Timed Extensions for SDL

Marius Bozga[a], Susanne Graf[a], Laurent Mounier[a], Iulian Ober[b], Jean-Luc Roux[b] and Daniel Vincent[c]

[a]VERIMAG, Centre Equation, 2 avenue de Vignate, F-38610 Gières,
e-mail: Marius.Bozga@imag.fr

[b]Telelogic Technologies Toulouse, 150 rue Nicolas Vauquelin, F-31106 Toulouse Cedex,
e-mail: Iulian.Ober@telelogic.com

[c]France-Telecom R&D, DTL, 2 avenue Pierre Marzin, F-22307 Lannion Cedex,
e-mail: Daniel.Vincent@rd.francetelecom.fr

Abstract: In this paper we propose some extensions necessary to enable the specification and description language SDL to become an appropriate formalism for the design of real-time and embedded systems. The extensions we envisage concern both roles of SDL: First, in order to make SDL a real-time *specification* language, allowing to correctly simulate and verify real-time specifications, we propose a set of *annotations* to express in a flexible way assumptions and assertions on timing issues such as execution durations, communication delays, or periodicity of external inputs. Second, in order to make SDL a real-time *design* language, several useful real-time programming concepts are missing. In particular we propose to extend the basic SDL timer mechanism by introducing new primitives such as cyclic timers, interruptive timers, and access to timer value.

All these extensions relies on a clear and powerful time semantics for SDL, which extends the current one, and which is based on *timed automata with urgencies*.

Keywords: SDL, Time Semantics, Timed Automata, Urgencies.

1. INTRODUCTION

The ITU–T Specification and Description Language (SDL, [11]) is increasingly used in the development of real-time and embedded systems. For example many recent telecommunication protocols (such as Rmtp-II [20], Pgm [19] or Mtp-2 [12]) integrate such real-time features in their architecture, and these non-functional aspects are essential in the expected behavior of the application.

This kind of systems imposes particular constraints on the development language and SDL is a suitable choice in many respects: it is formal, it is supported by powerful development environments integrating advanced facilities (like simulation, model checking, test generation, code generation, etc.), and thus it can cover several phases of the software development, ranging from analysis to implementation and on-target deployment.

It appears however that several important needs for a real-time systems developer are not covered by SDL. These problems range from pure *programming issues*, like the lack of useful primitives commonly available in real-time operating systems, to *specification issues*, like the difficulty to describe in a appropriate way the assumptions under which the system is supposed to be executed. Clearly, the needs are not the same for both uses of the langage, and, in many cases, the *programming side* has been given priority in the supporting tools to the detriment of the *specification side*.

Several proposals already exist to extend SDL with real-time features. We can mention for example the works carried out on performance evaluation ([9,15], [17], [14]), on schedulability analysis ([4]), or on real-time requirements ([13]). In this paper we are more concerned with the use of SDL as a specification language for real-time systems and its application for formal validation. In particular, one of the important question we address is what kind of real-time features should be modeled in SDL and at which level of abstraction.

The simplest use of time which is frequent in communication protocols, is the use of timeouts (those value is often meaningless) in order to avoid infinite waiting and deadlocks. The time semantics of SDL, together with the fact that timeouts are specified via a signal in the message queue of the process, corresponds exactly to this use: no guarantee can be given when the signal arrives and is dealt with, but it is after a finite time. Nevertheless, when SDL is used as a programming language, it is often done with much more restricted assumptions on the possible time behavior in mind, and, if they are correct, the implemented system will behave as expected. As such assumptions are not (and cannot be) expressed explicitly, the specification cannot be validated: the verification using the standard SDL time progress may invalidate even apparently time independent safety properties.

A typical workaround used for obtaining a convenient result at simulation time consists in using in one hand timers to force minimal waiting time, and, in the other hand, a very restricted interpretation of SDL standard semantics, allowing time progress only if the system is not active. This “synchrony hypothesis” is in general as unrealistic as the standard semantics. The right assumption would be that certain tasks will be executed timely, whereas for others this cannot be guaranteed and the correctness of the system must be verified even if they arrive later than expected.

The solution we propose to reconciliate these two extreme choices relies on a more flexible time semantics for SDL, based on timed automata with urgencies [5]. In particular, urgencies give a very abstract mean to express *assumptions* on the environment and on the underlying execution system, such as action durations, communication delays, or time constraints on external inputs. From the user point of view, all these “non functional” extensions are offered in a uniform way by means of *annotations* on the SDL specification.

The propositions presented in this paper are the results of the Interval IST project and preliminary work of its partners. The aim of Interval is to take into account real-time requirements during the whole development process of real-time systems and to define appropriate extensions to the languages SDL, MSC and TTCN.

The rest of the paper is organized as follows: in section 2, we give an overview on what are the real-time features we consider as missing concerning both programming and specification aspects, in section 3 we propose extensions allowing to improve SDL as a real-time specification language and in section 4 we propose necessary programming concepts. All new concepts are illustrated by examples illustrating their use and proposed syntax. Finally, in section 5 we draw some conclusions and perspectives.

2. REAL-TIME SDL: WHAT IS MISSING ?

SDL has the double aim of being on one hand a high-level *specification* formalism, meaning that it must abstract from certain implementation details, and on the other hand a *programming* formalism from which direct code generation is possible. These two roles of the language seem sometimes conflicting, as the needs at the different levels are not always the same.

