Path}) \\
\quad \equiv \texttt{if } \mathit{Path} = \mathit{path\_to\_env}(sr, chp) \texttt{ then} \\
\qquad\quad \texttt{if } \mathit{Delayed}(chp) \texttt{ then} \\
\qquad\qquad \textsc{EnqueueToChannel}(\mathit{SInst}, chp) \\
\qquad\quad \texttt{else} \\
\qquad\qquad \textsc{DeliverToEnv}(\mathit{SInst}) \\
\qquad \texttt{elif } \mathit{Path} = \mathit{path\_between\_blocks}(sr, chp, sr') \land \mathit{Delayed}(chp) \texttt{ then} \\
\qquad\quad \textsc{EnqueueToChannel}(\mathit{SInst}, chp) \\
\qquad \texttt{elif } \mathit{PID}_{sys}(\mathit{PId}) \texttt{ then} \\
\qquad\quad \mathit{buffer}(\mathit{PId}) := \mathit{buffer}(\mathit{PId})^\frown\langle \mathit{SInst}\rangle \\
\qquad \texttt{else} \\
\qquad\quad \texttt{choose } p : \ \mathit{PID}(p) \ \mathit{and} \ \mathit{procname}(p) = \mathit{PName} \\
\qquad\qquad \mathit{buffer}(p) := \mathit{buffer}(p)^\frown\langle \mathit{SInst}\rangle \\
\qquad \texttt{where} \\
\qquad\quad \textsc{EnqueueToChannel}(\mathit{SInst}, chp) \\
\qquad\quad\ \equiv \ \mathit{queue}(chp) := \mathit{queue}(chp)^\frown\langle \mathit{SInst}\rangle \\
\qquad\quad \mathit{Delayed}(chp) \equiv \mathit{Delaying}(\mathit{channel}(chp))
\end{array}
$$

Even in case that two or more processes concurrently perform send operations using the same channel path chp of some delaying channel $\mathit{channel}(chp)$, they

[31] To improve the readability, we employ here a simple *pattern matching* notation (with an obvious meaning) for accessing path components and thereby avoid the explicit definition and use of corresponding selector functions.

cannot interfere with each other (as implied by the notion of concurrency in the semantic model of multi-agent ASMs – see Sect. 2.2.1).

5.3 Signal Consumption

Consider a process instance that is waiting in one of the defined process states (see Sect. 5.1) to perform an **input** statement. For the process agent processing the flowchart that means that *stmt*(*node*(*Self*)) is equal to *state*, where *node*(*Self*) identifies the node in the flowchart specifying the current action.

In general, a transition can either be initiated by consuming an input signal from the input buffer or, if the process body contains the SDL construct **input none**, a transition may as well occur *spontaneously* (i.e., without consuming any input signals). The SDL concept of spontaneous transitions allows to choose nondeterministically between regular transitions and spontaneous transitions. Formally, this decision can be stated in abstract terms by introducing a unary monitored predicate *Spontaneous*[32], as in the following rule.

```
WaitingInState
  ≡ if Waiting then
        if RegularTrans then
           ConsumeInputSignal
        else
           SpontaneousTransition
    where
        Waiting ≡ stmt(node(Self)) = state
        RegularTrans
         ≡ ¬(none ∈ inputset(node(Self)) ∧ Spontaneous(Self))
```

The dynamic function *sender* (see Sect. 3.1.1) is used to store the PId of the sender (as represented by the **sender** expression of SDL). In case of a spontaneous transition, *sender* refers to the process instance performing the spontaneous transition (see Sect. 2.6.6 of [27]). In addition to updating *sender* accordingly, the process agent switches to the node of his flowchart to proceed with, as identified through *inputbranch*(*node*(*Self*), *none*).

```
SpontaneousTransition
  ≡ sender(Self) := Self
    node(Self) := inputbranch(node(Self), none))
```

According to the **input** declaration and the **save** declaration for SDL states, there are two disjoint sets of signal types to be associated with a given state. The

[32] Note that the value of *Spontaneous* depends on the particular process instance because there may be different results for different process instances simultaneously performing such a choice. For the purpose considered here, the ASM `choose`-construct offers an alternative way of modelling nondeterminism (e.g., by choosing an arbitrary element from *BOOL*) with an absolute identical meaning. Whether the decision on performing a spontaneous transition also depends on the environment (i.e., is treated as *don't-know* nondeterminism), as expressed by the predicate *Spontaneous*, or it depends only on the abstract machine model (i.e., is treated as *don't-care* nondeterminism), as expressed by the `choose`-construct, is actually a matter of taste and has no further impacts on the resulting behavior. In other words, both solutions are possible.

first one specifies which of the signals in the input buffer are actually considered as valid input signals. The second one specifies which signals must be retained in the input buffer (to be consumed in another state). Any other signal not addressed by one of these sets can be discarded through an *implicit transition* (some kind of empty transition leading immediately back to the same state).

