Towards a New Formal SDL Semantics

R. Gotzhein, B. Geppert, F. Rößler, P. Schaible
Computer Networks Group,University of Kaiserslautern
University of Kaiserslautern, Postfach 3049, D-67653 Kaiserslautern, Germany
Tel: +49 631 205-3426, Fax: +49 631 205-3956, Email: gotzhein@informatik.uni-kl.de

Abstract

In 1988, a formal semantics for SDL has been added as Annex F to the Z.100 SDL standard [5,6,7,8]. Along with the efforts to improve SDL, the semantics has been revised several times since then. To understand the semantics, intimate knowledge of the language Meta IV is required. Meta IV is a formal language based on synchronous communication between a set of concurrent processes. Essentially, Annex F defines a sequence of Meta IV programs that take an SDL specification as input, determine the correctness of its static semantics, perform a number of transformations to replace several language constructs, and interpret the specification. It has been argued that this style of defining the formal semantics is particularly suitable for tool builders.

With the ongoing work to improve SDL, and a new version called SDL-2000 to pass the standardization bodies shortly, it has become apparent that a decision concerning the SDL semantics needs to be taken. One alternative is to revise the existing SDL semantics, which has the advantage of continuity. Another alternative currently under consideration is to redo the SDL semantics, which offers an opportunity for improvement. In this paper, some of the choices that go along with a new definition of a formal SDL semantics are presented and discussed. Based on the results of this discussion, a coarse outline for a new formal semantics is proposed.

Keywords
formal description technique, formal semantics, SDL

1 SOME DESIGN OBJECTIVES

Among the primary design objectives of a formal SDL semantics is intelligibility, a prerequisite for acceptance, correctness, and maintainability. Generally speaking, intelligibility can be achieved by building on well-known mathematical models and notation, a close correspondance between specification and underlying model, and by a concise and well-structured documentation.

As SDL is an evolving language, it can be expected that there will be further versions in the near future, therefore, maintainability is of similar importance. There are currently several language extensions under discussion, including exception handling, dynamic modification of system structure, real time expressiveness beyond timers, and performance specification, as well as removal of some language features. Therefore, the semantical model has to be sufficiently rich and flexible such that these aspects can be formalized with a reasonable effort.

Apart from assigning a unique meaning to SDL specifications, a formal semantics should support formal analysis as well as implementation steps. Even if performance aspects are not directly covered by language constructs, it should be possible to assign a formal meaning to suitable annotations, based on the same semantical model. This would allow to use the same model both for qualitative reasoning as well as for performance and predictability analysis. Implementation support concerns the straightforward generation of correct code that can be executed with varying degrees of parallelism. This can typically be achieved by an operational semantics and a suitable model of concurrency.

Finally, SDL is a language that has been successfully used for about 20 years. This observation has an important implication: there is no room for experiments for defining a formal semantics, but the formal semantics must support SDL as it is currently used [5].

2 DESIGN CHOICES

Before and during the design of a formal SDL semantics, a number of decisions have to be made. Some of the options are addressed subsequently (for details, see also [4]).

2.1 Language coverage

In general, the set of syntactically correct SDL specifications for which a formal semantics is defined, called "*language coverage*" in the following, should be as large as possible. There are, however, certain limitations as well as pragmatic considerations that may reduce language coverage. Without going into technical detail, we will address some of the considerations in this respect.

An important criterion for language coverage is that any SDL specification for which a formal semantics is to be defined has to be "complete" from a formal point of view. A specification containing informal text elements - for instance, informal question and answers attached to a decision-symbol - would be incomplete in this sense. In general, a complete specification should formally define a set of computations or execution sequences, in one way or another. This rules out any informal elements other than comments or annotations, even if they are used in a syntactically correct manner.

A pragmatic approach to achieve a good language coverage is "bottom-up": starting from a set of *basic language constructs* for which a formal semantics is defined in a rigorous way, further language constructs are included step by step. This means that an SDL core language is defined, and that a formal semantics is assigned to specifications written in this core language. In subsequent steps, this core language is extended by language constructs that can be defined "syntactically", i.e. by translation to the SDL core language, sometimes called "normalization". Candidates are shorthands such as the *-symbol used in process graphs, structuring elements such as services, and object-oriented language constructs such as redefinition of virtual transitions. Normalization could simply be done by textual replacements, but may also involve transformations, as it would be the case when a set of services is replaced by their product automaton.