It is important that SDL can fully play this double role of being an implementation and a specification language, and all information needed for both uses of SDL must be expressible, but also in such a way that these two concerns are clearly separated. This feature is particularly crucial when dealing with real-time systems, in which non-functional elements need to be taken into account even in the early stages of the design.

We summarize here the main difficulties currently arising when trying to use SDL for the design and validation of real-time systems.

2.1. Real-time semantics

First of all, the semantics of SDL, as presented in Z.100, is very abstract in the sense that it allows to make no assumptions on time progress: actions take an indeterminate time to execute, and a process may stay an indeterminate amount of time in a certain state before taking the next fireable transition. This notion of time that is external, unrelated to the SDL system, is realistic for code generation, in the sense that any actual implementation of the system conforms to it. However, for simulation and verification, this total absence of controllability of time is not satisfactory: timer extents do not have any significance besides defining minimal bounds, any timer that gets in a queue may stay there for any amount of time, with the consequence that hardly any real-time property holds on models based on this abstract semantics.

Furthermore, any representation of the simulation graph of an SDL system representing its semantics (which is the starting point for simulation and verification) accounts for all possible combinations of execution times, timer expirations and timer consumptions. Since no control over time progress is possible, many unrealistic executions will be considered in this graph. A simulator that would use the semantics of time as described in Z.100, would not be able to make any assumption on the way time progresses. As a result, the simulator would not guarantee elementary properties like *when a timer expires, it is put into the queue of the concerned process in a* reasonable *amount of time* (whatever the notion of reasonable is), or even worse, *when two timers are set at the same time (for example as two consecutive assignments), the timer with the lower delay will be consumed first.*

In practice, existing simulation and verification tools have foreseen means for limited control over time progress. However, the control over time they propose is in general quite restricted and moreover these annotations are tool dependent whereas they are totally part of the specification in the sense that they describe assumptions on the system environment. The fact that designers and design languages neglect a clear description of relevant properties of the environment in which the system should be executed, is an important source of errors.

2.2. Non-functional annotations

The development of a complex real-time protocol usually needs to consider several preliminary stages, during which some abstract or uncomplete descriptions are produced. In order to properly validate (and document) these early designs, general assumptions on both the "environment" of the system and on its "non-functional" behavior have to be taken into account. For example, such assumptions concern the expected duration of some internal task (this task being either informal or fully specified), the periodicity of some inputs triggered by the environment, or even the expected behavior of the communication channels used within the system (i.e., these channels are reliable or not, what are the expected communication times, etc.).

Of course, some of these assumptions can already be partly included in the specification, either directly in SDL (e.g., using timers for explicit waiting) or using some internal formalisms offered by the verification tools (like the GOAL language [3] proposed in *Object*Geode to specify external observers). However, none of these two solutions is satisfactory: the first one leads to a specification in which external and non-functional assumptions do not appear as such, and this is obviously not desirable for code generation (these timers need not to be implemented), whereas the second one is restricted to a particular tool. Our objective is to provide a more suitable framework, based on standardized *annotations* on SDL specifications and compatible with the real-time semantics we propose.

2.3. Extended real-time primitives

Clearly, SDL has several characteristics that are attractive for real-time systems designers: asynchronous communication is a first class language feature, a specification is organized in a logical hierarchy that can be mapped in many ways to different physical configurations of software modules (and SDL code generators usually provide this feature), external code may be called from SDL, making it possible to use system libraries directly in SDL.

Several mechanisms that are commonly employed in real-time systems could however be natively implemented in the language. In particular, SDL timers are rather limited: the only available primitives are **set** and **reset** operations, the **active** function (which allows to determine if a given timer is running or not), and timeouts are always transmitted in the form of asynchronous signals.

2.4. Deployment information

Z.100 asserts that the agents composing a system are executed in a truly parallel environment. In the context of the very weak time semantics of SDL (only minimal waiting time can be enforced, and any action or message transmission takes either zero or an arbitrary amount of time), this simplifying assumption is possible, because any mapping on a set of processors and any notion of atomicity will lead to the same functional behavior.

However, the introduction of a more precise notion of time introduces also global constraints, so that different degrees of atomicity or different mappings on processors will lead to different time behaviors, and — if the functional behavior depends on real time constraints — even to different functional behaviors. An obvious example is the fact that, if a set of processes are executed interleaved, their execution times must be added, otherwise the global execution time will be the maximum of the individual execution times.

One must obviously be very careful by trying to make assertions on global execution times using no or very abstract assumptions on the architecture on which the system is executed. However, it is not always necessary to introduce much knowledge about the architecture:

- First of all, there exist time dependent properties which are not architecture dependent: for example, often the safety of a protocol may depend on the relative values of a set of timers, the expiration of which are used as implicit signal between processes. This is a common use of timers which cannot be expressed in the present SDL time semantics and still does not need any architecture indication.

- In a system where time is consumed either in communication or in requests to external systems (like a distant data base or anywhere else in the environment), the execution times will not depend on the mapping of processes on processors whenever the execution time of all activities consuming a negligible amount of time can be safely simplified to zero. Also in the case where time is consumed within the system, but within a single process, analysis of execution time is still possible in the same way.

- In the case where time is consumed in several parts of the system, it would be sufficient to indicate which parts of the system are executed in parallel and which not. The distinction between block agents, process agents and sub-agent gives some possibility to indicate such an architecture information.

Going one step further, scheduling policies also can influence the properties of the system in critical hard real-time systems.

