SDL for Real-Time: What Is Missing?[1]

Marius Bozga, Susanne Graf *Alain Kerbrat* *Daniel Vincent*
Laurent Mounier *Iulian Ober*
Verimag, Grenoble Telelogic, Toulouse France Telecom

Abstract. In this paper we give an overview on the main weaknesses of SDL for the development of real-time systems, both on the programming and on the specification side. In particular, the SDL semantics proposed in Z.100 asserts that time is totally external to the system specification, and does not allow any control over time progress, which is however essential for verification. To solve this problem, we propose to adopt a semantic framework for SDL based on timed automata with urgencies which is a simple and intuitive underlying model, allowing to express most of the real-time primitives. Finally, we illustrate our proposal by means of a small but realistic example, and we show how it is related to several other proposals.

keywords: real-time systems, timed semantics, performance evaluation, semantic profiles, verification

1 Introduction

The ITU–T Specification and Description Language (SDL, [1]) is increasingly used in the development of real-time and embedded systems. This kind of systems impose particular demands 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, auto-coding), and it supports many phases of software development, ranging from analysis to implementation and on-target deployment.

In this paper we review the needs of a real-time systems developer that are not covered, for various reasons, by SDL. The issues that we examine are heterogeneous, ranging from pure *programming issues*, like the difficulty to specify real timeout emergency procedures or to program atomic transactions, to *high-level modeling* issues, like the difficulty to model time non-deterministic system components or to use the standard formal semantics for simulation and verification. For most issues we strain to give solutions, although sometimes this only means that we favor one alternative among a set of incompatible, equally justified choices.

We propose a semantic setting for time in SDL that allows a flexible specification of *timing requirements* and *timing knowledge* about the system. With our proposal one can capture very general forms of conditions on the duration of actions or the duration between two events. We propose analysis methods that work on top of this semantic

[1] This work is funded by the French National Telecommunications Research Network (RNRT), `http://www.telecom.gouv.fr`

framework, and by which we can verify general timing properties of a system, such as the minimal/maximal time between two particular events.

To support our claim, we illustrate on a small (but realistic) example some of the main weaknesses of SDL regarding the specification of useful real-time features, and we show how our semantic setting allows to solve these problems. Finally, we briefly compare our work with other proposals [10, 16].

2 Types of Problems and General Proposals

2.1 Types of Problems

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. The problems that we identify in this paper refer to SDL either as a specification language, or as a programming language. The problems on each side are different because the needs on each side are different too.

On the specification side, we further have two kinds of problems: *expressivity* problems and *usability* problems.

1. An *expressivity* problem is the impossibility of SDL to capture meaningful information about a system (like, for example, the execution time boundaries of a piece of code).
2. *Usability* problems relate to the way an SDL model is used in analysis and early design: the designer must be able to simulate the system or to formally verify certain properties. A usability problem of the SDL semantics makes it practically or theoretically impossible to construct the global state graph of a system (graph that is used by simulation or verification tools).

2.2 Semantic Profiles for SDL

The semantics of SDL, as presented in [1], is rather crafted for code generation than for simulation and verification. Z.100 maintains that each action takes an indeterminate time to execute, and that a process stays 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 actual implementations of the system conform to it. However, for simulation and verification, this time semantics is not satisfactory: timer extents do not have any significance besides consisting minimal bounds, any timer that gets in a queue may stay there for any amount of time, whereas the verification of a property often depends on the fact that in the real system the upper bounds are *not* indefinite.

Any rigorous attempt to construct the simulation graph of an SDL system (which is the starting point for simulation and verification) must account for all possible combinations of execution times, timer expirations and timer consumptions. Since no control

over time progress is possible, many undesirable executions might be obtained during this exhaustive simulation. We have here a *usability* problem as characterized in section 2.

In practice, existing simulation and verification tools make simplifying assumptions on execution and idle times. The usual convention is that actions take 0 time to execute, and that the system executes immediately whatever it can execute. This option is justified by the fact that it generates the highest degree of determinism, thus reducing the state space by an important factor (and in fact, rendering the system practically analyzable). This is not the only point where the SDL semantics raises usability problems for simulation and verification, as we will see in the next sections.

We face here two alternative definitions of the semantics that are mutually exclusive and equally justified (one by the needs of code generators, one by the needs of simulators and verification tools).

This dichotomy cannot be surpassed by a single SDL semantics. The solution we propose is to adopt multiple semantic *profiles*. This idea is not new: the UML community is on the way to defining several profiles for UML [2], each one fitting for a particular application domain (real time, electronic commerce, etc). In the case of SDL, profiles would not correspond to different application domains but rather to different usages of an SDL model: code generation, schedulability analysis, simulation, performance analysis, model checking, test generation etc.

A semantic profile would define a semantics that is particularly suitable for a certain type of manipulation of an SDL model. A semantic profile for code generation, for example, would support real parallelism between the agents composing a system, while a semantic profile for simulation and verification would propose quasi-parallel execution of agents, that is, interleaving of transitions or transition actions.