SDL offers a number of object-oriented language constructs, including block types, process types, specialization, virtual types, and virtual transitions. Object-oriented constructs provide support for structuring a specification and thus to improve its readability and its maintainability. Also, they allow to postpone some design decisions by introducing supertypes that may be specialized or partially redefined. As noted before, it is required that SDL specifications for which a formal semantics is to be defined be complete. Therefore, it should in principle be possible to eliminate object-oriented language constructs through normalization steps.

SDL supports the specification of systems that are open in a topological sense, i.e. systems that may interact with some environment. Strictly speaking, this type of specification may be considered incomplete from a formal point of view, since the environment is not known. This, however, is a frequent case in distributed systems development, and therefore, it is important that specifications of open systems have a formal semantics. We will address this topic in more detail in a separate section below.

2.2 Mathematical model

An important design decision concerns the class of mathematical models that underlies the formal semantics. Apart from personal preferences that may be influenced by one's background, the pros and cons for possible choices should be considered carefully before a decision is made.

The pro of utmost importance for a transition system is the fact that intuitively, the underlying semantical model is a *set of extended finite state automata* communicating *asynchronously*. This intuition should somehow be reflected in the semantics, which would certainly make it better understandable. Also, the incorporation of time and performance aspects is well supported in the context of transition systems. On the other hand, abstract data types are specified algebraically in SDL, which would lead to a slightly heterogeneous formal semantics.

Important pros for a process algebra are the availability of a bulk of scientific results and the fact that algebraic specifications of abstract data types can be accommodated in a straightforward manner. On the other hand, the interaction paradigm of most process algebras is *synchronous* communication, which creates a gap between intuitive and mathematical model. Also, scientific results are usually based on *basic* process algebras, which are not sufficiently expressive to serve as an underlying semantical model.

2.3 Interleaving vs. non-interleaving

Most FDTs in the area of concurrent systems such as SDL, Estelle and Lotos have an interleaving semantics, which means that all activities are sequentialized in the underlying model. The intention, though, is to allow for parallel implementations. However, sequentializations are much easier to treat both to define the formal semantics and

when analyzing a given system. The reason is that at any time, one deals only with single transitions, not with sets of transitions. Alternatively, it may be possible to define a non-interleaving semantics, sometimes called "true-concurrency semantics" [1]. The advantage here could be that existing causal dependences are explicitly stated and not buried in the set of all possible interleavings.

2.4 Granularity of state transitions

An important design choice concerns the granularity of *state transitions*, i.e. state changes in the underlying semantical model[1]. The concept of "state transition" can be identified both in transition systems as well as in process algebras that have an operational semantics. On the set of state transitions, a transitive, asymmetrical relation called *causal dependence relation* can be defined. This relation captures the degree of concurrency: two state transitions are called *concurrent* if they are not causally dependent. The causal dependence relation can be represented in different ways. Direct representations exist, for instance, in non-interleaving models, while it is indirectly expressed in interleaving models.

Given a particular semantical model, the state transitions of that model are atomic execution units by definition. However, there are several alternatives of associating state transitions with SDL specifications. For instance, a state transition could correspond to an SDL action (e.g., task, output, set, call), some sequence of SDL actions, an SDL transition, or be more or less unrelated to the SDL specification. If state transitions are related to the SDL specification, the causal dependence relation can be established not only in the underlying model, but also on the specification level. We consider this as essential for the intelligibility of SDL specifications, because it is a prerequisite for understanding the specified concurrency. Therefore, we require that state transitions and SDL specifications be related.

In the existing SDL semantics, a state transition corresponds to an (explicit or implicit[2]) SDL *action* execution[3]. This satisfies our basic requirement that state transitions and SDL specifications be related. Moreover, it means that causal dependence is defined between SDL actions. Thus, in order to understand a specification, all possible sequences of SDL *action* executions that obey the causal dependence relation have to be taken into account. Alternatively, a state transition may correspond to an SDL *transition* execution, which would also satisfy the basic requirement stated earlier. Consequently, causal dependence can be established between SDL transitions. In order to understand a specification, all possible sequences of SDL transition executions that satisfy this relation have to be considered.

In general, it can be observed that a finer granularity of state transitions increases the specified system behaviour. Thus, their granularity has some impact both on the intelligibility and the analysis of SDL specifications. It is evident that in general, it is more difficult to foresee all possible system behaviour, if state transitions correspond to SDL action executions. We note here that it has recently been clarified in the standardization body that the atomicity of SDL is on the level of actions rather than on the level of transitions. Since the formal semantics must support SDL exactly as it is currently used, the design decision is predetermined in this case.