3. EXTENDING SPECIFICATION ASPECTS

As we mentioned in the previous section, what is missing in SDL to improve real-time systems *specification* is both a flexible time semantics together with more facilities to express some “non functional” parts of the system or its environment. Our proposal is to solve these two problems uniformly – from the user point of view – by offering the possibility to *annotate* the specification in a standardized way. In particular, we propose to distinguish between two kinds of annotations:

- *assumptions*, which express *a priori* knowledge or hypotheses about the environment or the underlying execution system (system architecture, scheduling policy, etc). The use of assumptions is twofold. First, they might be necessary during the verification phase to verify specific properties, which could not be verified otherwise. Second, they might be used during the code generation phase, both to guide some implementation choices or to add specific code in order to check their correctness at run-time, or during the test phase.

- *assertions*, which express expected (local) properties about the system components. Such properties have to be proved on the specification, during the verification phase, possibly taking into account some of the assumptions.

The annotations we propose concern respectively, control over time progress by means of *urgencies*, *durations* and *periodicity* of actions, and flexible *channel* specifications.

3.1. Urgencies

A very abstract – and still very powerful – manner of constraining the time environment of a system is by means of transitions urgencies [5]. Three types of urgency assumptions (*eager*, *lazy* and *delayable*) are enough to control the progress of time with respect to the progress of the system:

- **eager** transitions have priority over time progress: they are assumed to be executed "as soon as possible"; if, in a simulation state, there is some **eager** transition enabled, *time does not progress* unless all eager transitions are taken or disabled.

- **lazy** transitions have the same priority as time progress: no assumption is made concerning the time at which they occur; enabled **lazy** transitions do not inhibit time progress in any simulation state.

- **delayable** transitions are a combination of eager and lazy transitions: they are supposed to be executed in some interval of time in which they are enabled. A delayable transition usually has an enabling condition depending on time, such as $\mathbf{now} \leq x$ or $\mathbf{now} - x \leq y$ (where x and y are numerical values). Then, *time may progress* in the simulation state *until* $\mathbf{now} = x$ (or $\mathbf{now} - x = y$) and, when this time point is reached the transition becomes urgent.

  Notice that the attribute "delayable" is not primitive: a delayable transition a guard $\mathbf{now} < x$ (for instance) can always been replaced by two transitions, a lazy one with the same guard and an eager one with guard $\mathbf{now} = x$. Furthermore, as long as in SDL explicit time guards (others than timeouts) are not commonly used, delayable transitions are also rarely useful explicitly.

Expressed in terms of these urgencies, the semantics of time in Z-100 considers all transitions as lazy: time progress is not constraint at all, whatever transitions are enabled. Nevertheless, most SDL tools use an eager semantics: transitions are fired as soon as they are enabled without letting time progress. It appears in practice that none of these two extreme interpretations of time progress in isolation allows to obtain satisfactory models of real-time systems. It is often appropriate to mix these two extreme views of time progress:

- one would like to consider some of the inputs as lazy (which is the standard point of view). When such a transition is enabled, the system can choose to react immediately or to wait. In the case of an *external* input laziness denotes the absence of knowledge about the possible arrival time of the input; for *internal* inputs laziness can be used to consider internal action durations or internal propagation delays as unconstraint or unknown.

- On the other hand, one would like to consider some of the inputs as eager to express that the system cannot ignore them when they are enabled and must react immediately without waiting. This can be used to guarantee an immediate response to *critical* events such as timeout expirations or other prioritary inputs.

  Some care must be taken here since one can easily write system specifications in which time will never progress because at least one eager transition is always enabled (this is called a *Zeno behavior*). This is particularly problematic in the case where all transitions are considered eager.

Default choices, depending on the type of the system and on the type of transitions, can be envisaged in order to provide an user-friendly way of specifying how urgencies are associated with system transitions.

Example 3.1 *A generic situation is presented in figure 1. The informal action* open *must be executed in the interval defined by timers* t_early *and* t_late*, as soon as the external signal* open *is received. Thus, if the* open *signal arrives too early, it must be saved; if it does not arrive in time, the action is executed anyway at the expiration date of* t_late*. In order to obtain a realistic time behavior of such a system, timeout consumptions have to be considered as eager: the action* open *is assumed to be executed at time* t_late *at latest. In order to specify all possible behaviors, the (external)* open *signal input must be considered as lazy. Considering it as eager prevents the timer* t_late *to expire since this transition is always enabled in state* wait *(in absence of an environment process that sends only a limited amount of* open *signals). In the case where the* open *signal is sent from within the system, the consuming transition might be considered as eager (meaning that it has to be executed as soon as possible within its allowance interval) and one may want to* verify *if the signal arrives always in the given interval depending on the time constraints of the other parts of the system.*

Timer semantics

Timers are the most used primitive to control and to observe the time progress. Nevertheless, the semantics of timer expirations must be slightly revisited. The behavior of a timer, as described by Z.100, can be sketched by the automaton given in figure 2. Basically, it switches between *inactive* and *active* states depending on the set and reset actions performed on it. When active, once the expiration time is reached, it will *expire* and send a timeout signal to the corresponding process instance. Unfortunately, according to Z-100 this expire transition is lazy, which means that one cannot make any assumption on the maximal time elapsed since its last setting. In other words, nothing can prevent the automaton from remaining in the active state after the expiration time.