If we accept the need for different semantic profiles, it follows that the definition of profiles should be parameterized. Parametric semantic profiles allow to reuse the common part of two different profiles, and to outline only the differences. For example, a semantic profile for simulation and verification could have a parameter which determines whether whole transitions are atomic, or whether only the SDL statements (OUTPUT, TASK, SET, RESET, etc) should be atomic (the fact that TAU SDL Validator [17] supports such a parameter suggests that it is useful). There is no need to have two complete profiles for these two cases, since most of the semantics is the same in the two cases. A parameter would be a simple and clean solution.

Semantic profiles allow the SDL standard to follow the path that already undertaken by SDL tools which are highly parameterized. The parameters used by tools such as *Object*GEODE [18] and TAU [17] represent in fact small variations in the semantics used by the tools.

3 What is Missing on the Programming Side?

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 from SDL.

There are however several mechanisms, often employed in real-time systems, which should be natively implemented in the language. We enumerate some of them here.

3.1 Interruptive Timers

SDL offers native mechanisms for writing time dependent code: one can consult the system clock (through the implicit variable **now**), set timers, wait for a timer to expire or receive an asynchronous message on expiration.

SDL timer 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 timeout emergency procedures using timers. To ensure that a piece of code is executed "*immediately*" 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 asynchronous timer message from the message queue only when it finishes its current job, which may be too late.

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.

Our proposal goes towards the introduction of the notion of interruptive timer in the language. An interruptive timer raises an exception instead of sending an asynchronous message when it expires. With interruptive timers, one can easily set up real timeout emergency procedures.

3.2 Atomic Code Sequences and Synchronization

Atomicity and mutual exclusion can be achieved by inserting system calls into the SDL code. However, these are patterns that are very common in real-time systems, and SDL should benefit from native constructs for expressing atomicity and mutual exclusion.

Additionally, inserting system calls in SDL has a severe drawback: as we mentioned in the beginning of Section 3, one advantage of SDL is that it can be mapped to different physical software configurations. System calls for obtaining mutual exclusion are different when agents are mapped to threads and when they are mapped to processes. This means that the SDL code must differ from configuration to configuration, which is a regression.

With native SDL constructs for atomicity and mutual exclusion, a code generator could generate the right synchronization, rollback or deadlock protection code in every possible mapping. Moreover, atomicity and mutual exclusion would be taken into account in simulation (which is not the case when using system calls), and deadlocks or other kind of errors that they may introduce could be detected earlier.

The same discussion stays valid for general purpose synchronization code. Some forms of synchronization between SDL agents may be achieved only through external system calls. There too, native SDL constructs would be be helpful.

4 What Is Missing on the Specification Side?

On the specification side things are more critical. As mentioned in the introduction, the role of SDL in the system development process is twofold: on one hand it is a *specification* language that must be capable to abstract away certain implementation details while still capturing an accurate image of the system under development, on the other hand it is a *description* language that must be able to express an implementation down to the last details. These two roles of the language are sometimes conflicting, and in many cases the *description* side has been given priority, to the detriment of high-level specification.

4.1 Control over Time Progress

The problem used as an example in the beginning of Section 2.2 is an important usability problem in itself. A simulator that would use the semantics of time as described in Z.100, would have no control over the way time progresses. As a result, the simulator would not guarantee elementary properties like:

1. when a timer expires, it is treated in a "reasonable" amount of time (whatever the notion of reasonable one might use).
2. when two timers are set at the same time, the timer with the lower delay will be consumed first.

This will lead to the exploration of a number of undesirable execution paths that can never actually happen in the system implementation.

A semantic profile for simulation must give the simulator some control over time progress. Existing simulation tools do this, by assuming that actions take 0 time to execute, and that time never progresses while the system has something to execute.

The usefulness of this extreme control over time progress in simulation is limited. There are cases when time progress needs to be controlled in more flexible ways:

- to specify that in a certain state, an unlimited amount of time may pass, even though the system has something to execute;
- to specify that in a state, a bounded amount of time may pass regardless of whether there is something to execute or not. In this case, there is a number of consequent problems as to the specification of the amount of time (fixed or with lower and upper bounds; specified statically or dynamically).

We propose a concrete solution to this problem in Section 5.

4.2 Assumptions on Execution Times

There is also an expressivity problem related to the usability problem of section 4.1: since the standard semantics of SDL assumes that an indeterminate amount of time may pass while the system is in a state or while it executes an action, there are no means to specify the execution times of (sequences of) actions.

Such information may be meaningful in simulation or in verification. The well functioning of the system may depend on the assumptions on execution times.

Currently, in order to introduce assumptions on minimal execution times, the user is forced to use timers and to introduce explicit waiting. For maximal execution times, the user must also introduce timers and additional invalid states that will have to be considered as unreachable when the state graph is built. Thus, in order to express high-level specifications, one needs to use programming features.