2.5 Open Systems

SDL supports the specification of (topologically) *open systems*, i.e. systems that have the ability to communicate with any environment through a well-defined external interface, as well as the specification of closed systems, i.e. systems without this ability. This language feature enhances the applicability of SDL to system development substantially, since in many cases, a system will be embedded into some pre-existing environment that is not specified nor developed with SDL. For instance, a protocol machine can be specified independently of concrete user modules or some underlying service provider.

As noted before, specifications of open systems may be considered incomplete from a formal point of view, since the environment is not known. From the line arguments w.r.t. language coverage, it follows that no formal semantics needs to be associated with these specifications. However, this would mean that in many cases of practical interest, specifications would not have a unique meaning, and therefore, the general benefits of using FDTs would not apply.

1. We distinguish between specified transitions (called "SDL transitions") and transitions of the underlying model (called "state transitions").
2. SDL statements are called "implicit" if they do not appear in the specification, but are executed according to the semantical model. For instance, the behaviour of delaying channels is captured by implicit statements.
3. In fact, an SDL statement execution is related to more than one state transition in some cases (e.g. in a decision, the evaluation of the question may need several steps). However, this concerns only SDL statements with effects that are local to the SDL process, therefore, the simplification made here does not lead to any inconsistencies.

To broaden the applicability of SDL as a *formal* description technique, it is essential that a precise meaning be associated also to specifications of systems that are open in the discussed sense. Openness can be understood as a particular type of incompleteness, so the question arises how a formal semantics could be assigned in this case. In order to exhibit the intended behaviour, an open system needs stimuli of the environment. Without these stimuli, only a (possibly empty) subset of the intended behaviour can occur. Note that the environment is not known at this point, so the actual stimuli are undetermined. A possible solution would be to define a formal semantics that always takes the influence of all possible environments on the behaviour of the specified open system into account. Technically, this can be achieved by taking, at any point of execution, all stimuli that are legal according to the interface definition into account, leading to a maximal set of execution sequences. This set is reduced as soon as the environment is (partially) determined, i.e. either by specifying a particular environment or by reducing the set of possible environments through behavioural constraints.
It should be noted that the existing SDL semantics does not address open systems explicitly, and, as a closer look reveals, also not adequately. The reason is that the possible stimuli of the environment are not taken into account. Therefore, the existing semantics covers only the behaviour among the system processes plus signals to the environment, but no behaviour triggered through incoming signals. Therefore, any reasoning that involves communication with the environment lacks a formal basis and must be done informally.

2.6 Compositionality

SDL specifications are composed of different kinds of components, including blocks, processes, services, procedures, channels and signal routes. For certain such compositions that are syntactically correct and satisfy further constraints, a formal semantics is defined. The current SDL semantics is defined for the entire specification, not for single components. This means that if, say, a certain process is to be analyzed independently, this is not supported by the semantics as this process has no meaning independently of its specification context. To avoid this problem, one could think of a specification reduced to this process only. However, this would be the specification of an open system, as this process has some environment with which it exchanges signals. Therefore, analysis of a single process can currently not be done on a semantical basis, but is pure intuition.

The problem would be alleviated by a semantics for open system specifications as discussed before. Also, a solution where a formal semantics is assigned to each of the listed components, and rules are defined how these fragments may be composed to yield the semantics of an entire specification, is perceivable. Yet a third option exists where open system specifications may be composed on a syntactical level. This would, however, require some language extensions, which are not under discussion at the moment.

2.7 Time

As it is possible in SDL to explicitly refer to real time and to use timer mechanisms, it is necessary to define the meaning of these constructs formally. Also, real time is used for performance and predictability analysis, therefore, the semantic model must include a precise notion of time.

There are many research activities on formal methods for real time computing. This demands for some foresight when designing a real time semantics for SDL (which is itself an evolving language especially regarding real time). It would be advantageous to have a semantic time model that supports a wide range of possible syntactical time extensions and validation methods. The question arises whether there exists a time model that encompasses at least some of the proposals currently discussed in the literature. Timing extensions of transition systems usually model possible system behaviour as timed state (or transition) sequences, which associate system states (or transitions) with time instants or time intervals. Possible timed sequences are specified, e.g., by restricting the times at which state transitions may occur with lower-bound and upper-bound requirements, or each state puts constraints on certain clock values, which allow the automaton to reside in a particular state only as long as these constraints are met. Some dual-language approaches even employ real time temporal logic to specify timing requirements.