The detailed input behavior is specified in terms of the abstract machine instruction CONSUMEINPUTSIGNAL using an auxiliary function *next_signal* (see below) for computing the content of the actualized input buffer and the input signal to be consumed. (For simplicity, the assignment of signal values to the variables of the process performing the input operation is left abstract).

$$
\begin{array}{l}
\textsc{ConsumeInputSignal} \\
\quad \equiv \texttt{let}\ \langle \mathit{new\_buffer}, s\rangle = \mathit{next\_signal}(\mathit{empty\_buffer}, \mathit{buffer}(\mathit{Self}), \mathit{Self})\ \texttt{in} \\
\qquad\quad \mathit{buffer}(\mathit{Self}) := \mathit{new\_buffer} \\
\qquad\quad \texttt{if}\ s \neq \mathit{undef}\ \texttt{then} \\
\qquad\qquad \mathit{sender}(\mathit{Self}) := \mathit{senderid}(s) \\
\qquad\qquad \textsc{AssignValues}(\mathit{inputvars}(\mathit{node}(\mathit{Self}), \mathit{signame}(s)), \mathit{values}(s)) \\
\qquad\qquad \mathit{node}(\mathit{Self}) := \mathit{inputbranch}(\mathit{node}(\mathit{Self}), \mathit{signame}(s))
\end{array}
$$

The recursively defined function *next_signal* searches the input buffer (in a FIFO order) for a valid input signal. If a matching signal instance is found, it returns a tuple $\langle \mathit{new\_buffer}, si\rangle$ consisting of an actualized input buffer *new_buffer* and an input signal instance s; otherwise, *new_buffer* is identical with the old input buffer and s is set to *undef*.

If the input operation succeeds, *new_buffer* is obtained from a given input buffer by discarding (from the searched part of the input buffer) the signal instance which is to be consumed as well as all those signal instances which need not to be saved in the current state.

Otherwise, if the input operation fails, the input buffer is completely searched discarding all signal instances that are not to be saved. (In the flowchart the information on the input signal set and the set of signals to be saved is attached to $\mathit{node}(\mathit{Self})$ and can be accessed through corresponding functions *inputset*, *saveset*).

$$
\mathit{next\_signal}(\mathit{Saved}, \mathit{Buffer}, \mathit{Pid}) =
$$

$$
\begin{cases}
\langle \mathit{Saved}, \mathit{undef}\rangle, & \text{if}\ \mathit{Buffer} = \mathit{empty\_buffer}, \\
\langle \mathit{Saved} \frown \mathit{Rest}, si\rangle, & \text{if}\ \mathit{Buffer} = \langle si | \mathit{Rest}\rangle \\
 & \quad \wedge\ \mathit{signame}(s)\ \mathit{in}\ \mathit{InputSet}, \\
\mathit{next\_signal}(\mathit{Saved}, \mathit{Rest} \frown \langle si\rangle, \mathit{Pid}), & \text{if}\ \mathit{Buffer} = \langle si | \mathit{Rest}\rangle \\
 & \quad \wedge\ \mathit{signame}(s)\ \mathit{in}\ \mathit{SaveSet}, \\
\mathit{next\_signal}(\mathit{Saved}, \mathit{Rest}, \mathit{Pid}), & \text{if}\ \mathit{Buffer} = \langle si | \mathit{Rest}\rangle \\
 & \quad \wedge\ \mathit{signame}(s)\ \mathit{not\ in}\ \mathit{SaveSet} \\
 & \quad \wedge\ \mathit{signame}(s)\ \mathit{not\ in}\ \mathit{InputSet}
\end{cases}
$$

$$
\begin{array}{l}
\texttt{where} \\
\quad \mathit{InputSet} \equiv \mathit{inputset}(\mathit{node}(\mathit{Pid})), \quad \mathit{SaveSet} \equiv \mathit{saveset}(\mathit{node}(\mathit{Pid}))
\end{array}
$$

5.4 Timer Operations

An SDL timer is owned by a process instance which controls the timer through operations **set** and **reset**. At any given moment of time a timer may be *active* or *inactive*. With an active timer SDL associates either a time value indicating the expiration time or a *timer signal*. When a timer expires it creates a timer signal and appends it to the input buffer of the owner process in order to notify the owner process about the timer event. A timer expires as soon as the value of **now** becomes equal or greater than the expiration time.