We propose to introduce a more reliable timer concept ensuring the output of the timeout signals *exactly* at expiration time. This can be done simply by considering the implicit expiration transition as eager. Note that this cannot be captured by the urgency annotation described before since it relies on an implicit semantic choice which cannot be explicited at the specification level. Also, this is not too restrictive: it still allows the situation in which the timeout is really consumed at a later point of time as the consumption time depends on the urgency of the timeout transition.

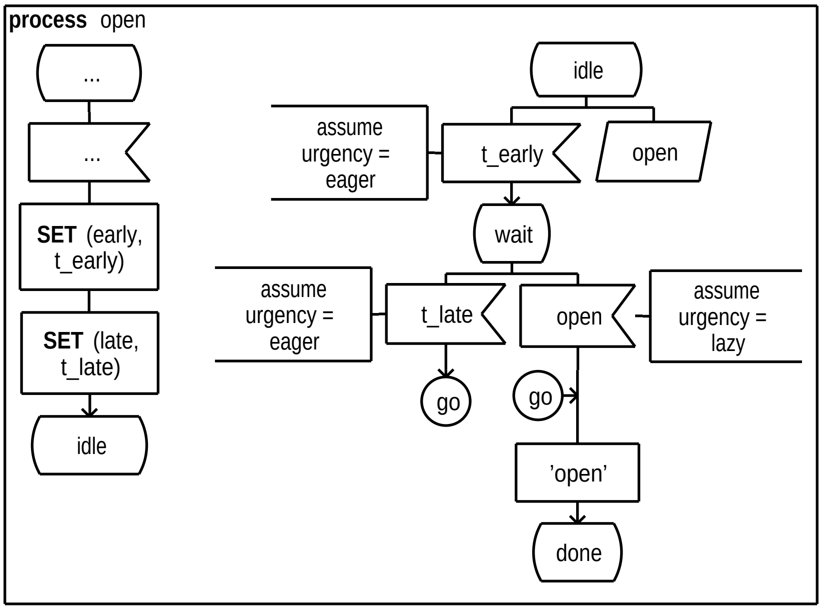

Figure 1. Urgency assumptions - an example.

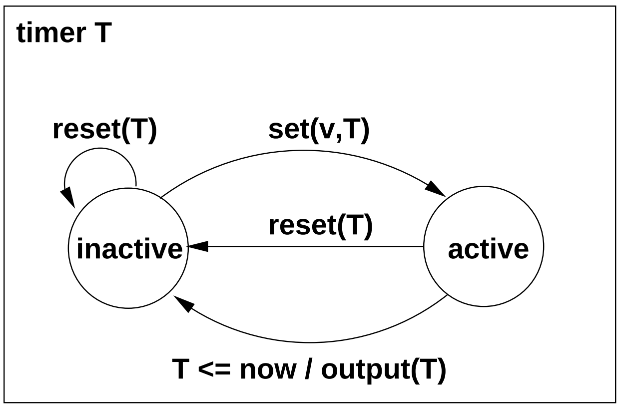

Figure 2. Timer behavior.

Example 3.2 *A second generic process is illustrated in figure 3. The informal actions* action_1*,* action_2*,* action_3 *must be executed* precisely *at moments, respectively* T0-d1*,* T0-d2*,* T0-d3*, where* T0 *is given as parameter to the process. Let us assume first that the informal actions correspond to external commands which need just to be initiated by the process e.g, actioning some external devices indicated. Unfortunately, even if the* SDL *description of the process seems intuitive and concise, based on the standard semantics of* SDL*, no assumption can be made about the time at which the actions will be executed. Considering the timeout transitions as lazy, the actions are executed* not earlier than *the required moment. The eagerness of the timeout consumption transitions only expresses that actions are executed at the moment at which the timer expirations are available, which is again* not earlier than *the required moment. Only assuming both timeout delivery* and *timeout consumption as eager obliges the system to react* exactly *at the required moment.*

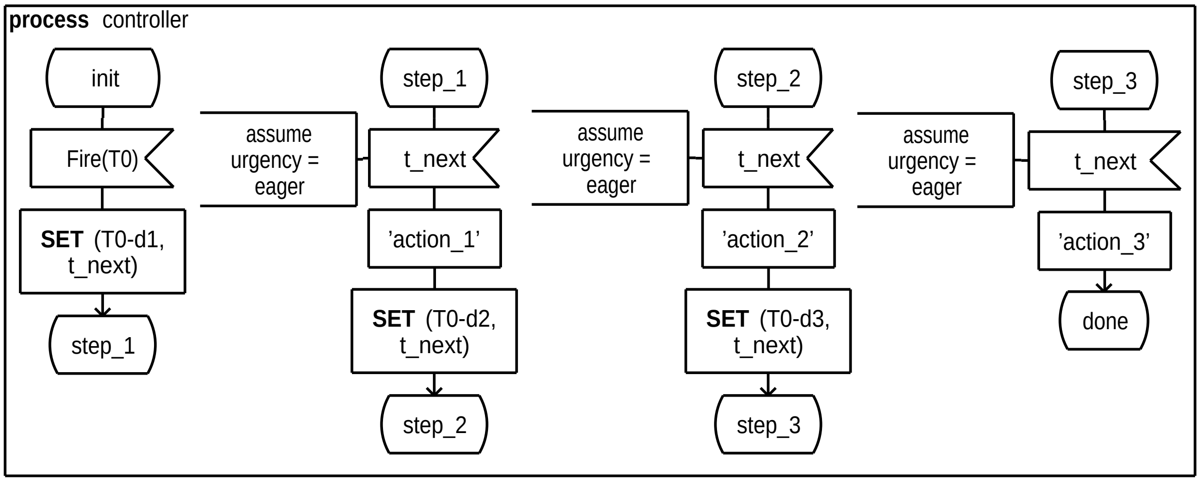

Figure 3. timeout urgencies - an example.