There exist several validation techniques provided with these time models ranging from automata-theoretic reasoning or simulation mappings to model checking. The variety of techniques to cope with timeliness can of course be supplemented by queueing and scheduling theory or simulative techniques. However, the problem with these approaches lies in the construction of a separate mathematical or computer-executable model, which seems to be solvable by a suitable time semantics.

A semantic time model generally establishes some mapping between system executions and some kind of time axis. Thereby, time passage can be modeled either by explicit time passage transitions (similar to the current SDL approach), or time passage is strictly bound to normal state transitions, e.g., in terms of delays before or execution times for transition firing. The semantic model must be rich enough to allow for different degrees of determinism regarding the timing behaviour, i.e., in early phases of system development we expect a rather loose (resp. undefined) mapping between system behaviour and reference time while additional knowledge and design decisions will increasingly determine timing behaviour in later phases (so that mappings are completed).

3 OUTLINE OF A NEW FORMAL SDL SEMANTICS

Taking the discussion in Section 2 into account, we now give a coarse outline for a new formal SDL semantics. It is based on the definition of a transition system as the underlying mathematical model. Concurrency is modeled by interleaving, with SDL actions as atomic execution units. The semantics covers open systems, and is compositional. Time passage is coupled to state transitions.

3.1 Outline by example

A transition system is defined by a state set *St*, a set of start states $St_0 \subseteq St$, and a state transition relation δ*: St* → *Power (St)* that determines, for each state, the next state set. The formal SDL semantics defines, for each "complete" SDL specification, one such transition system with a structure that closely corresponds to the SDL specification structure. Intuitively, an SDL specification determines, at any point in execution, a set of *active objects*, i.e. instances of processes, services, procedures, delaying channels, and *passive objects*, i.e. instances of variables. We include, for reasons of intelligibility and maintainability, instances of block sets, blocks, and process sets.

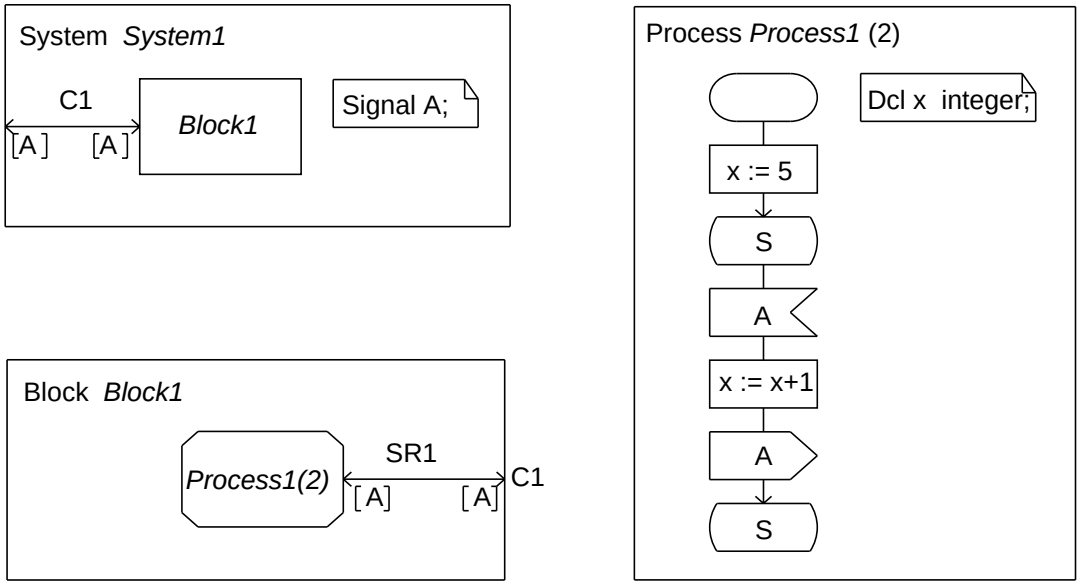

Figure 1: Example of an SDL system specification

An active object is characterized by its state set, the set of its start states, and its actions. Actions have a firing condition and effects that can be local as well as non-local, e.g., if signals are sent on non-delaying communication paths. The structure of a state set depends on the object type. Consider the example shown in Figure 1, where a system of two instances of *Process1* is specified. The state set of each of these instances is defined by:

$St_{Process} = S \times Var \times InQ \times Timers \times Expr \times Clock$, where

- *S* is the set of control states (*Process1*: start, S),- *Var* is the set of variable values (*Process1*: Integer),
- *InQ* is the set of input queue contents (*Process1*: $\{A\}^*$),
- *Timers* is the set of timer states (*Process1*: $\varnothing$),
- *Expr* is the set of values of predefined expressions (e.g., SELF),
- *Clock* is the set of process time values.