An active timer remains active after it has expired until one of the following events occur: (1) the timer signal is consumed by the related process instance, or (2) the related process instance explicitly inactivates the timer through a **reset** operation. The latter event also means that the corresponding timer signal is discarded from the input buffer of the related process instance.

A process instance may in general employ any number of concurrently operating timer instances, where the timer signals uniquely identify the respective timer instance they originate from. In other words, timer instances have their own behavior. The abstract machine model thus identifies timer instances with corresponding ASM agents, called *timer agents* (see Sect. 6). Though several timer instances of the same process instance may expire simultaneously, they somehow agree upon the order in which the respective timer signals are to be appended to the input buffer of the owner process (as already explained, this property is ensured by the notion of partially ordered run – Sect. 2.2.1).

Representation of Timer Instances. A timer instance t is uniquely identified through a triple *(Pid,Timer,Params)*, where Pid denotes the process instance, *Timer* refers to the name and $Params$ to a list of values[33] associated with t. We therefore introduce a dynamic function

$$timerinst : PID \times TIMER \times VALUE^* \rightarrow TIMERINST$$

together with corresponding selector functions *owner*, *timername*, *params* from *TIMERINST* into *PID*, *TIMER* resp. $VALUE^*$. A dynamic function *expire* from *TIMERINST* to *TIME* indicates the expiration time associated with t, where $expire(t)$ is set to *undef* each time t expires.

ASM agents representing timer instances are created dynamically at the moment a timer instance is referred to for the first time. In general, the first use of a timer may be a **set** operation as well as a **reset** operation (see Sect. 2.8 of [27]),

$$\textbf{set}\ (Time, TName(Params)),\ \ \textbf{reset}\ (TName(Params)),$$

where $Time$ specifies a value of *TIME*, *TName* specifies a value of *TIMER* and $Params$ is an element from $VALUE^*$. To introduce a new timer instance, a process agent performs the following abstract machine instruction[34] (where *Self*

[33] In SDL, a timer declaration specifies a timer name in combination with a (possibly empty) list of SDL sorts (see Sect. 3.1) representing admissible timer values. Several timers having the same name may thus be distinguished through different lists of values associated with the timer name.

[34] Recall from the definition of multi-agent ASMs (see Sect. 2.2.1) that the function *Mod* allows to dynamically assign a behavior to an arbitrarily chosen element from the underlying base set.

identifies the process agent).

$$
\begin{array}{l}
\textsc{CreateTimerInstance}(\mathit{TName}, \mathit{Params}) \\
\quad \equiv \texttt{extend}\ \mathit{TIMERINST}\ \texttt{with}\ t \\
\qquad\quad \mathit{timerinst}(\mathit{Self}, \mathit{TName}, \mathit{Params}) := t \\
\qquad\quad \mathit{timername}(t) := \mathit{TName} \\
\qquad\quad \mathit{params}(t) := \mathit{Params} \\
\qquad\quad \mathit{owner}(t) := \mathit{Self} \\
\qquad\quad \mathit{Mod}(t) := \mathsf{Timer\_Module}
\end{array}
$$

Finally, we have to formalize the meaning of the rule StopTimerInstances introduced in Sect. 5.1.

$$
\begin{array}{l}
\textsc{StopTimerInstances} \\
\quad \equiv \texttt{do forall}\ t:\ \mathit{TIMERINST}(t) \\
\qquad\quad \texttt{if}\ \mathit{owner}(t) = \mathit{Self}\ \texttt{then} \\
\qquad\qquad \mathit{Mod}(t) := \mathit{undef} \\
\qquad\qquad \mathit{TIMERINST}(t) := \mathit{False}
\end{array}
$$

6 Behavior of Timers

Timer agents execute the module Timer_Module consisting of the rules defined in this section. We start by introducing a few auxiliary predicates. To express the relationship between timer agents and process agents (see Sect. 5.4), we assume to have a corresponding mapping *owner* from *TIMERINST* to *PID*. Additionally, we assume that *timer* denotes a mapping from *SIGINST* to *TIMERINST* associating the timer signals in *SIGINST* with the timers they originate from[35].