3.2. Durations

SDL does not foresee the possibility to impose or to assume any quantitative restriction on the duration that an action may take to be executed or how long a process may stay idle in a state before executing one of the enabled actions. Also, concerning the amount of time that a signal may need to travel through a channel, only two cases can be distinguished: either 0 time or an arbitrary amount of time. Nevertheless, in real-time designs, such execution times may not only influence the performance, but also the functional behavior of the system.

Currently, it is possible in SDL to describe minimal and maximal durations by means of explicit waiting using timeouts and a notion of "invalid state" to mark the executions taking more than the maximal execution time as "uninteresting". This is used frequently in SDL specifications, but it is a bad solution as it is cumbersome and it uses a programming construct in order to indicate assumptions about the environment. Such a specification can not be used directly for automatic code generation, since a code generator is not able to detect the nature of these timeouts automatically.

We suggest to explicitly indicate such durations using predefined annotations. For instance, we propose to use either interval constraints (e.g. delay=[9.0,11.0]) or mean-values plus jitter (e.g., delay=[10.0±5%]) attached to the corresponding action or channel.

Moreover, such annotations could be enriched with probability laws. This extension is mandatory for performance evaluation, but still not sufficient, as we also need a model of available computation resources. Note that from a functional verification point of view probabilities are not necessary since all the behaviors have to be analyzed independently of their probability: verification wants to ensure absence of errors and not just "low probability".

3.3. Periodicity

Many telecommunication applications are expected to cope with large streams of data arriving with high and continuous rates (e.g, multimedia services). In practice, components of such applications are designed to fit in particular environments, able to deliver multiple inputs at given frequencies.

Therefore, periodicity of inputs tends to be an important feature that has to be expressed at the specification level. Similar to duration, we propose to annotate external inputs with interval constraints (or mean-values plus jitter) describing the expected period (where applicable). Such annotations are not only mandatory for verification but they also clearly improve the readability of the specification since they allow to describe in a synthetic manner the relevant characteristics of the assumed environment.

Example 3.3

A typical producer/consumer system is illustrated in figure 4. The producer reacts to external request *signals and sends* data *signals to the* consumer*. When the* consumer *receives* data*, it sends an acknowledgment back to the* producer*. Several annotations are used here. First, we assume that* request *signals come from the environment at the given rate of one signal every 10 or 11 time units. Also, both data production and consumption times are supposed to be not negligible, the former being between 5 and 7 and the second between 4 and 5 time units.*

Notice that, on the contrary to any description using timers or the global time ***now****, here it is obvious to see that a production cycle can be greater than the period of* request *signals. This may be considered as problematic, as there exist scenarios in which the* request *signals accumulate indefinitely in the input queue of the* producer*. At this point, the use of probabilities may become important to decide if the design is acceptable or not.*

3.4. Channel reliability

In SDL, only reliable FIFO channels exist as primitive concept. In its use as a specification language, however, a channel is often considered to be part of the environment of the implemented system and may represent an abstraction of a whole network. Many protocols are supposed to implement reliable communication through unreliable channels or networks. For verification it is therefore necessary to consider the situations where channels lose or reorder messages.

Thus, by flexible channel specification we mean typically that messages can be lost, reordered or delayed to some extent, leading to a well-defined set of channel types. It is possible to describe any of these channel types by means of additional SDL processes, but

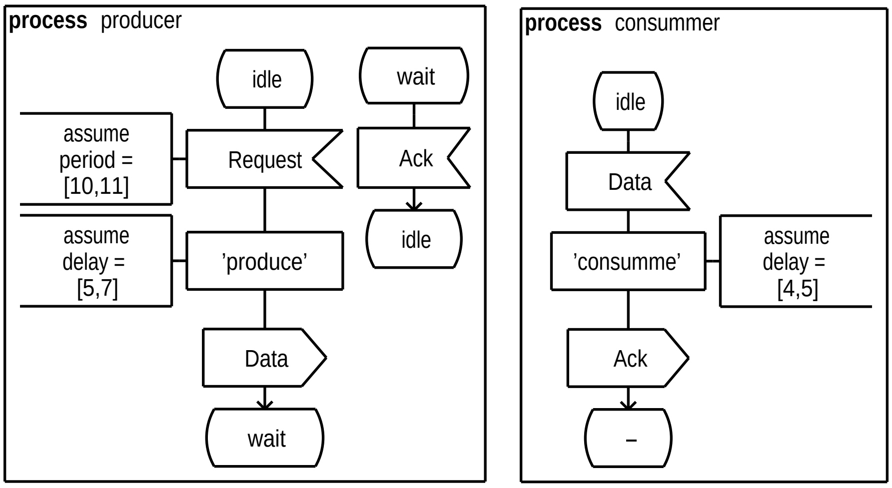

Figure 4. Delay and periodicity assumptions.

it is not necessarily desirable to do so, for several reasons. First of all, these processes serve only for simulation and not for implementation. Also, in some cases it may be problematic as the transported signals do not carry the pid of the original sender anymore. Finally, the fact to have a predefined set of channel attributes is an advantage for simulation and verification tools as this knowledge can be directly exploited by appropriate techniques and lead to more efficient algorithms (notice for example that an interesting set of properties is decidable for finite state machines communicating through lossy channels whereas they are undecidable in case of communication through reliable channels [2,1]).