Thus, the state set of *System1* is defined by[4] $St = St_{Process1} \times St_{Process1}$. The set of start states of a process instance is determined by an action of the creating object instance, which in this example is an object of type process set

4. We simplify in this example by neglecting instances of block sets, blocks, process sets, and variables.

(not shown here). Similar actions are associated with with instances of block sets and blocks, yielding the start configuration of *System1*.

Process-definition
Process-name
Process1
Number-of-instances
initialNumberOfInstances = 2
maxNumberOfInstances = ∞
2
Variable-definition-**set**
varNames = *{x}*
Variable-name
x
Process-graph
major states = *{start, S}*
minor states = *{s1, s2}*
Process-start-node
Transition
Graph-node *
Task-node
Pre = *(obj.st.s = start)*
Eff = *(assign (obj,x,val(5)); obj.st.s := S)*
x := 5
Terminator
S
State-node-**set**
S
Input-node-**set**
Input-node
Pre = *(obj.st.s = S ∧ first(obj.st.inQ).sigType = A)*
Eff = *(obj.st.expr.sender := first(obj.st.inQ).sender;*
remove(obj.st.inQ); obj.st.s := s1)
Signal-identifier
A
Transition
Graph-node *
Task-node
Pre = *(obj.st.s = s1)*
Eff = *(assign (obj,x,val(x+1)); obj.st.s := s2)*
x := x+ 1
Output-node
sigDest
Pre = *(obj.st.s = s2)*
Eff = *(∀obj' ∈ receivers (A,obj.st.expr.self,**sigDest**).*
obj'.st.inQ := add(obj'.st.inQ,
(A,obj.st.expr.self)); obj.st.s := S)
A
Terminator
S

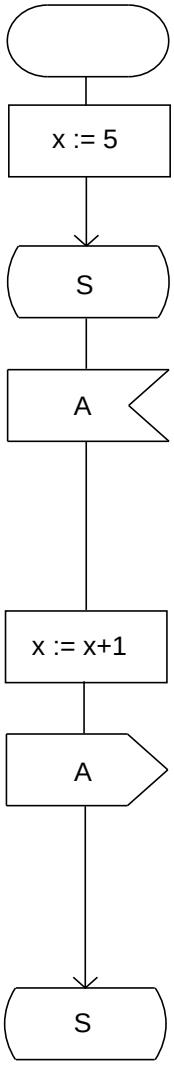

Figure 2: Excerpt of the Abstract Syntax Tree with Attributes: Process-definition

In order to define the state transition relation δ, we apply the formalism of attribute grammars. Note that for SDL, several grammars are defined in the standard: the concrete graphical grammar (used in Figure 1), the concrete textual grammar, and the abstract grammar. The SDL standard defines how to transform an SDL specification into the

corresponding abstract syntax representation. Since this representation is more suitable than the concrete form, we use it as a starting point for assigning a formal semantics to SDL specifications.

In Figure 2, an excerpt of the abstract syntax tree (AST) of *System1* is listed. With some of its nodes, we associate attributes that can be formalized by evaluation rules of a suitable attribute grammar. For instance, the attribute ***varNames*** denotes the set of variables that are declared in *Process1*. Using these attributes, further attributes called ***Pre*** and ***Eff*** are derived for designated nodes, such that each pair of such attributes defines an action. For instance, the attribute ***Pre*** associated with Input-node, when interpreted in the context of an active object *obj*, expresses that the control state *s* of *obj*'s state component *st* is *S*, that the input queue *inQ* is not empty, and that the signal type of the first signal of the *inQ* is *A*. The attribute ***Eff***, when interpreted in the context of *obj*, expresses that the predefined expression *sender* is assigned a certain value, that the first signal is removed from *inQ*, and that the next control state is *s1*.