$\mathit{ActiveTime}(t) \equiv$

$$\mathit{TIMERINST}(t) \wedge \mathit{expire}(t) \neq \mathit{undef}$$

$\mathit{ActiveSignal}(t) \equiv$

$$\mathit{TIMERINST}(t) \wedge \exists\ s \in \mathit{SIGINST}:\ t = \mathit{timer}(s)$$

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

$\mathit{Active}(t) \equiv$

$$\mathit{ActiveTime}(t) \vee \mathit{ActiveSignal}(t)$$

A timer agent t watches the activities of $\mathit{owner}(t)$ and becomes involved each time $\mathit{owner}(t)$ encounters a **set** or **reset** instruction (see Sect. 5.4). 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 $\mathit{expire}(t)$ in order to generate a timer signal.

[35] Note that for all signals s in *SIGINST* not being timer signals the value of $\mathit{timer}(s)$ is *undef*.

In the definition of TIMEROPERATION below, the condition $MyAction$ triggers the timer agent. In case that $MyAction$ holds on $Self$, $Action$ is either set or $reset$; 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) ∧ now ≥ expire(Self) then
            expire(Self) := undef
            CreateTimerSignal

    where
        Action ≡ stmt(node(owner(Self)))
```

The value of $MyAction(Self)$ depends on the attribute values of the flowchart node currently being inspected by $owner(Self)$.[36] To denote these values, we introduce $Action$ and Arg as abbreviations for $stmt(node(owner(Self)))$ and $args(node(owner(Self)))$. We can then define the precise meaning of $MyAction$ as follows.

$$
\begin{aligned}
&MyAction(Self)\\
&\quad\equiv\ \exists\, x \in TIMER, par \in VALUE^{*}:\ timerinst(owner(Self), x, par) = Self\\
&\qquad\wedge(\ (\exists\, t \in TIME : Arg = \langle t, x, par\rangle \wedge Action = set)\\
&\qquad\quad\vee(Arg = \langle x, par\rangle \wedge Action = reset)\ )
\end{aligned}
$$

[36] Alternatively, we could have modeled the operations **set** and **reset** such that they are executed by the process agent (instead of the timer agent). Nevertheless, there arises a problem (which is not addressed in Z.100): if a timer is reset at exactly the moment it expires, then it generates a timer signal which can not be eliminated by the process agent since it is not yet contained in the input buffer of the process agent. Of course, there are solutions to cope with that problem—our model however simply avoids that problem.

The value expressed by the unary function *duration* in the following rule is either "0" or a default value derived from the timer definition[37].

$\textsc{SetExpirationTime}(Time)$
$\quad \equiv$ `if` $Time = undef$ `then`
$\qquad expire(Self) := now + duration(timername(Self))$
$\quad$ `elif` $Time \leq now$ `then`
$\qquad expire(Self) := undef$
$\qquad \textsc{CreateTimerSignal}$
$\quad$ `else`
$\qquad expire(Self) := Time$

Recall the meaning of the function *sender* defined on process instances (see Sect. 5.3). On receiving a timer signal the PId of the sender of that timer signal is defined to be the PId of the process instance that receives the timer signal (see Sect. 2.8 of [27]). In the definition of CreateTimerSignal below the value of *owner*(*Self*) (i.e., the PId of the owner process) thus identifies the sender of the timer signal s.

$\textsc{CreateTimerSignal}$
$\quad \equiv$ `extend` $SIGINST$ `with` s
$\qquad timer(s) := Self$
$\qquad senderid(s) := owner(Self)$
$\qquad receiverid(s) := owner(Self)$
$\qquad signame(s) := timername(Self)$
$\qquad values(s) := params(timername(Self))$
$\qquad buffer(owner(Self)) := buffer(owner(Self))^\frown\langle s\rangle$

DiscardTimerSignal operates on the input buffer of *owner*(*Self*). A recursively defined function *cleaned_buffer* computes the resulting input buffer obtained by discarding a timer signal s such that $timer(s) = Self$. (Note that *buffer*(*owner*(*Self*)) contains exactly one such signal s if *ActiveSignal*(*Self*) holds.)

$\textsc{DiscardTimerSignal}$
$\quad \equiv$ `if` $ActiveSignal(Self)$ `then`
$\qquad buffer(owner(Self)) := cleaned\_buffer(empty\_buffer, buffer(owner(Self)))$

$$cleaned\_buffer(Searched, Buffer) =$$

$$\begin{cases} Searched^\frown Rest, & \text{if } Buffer = \langle s|Rest\rangle \wedge\ timer(s) = Self, \\ cleaned\_buffer(Searched^\frown\langle s\rangle, Rest), & \text{if } Buffer = \langle s|Rest\rangle \wedge\ timer(s) \neq Self. \end{cases}$$

[37] 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 [27]).

7 Conclusions

The mathematical modeling concept of multi-agent real-time ASM and the SDL view on asynchronously communicating systems clearly coincide with respect to essential properties of the underlying computation models. To obtain a concise representation, we have utilized another important advantage: *scalability* of the abstraction mechanism—i.e., the flexibility to freely choose and combine various abstraction levels within a single mathematical model. By avoiding additional (or unnatural) formal overhead we were able to produce a description of a reasonable size.