We propose annotations on channels which allow to specify a propagation delay in a similar way as execution times and distinguish between fully reliable (which do not lose messages) and unreliable channels (where arbitrary losses are possible), and between ordered and unordered ones. Here again, for performance evaluation purposes, it is possible to extend these annotations to include a probability law which specifies a rate of message losses or a degree of reordering.

Example 3.4

Figure 5 gives an example of use of annotations to denote propagation delays and reliability of channels. Notice that the default option for a channel is to be ordered.

4. EXTENDING PROGRAMMING ASPECTS

We make here two concrete proposals to improve SDL as a *programming* language for real-time systems. These proposals concern respectively the extension of the timer concept and the introduction of standardized packages to provide at the SDL level useful synchronization and atomicity primitives.

4.1. Extension of timer concepts

Timers play a central role in SDL to control and observe time progress. They are however limited with respect to what is commonly offered in real-time systems, and we

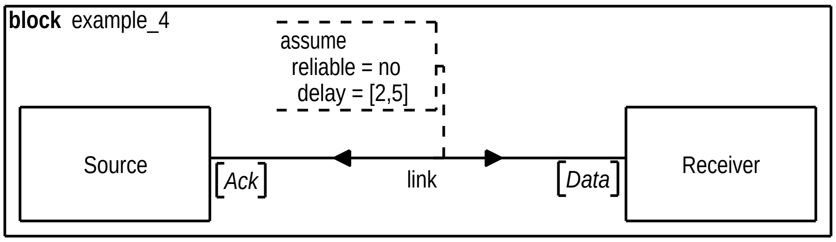

Figure 5. Channel assumptions.

propose to extend them in several directions.

Interruptive timers and signals

SDL timeouts are always received in the form of asynchronous messages. For general-purpose time dependent code this is usually fine, but it is difficult to write real emergency procedures using asynchronous timeouts. To ensure that a piece of code is executed immediately, or in a specified interval, after the expiration of a timeout, the SDL programmer must first make sure that the corresponding agent (process) is idle when the timeout message is received, otherwise, the agent may consume the timeout message from the input queue only when it finishes its current job, which may be too late. This obliges to artificially restructure the system, whereas in the implementation this is useless because of the existence of interrupts. Therefore, SDL needs a notion of emergency timer, whose expiration is taken immediately into account by the receiving agent. Emergency actions which interrupt the normal execution of an agent were already introduced in SDL'2000, with the advent of exceptions. All we need is a link between the exception mechanism and system time.

We propose to define interruptive timers using an optional attribute (called *interruptive*). The behavior of interruptive timers will rely on an extension of the exception mechanism already existing in SDL. More precisely, when an interruptive timer expires, instead of sending a timeout signal via the input port of the process, it will raise a timeout-exception in the concerned process. The handling of this exception is left to the user. However, special care is required to clarify what happens if interruptive timer wants to interrupt a transition which is required atomic In particular, when an interruptive timer expires in a service, while another service (within the same process) is running and executes a time-consuming job.

Example 4.1 *In figure 6 we illustrate the use of an interruptive timer. When set, it receives as parameter the maximal allowed execution time* T*. Then, the process enters a loop executing some time-consuming job* "decode"*. Two cases are possible: either the entire job is finished in time, or the maximally allowed time is reached before, when the process is processing a* "decode"*. In the second case, an interruptive timer* alarm *is needed in order to break the normal execution flow, i.e, to abort the execution of* decode *and to stop the process immediately.*

Note that for sake of simplicity we use the same notation for normal timeouts and *interruptive timeouts. Nevertheless, the meaning is quite different: whereas the former denotes a normal* SDL *transition from a state, the latter is a shorthand notation for an exception handling at this state (and implicitly* in all implicit states and actions within the state scope*).*

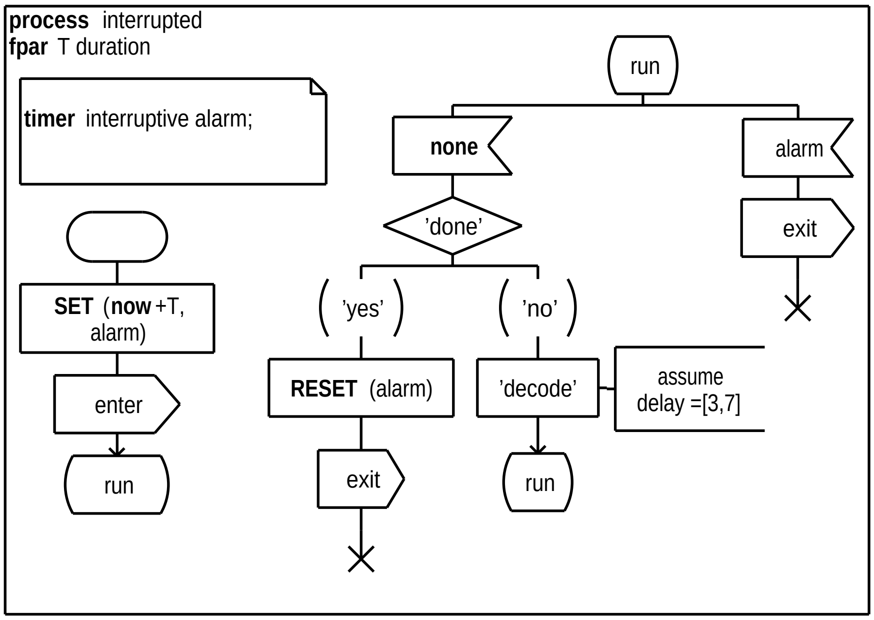

Figure 6. Interruptive timer example.