With these preparations, we can now define the next state relation δ of *System1* as follows:

$\forall st' \in St.\ \delta(st') = \{st'' \in St \mid \exists a \in \mathbf{A}_{obj1} \cup \mathbf{A}_{obj2}.\ (st' \in a.\mathbf{Pre} \wedge (st',st'') \in a.\mathbf{Eff})\}$, where

- $\mathbf{A}_{obj} = \{(\mathbf{Pre},\mathbf{Eff}) \mid \exists$ node $\in$ nodeSet(*Process1*). ***Pre*** = node.***Pre*** and ***Eff*** = node.***Eff***$\}$)

Informally, for each state *st'* of *St*, the next states are the states *st"* that are reachable from *st'* by some action *a* of an active object. Thereby, the attributes ***Pre*** and ***Eff*** become relations on the state set *St* when interpreted in the context of an active object: ***Pre*** determines the firing condition, i.e. the constraint under which the action may be selected for execution. ***Eff*** defines the effects of the action on the system state. By writing nodeSet(*Process1*), we refer to certain nodes of the abstract syntax tree where actions that belong to each instance of *Process1* are defined.

3.2 Open systems

Next, we outline how the semantics of open systems can be defined. As noted before (see Section 2.5), a solution would be to define a formal semantics that always takes the influence of all possible environments on the behaviour of the specified open system into account. Technically, this can be expressed by adding an environment action to the transition system, as shown in Figure 3: every time this action is executed, some signal from the environment enters the system and is appended to the input queues of some set of receivers, according to the addressing scheme used in the output-statement.

System-definition
inSignalSet
signalDest
Pre *= ()* /* environment action */
Eff *= (* $\exists$*inSig* $\in$ ***inSignalSet***. $\exists$*sigDest* $\in$ ***signalDest***.
$\forall$*obj' $\in$ receivers(inSig,obj.st.env,sigDest). obj'.st.inQ := append(obj'.st.inQ, (sig,obj.st.env)))*
System-name
System1

Figure 3: Excerpt of the Abstract Syntax Tree with Attributes: System-definition

3.3 Generalization

The ideas illustrated in the example can be generalized, arriving at the dynamic semantics of an arbitrary SDL specification which is given by a transition system $T = (St, St_0, \delta)$, where

- $St = \cup_{objectSet \in ObjectSet} (\times_{obj \in objectSet} obj.St)$
  with *objectSet* being a finite set of objects that may exist simultaneously under the static constraints of the specification, and *obj.St* being the state set of the object *obj*. Note that only some of the states of *St* may actually be reachable according to the dynamic semantics of the specification, in particular, only some of the object sets may exist under the static constraints.

- St_0 = set of initial states of the start object (e.g., “system”, or “processSet”)
- δ: *St* → *Power (St)*
  $\forall st' \in St.\ \delta(st') := \{st'' \in St \mid \exists obj \in objectSet(st').\ \exists a \in \boldsymbol{A}_{obj}.\ (st' \in a.\boldsymbol{Pre} \wedge (st',st'') \in a.\boldsymbol{Eff})\}$
- $\boldsymbol{A}_{obj}$ = {(***Pre***,***Eff***) | ∃ node ∈ nodeSet(node(*obj.identifier*),includeNodes(*obj.type*), delimitingNodes(*obj.type*)). ***Pre*** = node.***Pre*** and ***Eff*** = node.***Eff***})

Generally speaking, the transition system associated with an SDL specification is obtained from the definition of objects in a compositional way: global state set, start states, and state transition relation are defined in terms of the states and actions of objects. Since all objects are defined in a uniform way, the formal semantics is modular, which improves its maintainability. Also, the way actions are defined supports the generation of code. Finally, incoming signals from the environment can be incorporated in the same uniform way, by introducing an additional action.

3.4 Time

To model time, we introduce a conceptual clock for all active objects. Note that these clocks are used to model the laws of nature, they should not be confused with physical clocks. Each clock keeps track of the local object time, it is advanced when the object executes actions, or when the object is waiting and other actions occur. Due to the interleaving semantics, clock values may differ, and the maximum clock value is perceived as the global time.