Our interpretation model of Basic SDL, as given here, captures all relevant aspects of the dynamic semantics, except for the control flow operations (which are simple to model but are left out for brevity). The complete model (without the initial state definition) has about double the size of the one given here.

An obvious way of extending the current model is to include the full structuring facilities of SDL. Based on our experience with VHDL [5], which offers structuring concepts similar to block partitioning in SDL, we can state that this does not cause any real problems.

It should also be mentioned that we have tools to run ASM models on real machines [12, 11]. Provided that the details which we left abstract in our SDL model (this mainly concerns interfaces to the external environment) are handled properly (e.g., by specifying them through user inputs, external processes etc.), we are able to produce an executable version of our SDL model.

Finally, there are existing ASM-based concepts and tools addressing modeling of static semantics [23]. Combining these approaches with our model will allow to produce a complete semantic model including all static and dynamic properties of SDL.

Acknowledgements. We thank Giuseppe Del Castillo, Egon Börger and Peter Päppinghaus for inspiring discussions contributing to our work as well as three unknown reviewers for receiving helpful comments and valuable criticism on a draft version of the work presented here.

References

1. Ch. Beierle, E. Börger, I. Đurđanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *LNCS (State–of–the–Art Survey)*, pages 52–78. Springer-Verlag, 1996.
2. F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Carl Hanser Verlag / Prentice Hall International, 1991.
3. 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.
4. E. Börger and U. Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon, editors, *Proc. of the IFIP 13th World Computer Congress 1994, Volume I: Technology and Foundations*, pages 402–409. Elsevier Science Publishers B. V., 1994.

5. E. Börger, U. Glässer, and W. Mueller. 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.
6. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'92 descriptions. In *Proc. of EURO-VHDL'94 (September 19-20, Grenoble)*, 1994.
7. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *Journal of Universal Computer Science (J.UCS)*, 3(5):603–665, May 1997.
8. E. Börger, I. Đurđanović, and D. Rosenzweig. Occam: Specification and compiler correctness. part i: The primary model. In E.-R. Olderog, editor, *Proc. of PRO-COMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.
9. 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.
10. Manfred Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing 3*, (3):21–57, 1991.
11. G. Del Castillo, I. Đurđanović and U. Glässer. An evolving algebra abstract machine. In H. Kleine Büning, editor, *Computer Science Logic*, volume 1092 of *LNCS*, pages 191–214. Springer-Verlag, 1996.
12. Giuseppe Del Castillo. ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial. Technical Report (to appear).
13. O. Færgemand and A. Olsen. Introduction to SDL-92. *Computer Networks and ISDN Systems*, (26):1143–1167, 1994.
14. 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.
15. J. Fischer, St. Lau, and A. Prinz. A Short Note About BSDL - Semantic Issues for SDL. *SDL Newsletter*, (18), January 1995.
16. Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
17. Y. Gurevich and J. Huggins. The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In H. Kleine Büning, editor, *Computer Science Logic*, volume 1092 of *LNCS*, pages 266–290. Springer-Verlag, 1996.
18. Yuri Gurevich. Evolving algebras – a tutorial introduction. *Bulletin of the EATCS*, (43):264–284, February 1991.
19. Yuri Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
20. Yuri Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan–Ann Arbor, 1997.
21. 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.
22. René Karges. Formale Semantik von SDL als abstraktes Zustandssystem. (Diploma thesis), Fachbereich Mathematik-Informatik, Universität-GH Paderborn, March 1997.
23. P. W. Kutter and A. Pierantonio. Montages: Specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
24. St. Lau and A. Prinz. BSDL: The Language – Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995.
25. A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. Elsevier Science B. V., 1994.

26. Markus Rinderspacher. A Verification concept for SDL systems and its application to the Abracadabra Protocol. Interner Bericht 14/94, Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, 1994.
27. ITU-T Recommendation Z.100. *Specification and Description Language (SDL).* International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.