Timer consultation

The second extension we propose concerns the ability to consult the expiration time (which is of predefined sort **Time**) assigned to a timer. In order to do so, our proposal is simply to add a predefined **value** construct on timers (similar to the existing **active** construct).

Such a primitive reduces the current distinction made in SDL between timers and other variables. In particular, the value of the timer can be passed via communication signals from a process to another, used to compute the remaining duration until its expiration, or used to set another timer depending on it, etc.

Obviously, the same result can be obtained using variables of sort **Time** instead of – or in addition to – timers. But the use of an explicit timer concept (with a set of well defined primitives) makes the specification more readable and the mapping into an implementation easier.

Cyclic timers

Finally, the third extension concerns cyclic timers. The idea is to eliminate the current limitation in SDL, where timers are one-shot and have to be explicitly reset in order to model a periodic behavior. This could be done by simply considering an optional timer attribute (called **cyclic**) which fixes the nature of the timer at its declaration. When a cyclic timer is explicitly set in the specification, its period is computed (by subtracting the value of **now** from the expiration time). After that, at the expiration time (when the corresponding timeout signal is sent) this timer will be implicitly set again using the period and the value of **now** at expiration. This will continue until either an explicit reset or set occurs (the later restarts the whole behavior, possibly with a new period). Finally, note that interruptive and cyclic attributes can be safely combined (i.e., the same timer can be both cyclic and interruptive).

4.2. Atomicity

There are no atomicity and synchronization primitives foreseen in SDL, the idea being that these are implementation details which should not be mentioned at SDL level. The fact is that in implementation oriented SDL descriptions, they are necessary and several SDL users have expressed the need for native SDL constructs for synchronization [10], especially to achieve atomicity and mutual exclusion. The reason is that such constructs are often used in the specification and the implementation of real-time systems such as those developed with SDL.

The current practice in SDL is to use calls to external code (e.g. calls to OS primitives) in order to achieve these functionalities. This approach has at least the following obvious inconvenients: the SDL specification, which is supposed to be high-level, becomes unnecessary configuration and platform dependent, and the external code cannot be handled properly by simulation and verification tools.

The need stated above can be addressed without making first-order extensions to SDL. Synchronization behavior can be expressed in terms of existing primitives of SDL, such as asynchronous signal exchange and remote procedures. We propose the use of (standard) libraries for this purpose, as it is the case for other languages.

Example 4.2 *For example, a semaphore may be specified in* SDL *as a process type exporting two (empty) procedures* P *and* V*, implementing the usual operations on semaphores. The specification of such a semaphore type is shown in figure 7. The only prerequisite for this implementation to work is that the atomicity of* P *and* V *are preserved for a same instance of semaphore. This prerequisite is ensured by the execution semantics of* SDL*. Moreover, concurrent wait operations* P *which arrive after the semaphore is already blocked on a wait are implicitly sequentialized by the save operation from the* busy *state.*

5. CONCLUSION AND PERSPECTIVES

We have defined a number of real-time extensions of SDL in order to obtain a specification and modeling language really adequate for real-time systems based on general ideas and reflections we have proposed in [8]. The proposed extensions include

- a more flexible time-progress semantics for SDL, including as particular cases the time progress semantics of Z100 and the time-progress semantics used in verification tools
- powerful and flexible annotations allowing to express non-functional aspects: as well very abstract ones necessary already at an early design stage as well as more concrete ones useful for code generation, code annotation and performance analysis.

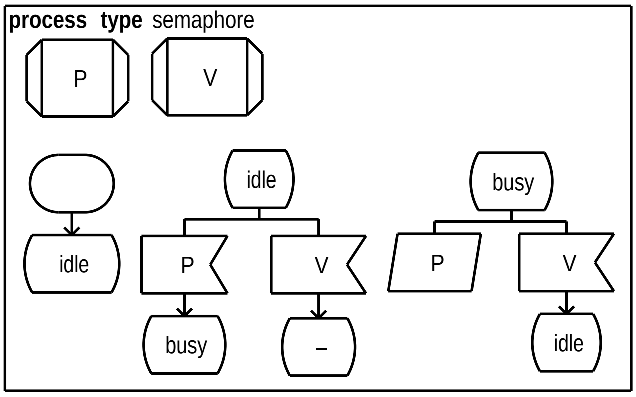

Figure 7. Semaphore example.

- some primitives which can be expressed in SDL as it is but which allow to make specifications simpler, easier to understand and which allow a better separation of implementation oriented and specification oriented features

- implementation oriented features like emergency timeouts which are mandatory for the use of SDL as a real-time modeling language

All these extensions are based on a unified and sound semantic framework which can be integrated smoothly into the existing semantics of SDL. In particular, almost all of them allow flexible restrictions of this very non-deterministic semantics.

These extensions have been submitted as a first draft to the ITU-T (Study Group 10) within the framework of Question 7 ("Time expressiveness and performance annotations on ITU-T modeling languages").