Process-graph
  Transition
    Task-node
    minMaxET = (2,5)
    Pre, ***Eff***
      x := 5
  State-node-**set**
    S

    Input-node
    minMaxET = (2,9)
    Pre = *(action selected for execution ∧*
      global time is advanced a minimum distance ∧ ...)
    Eff = *(...; obj.st.clock := obj.st.clock + executionTime;*
      clocks of waiting processes := obj.st.clock;
      for all waiting processes: select action for execution)
      Signal-identifier
        A
      Transition
        Task-node
        minMaxET = (2,5)
        Pre, ***Eff***
          x := x+ 1
        Output-node
        minMaxET = (4,5)
        Pre, Eff
          A
      Terminator
        S

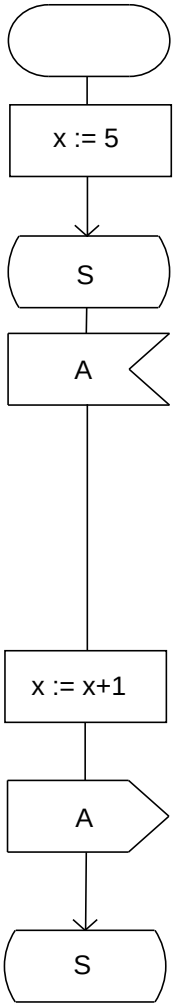

Figure 4: Excerpt of the Abstract Syntax Tree with Attributes: Real-time

In SDL, it is currently possible to refer to the current time, and to use timer mechanisms. In the example shown in Figure 4, we go one step further in assuming that an execution time interval can be associated with each action. The intended meaning is that if an action is selected for execution, it will consume an arbitrary amount of time in this interval. To model the timing behaviour, we include the selected action of an active object into the object state. Furthermore, we extend the firing condition of an action by requiring that the action has been selected for execution, and that by executing this action, the global time is advanced a minimum distance. The latter condition is necessary to avoid effects such as signals arriving in the past. The additional effects of an action are the advancement of the local clock by the execution time of the action, the advancement of local clocks of all waiting active objects to the global time, and the selection of executable actions including an execution time within the specified time interval for all waiting objects (in that order).

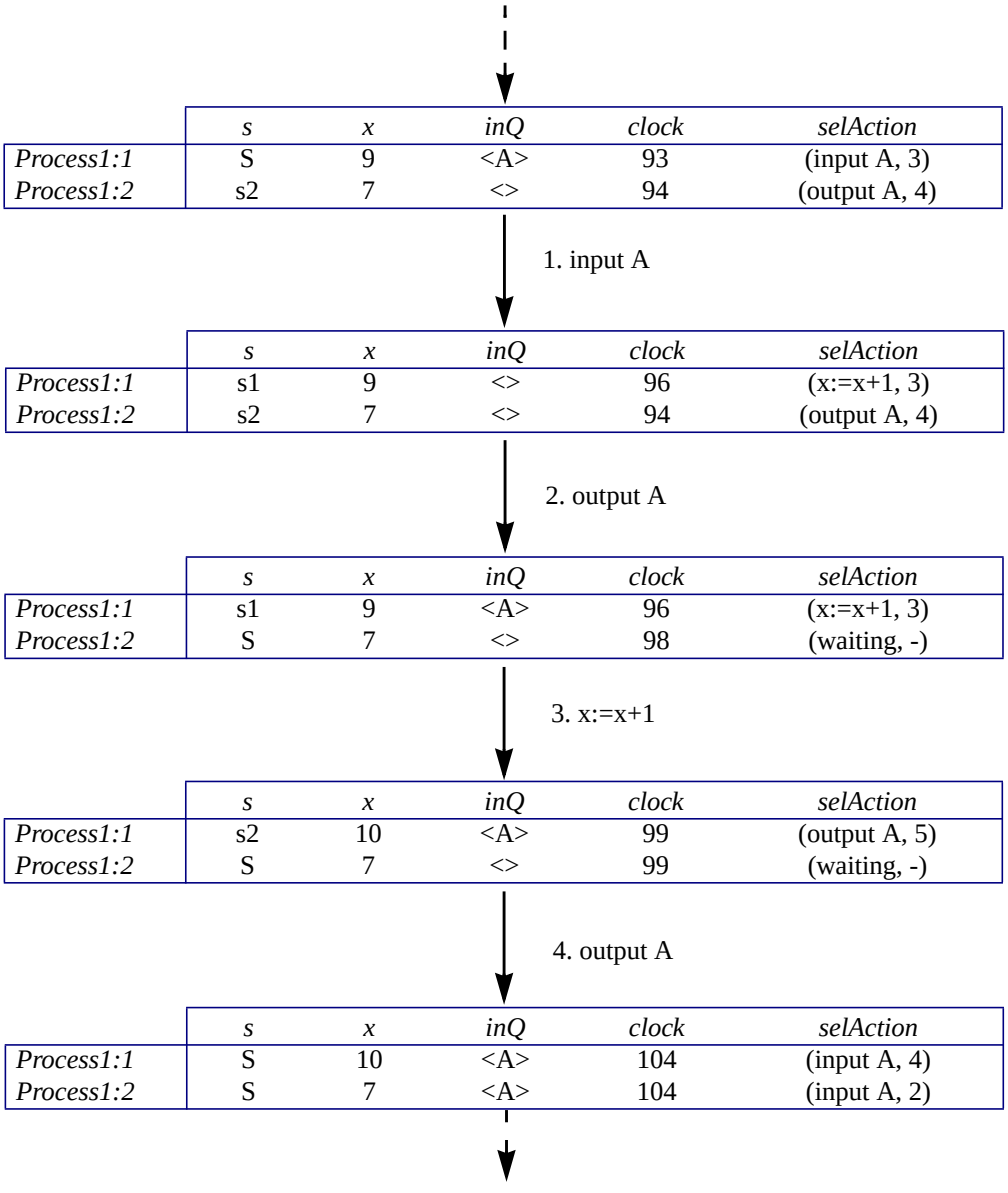

Figure 5: Excerpt of a reachability graph: Real-time

Figure 5 shows an excerpt of a labeled reachability graph of *System1*. Each node represents a reachable state, which is characterized by the control states, the variable values, the input queues, the clock values, and the selected

action of the process instances. In the first state, the action “input A” will be selected, because its execution will advance the global time by a minimum distance, i.e. to 96. When “input A” has been executed (with the effect that the signal A is removed from the input queue, and that the new control state is s1), the local clock is advanced, and a new action together with an execution time within the specified interal is selected. The next action to be executed is “output A”, which brings *Process1:2* into a waiting state, since there are no fireable actions. For this reason, its local clock has to be advanced after the next action of *Process1:1*, and also after the action “output A”, before a further action can be selected. It is important that this selection takes place after the advancement of the clock, because at local time 99, no signal A has been in the queue of *Process1:2*.

4 CURRENT STATUS

Currently, the proposed new formal SDL semantics is being worked out in detail. Formalization is based on the abstract SDL grammar, which is defined in the Z.100 Recommendations. The abstract grammar is extended by evaluation rules, yielding an attribute grammar. Attributes include actions, determined by firing condition and effects, which are defined for about 20 different node types. This defines, for each “complete” SDL specification, an attributed abstract syntax tree, from which the transition system is then directly obtained. We expect the resulting semantics to be extremely concise.

The outline presented in this paper leaves enough room for improvements concerning, e.g., the notation that is used. In [2,3], a formal dynamic semantics for what is termed Basic SDL is provided in terms of an Abstract State Machine (ASM). It should be investigated how this approach is related to the semantics proposed in this paper, whether it is applicable to full SDL, and what aspects of real-time can be modeled. A major advantage of ASMs is the existence of a tool environment supporting the execution of ASM models.

5 REFERENCES

[1] E. Brinksma, J.-P. Katoen, R. Langerak, D. Latella: *Partial Order Models for Quantitative Extensions of LOTOS*, Computer Networks and ISDN Systems, Special Issue on “Trends in Formal Description Techniques”, 1998 (forthcoming)

[2] U. Glässer: *ASM Semantics of SDL: Concepts, Methods, Tools* (this volume)

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

[4] R. Gotzhein, F. Rößler, P. Schaible: *Towards a Formal SDL-Semantics - Design Choices and Outline*, ITU-T SG10 TD 79-E, Geneva, March 24 - April 1, 1998

[5] ITU-T Recommendation Z.100, *Specification and Description Language (SDL)*, International Telecommunications Union, Geneva, 03/93

[6] ITU-T Recommendation Z.100, Annex F.1, *SDL formal definition: Introduction*, International Telecommunications Union, Geneva, 03/93

[7] ITU-T Recommendation Z.100, Annex F.2, *SDL formal definition: Static semantics*, International Telecommunications Union, Geneva, 03/93

[8] ITU-T Recommendation Z.100, Annex F.3, *SDL formal definition: Dynamic Semantics*, International Telecommunications Union, Geneva, 03/93