We have already started to extend existing SDL tools in order to deal with these extensions and they already proved their usefulness in real applications:

- we have implemented a translation of SDL with our proposed time extensions into communicating timed automata with urgencies which are the input language of the IF toolset [7,6] which has been developed at VERIMAG for the purpose of prototyping timed extensions of SDL-like languages. There exists a model-checker based on extensions of the Kronos-tool [21] allowing to verify time extended models with reasonable restrictions (e.g. durations are open intervals)

- we have implemented these extensions also in the *Object*Geode simulator [16] with similar restrictions

- in both cases we have good results in both what we can express and what analyses we can perform on annotated models. We are presently rewriting the RMTP-II protocol [20] using these annotations.

The main approaches which need to be compared with ours are QSDL[9,15] and UML related real time extensions (such as ROOM[18] and planned real-time standard extensions):

- the foreseen real-time extensions of UML are all of a very syntactic nature. It is not clear if there will be a semantic time model at all. However, it is interesting to note that non-functional annotations concerning Quality of Service (such as throughput of channels, execution times,...) are planned and can take likewise the form of *requirements* and *assumptions*.
- QSDL (Queuing SDL) is a performance analysis oriented extension of SDL using a resource mapping and annotations of tasks in terms of "computation power" from which execution and waiting times are computed depending on some scheduling policy which can be user defined. The underlying semantics considers all non annotated transitions as taking zero time and all transitions are considered urgent, respectively delayable if they may take variable amount of time. In so far, this framework is compatible with ours, but is exclusively performance analysis oriented features.

Perspectives

The proposed extensions are not completely satisfactory from the point of view of a user. Some more extensions may well appear to be useful. There is a strong need for a powerful notion of deployment diagram, allowing to define the mapping of processes to resources, scheduling policies and high-level QoS annotations. The annotations will then have 3 different sources:

- some of them may be user defined, especially at an early design stage
- some of them will be generated from information extracted from such a deployment diagram (which goes far beyond the resource mapping of the QSDL proposal)
- some will be obtained from analysis results of other parts of the system, especially in a compositional verification approach (which is the only one able to deal with large specifications).

REFERENCES

1. P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In A .Hu and M. Vardi, editors, *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, pages 305–318. Springer, June 1998.
2. P. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Journal of Information and Computation*, 1996.
3. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *Proceedings of SDL FORUM'95*. Elsevier, 1995.
4. J.M. Alvarez, M. Diaz, L.M. Llopis, E. Pimentel, and J.M. Troya. Integrating Schedulability Analysis and SDL in an Object-Oriented Methodology for Embedded Real-Time Systems. In R. Dsoulli, G.v. Bochmann, and Y. Lahav, editors, *Proceedings of SDL-FORUM'99 (Montreal, Canada)*, pages 241–256. Elsevier, June 1999.
5. S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.

6. M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 (Toulouse, France)*, volume 1708 of *LNCS*, pages 307–327. Springer, September 1999.
7. M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF: An Intermediate Representation for SDL and its Applications. In R. Dssouli, G. Bochmann, and Y. Lahav, editors, *Proceedings of SDL FORUM'99 (Montreal, Canada)*, pages 423–440. Elsevier, June 1999.
8. M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for Real-Time: What is Missing ? In *Proceedings of SAM'00: 2nd Workshop on SDL and MSC (Grenoble, France)*, pages 108–122. IMAG, June 2000.
9. M. Diefenbruch, E. Heck, J. Hintelmann, and B. Müller-Clostermann. Performance evaluation of SDL systems adjunct by queueing models. In R. Braek and A. Sarma, editors, *Proceedings of SDL Formu'95*. Elsevier Science B.V., 1995.
10. Interval Consortium. Requirement Analysis Report. Technical Report D11, Interval Deliverable, October 2000.
11. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, 1994.
12. ITU-T. Specification of signalling system no. 7 – mtp level 2 test specification. Technical Report Q-781, International Telecommunication Union – Standardization Sector, 1999.
13. S. Leue. Specifying Real-Time Requirements for SDL Specifications – A Temporal Logic-Based Approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95*. Chapmann & Hall, 1995.
14. M. Malek. PerfSDL: Interface to Protocol Performance Analysis by means of Simulation. In R. Dsoulli, G.v. Bochmann, , and Y. Lahav, editors, *Proceedings of SDL-FORUM'99 (Montreal, Canada)*, pages 441–455, 1999.
15. A. Mitschele-Thiel and B. Müller-Clostermann. Performance Engineering of SDL/MSC Systems. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in SDL and MSC (Erlangen, Germany)*, February 1998.
16. I. Ober, B. Coulette, and A. Kerbrat. Timed SDL Simulation and Verification - Extending SDL with Timed Automata Concepts. Technical report, Telelogic Technologies Toulouse, 2000.
17. J.-L. Roux. SDL Performance Analysis with ObjectGeode. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in SDL and MSC (Erlangen, Germany)*, February 1998.
18. B. Selic, G. Gullekson, P. Ward, B. Selic, and J. McGee. *Real-Time Object-Oriented Modeling*. Wiley Proffesional Computing, 1994.
19. T. Speakman, D. Farinacci, S. Lin, and A. Tweely. PGM Reliable Transport Protocol Specification. Internet draft, IETF, 1999.
20. B. Whetten, M. Basavaiah, S. Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. The Reliable Multicast Transport Protocol, version 2 (RMTP-II). Internet draft, IETF, 1998.
21. S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.