There exists already several approaches to introduce execution time assumptions in SDL specifications. The *Object*GEODE Simulator [16] uses a syntactic extension by which one can associate an execution time (interval) to an action. [10] uses a more elaborate approach in which execution times are dynamically calculated by means of queuing machines, so that they are depending on the amount of work and on the charge of the system. Other related works are the ones dedicated to the introduction of scheduling policies within SDL, like [5].

We will not introduce here new SDL extensions for expressing execution times. Instead, we introduce a semantic framework that allows a simulator to control the progress of time (Section 5) and we show how existing approaches for expressing execution times ([16, 10]) are complementary to our semantic framework, with benefits in terms of analysis power.

4.3 Atomicity of Transition Elements

The lack of programming constructs for expressing atomicity, mutual exclusion and synchronization was outlined in Section 3.2. The same problem may be characterized as an expressivity problem of SDL as a high-level specification language.

Besides that, the lack of a notion of atomicity poses usability problems. Z.100 [1] asserts that the agents composing a system are executed in a real parallel environment. In order to work, a simulator has to assume a certain degree of atomicity. Existing SDL simulation and verification tools make simplifying assumptions: that statements are atomic, or that entire transitions are atomic, or that sequences of statements that take 0 time to execute are atomic.

The place for such assumptions would ideally be a semantic profile for simulation and verification.

4.4 Flexible Channel Specifications

SDL defines channels as reliable means for transporting messages: a channel never loses messages. Additionally, a channel may either be non-delayable (i.e. messages arrive instantaneously at the other end) or with non-specified delays (but keeping the order of the conveyed messages).

These attributes are insufficient for characterizing real communication channels. For example, SDL is used to describe flow control protocols such as the alternating bit protocol from the OSI stack. Such protocols are built upon the assumption that certain channels are unreliable, and it is their mission to make them reliable through software.

If the assumptions on channels cannot be marked in SDL, the resulting protocol cannot be used for simulation: the simulator will never cover the parts handling signal loss.

In practice, when one needs to model a channel which loses messages or which delays messages, one has to explicitly describe the behavior of the channel in SDL(with an SDL process, for instance). This approach has several drawbacks:

- once the behavior of the channel is specified, all messages will arrive at destination with a wrong **sender** PID
- the channel description must be replicated over and over again for every lossy channel in the system (note that a generic Process Type cannot be used, because the channel description depends on the types of the conveyed signals, differing from channel to channel)
- dynamic creation of timers is needed in order to transport an indefinite number of messages at once on a delayable channel.

A simple solution to this problem is to allow the user to specify in SDL:

1. whether a channel loses messages or not, and the loss probability
2. upper and lower time bounds for the delays applied to the message conveyed by a channel, as well as the probability laws of the delays

More complicated solutions which take into account the type and size of a message can be imagined. Again, the ideal place for such extensions would be a semantic profile for simulation, verification and performance analysis.

5 Timed SDL Semantics Based on Transition Urgencies

In this section we introduce a semantic framework that can be used in connection with SDL to solve the problems of controlling time progress in simulation, problems described in Section 4.1. Basically, we introduce a set of constructs for controlling simulation time progress, for which we have analysis methods allowing to derive interesting timing information (such as the minimal/maximal time span between two events) and thus to verify timing properties of SDL systems.

The framework presented here is not directly a solution to the problems described in Sections 3 and 4. Instead, it may constitute the *underlying semantics* for many temporized extensions of SDL (such as [10, 16]), which solve the above mentioned problems, and which are closer to the abstraction level of SDL. Therefore, the constructs we introduce below are not meant to be used as such by SDL modelers.

The constructs identified here are inspired from timed automata with urgencies, a high-level formalism for modeling temporal properties of reactive systems. For a thorough understanding of the semantics behind these constructs, the reader is referred to [3, 4] (timed automata), and [6] (timed automata with urgencies).

As stated in Section 4.1, in order a semantics to be usable in simulation and verification, the simulator needs to have control over system time. In SDL, system time is represented by the value of the implicit variable **now**. Our idea is the following: we

consider that time may only progress while the system stays in a "simulaton state", and time does not progress while the simulator executes a "simulation transition" (that is, **now** is not modified). Note that we talk about "simulation states" and "simulation transitions", which may differ from SDL states and transitions: for example, in the case that a complete SDL transition is not considered as atomic but an individual statement is, there will be a new "simulation state" and a "simulation transition" for each individual SDL statement.

Moreover, the progress of time in a simulation state is controlled by the transitions that can be triggered next. We identify three categories of transition *urgencies*:

1. **eager** transitions, which have priority over time progress. If in a simulator state there is an **eager** transition enabled, *time cannot progress* unless the transition (or another enabled transition) is taken.
2. **lazy** transitions, which have the same priority as time progress. An enabled **lazy** transition does not inhibit the progress of time in the simulation state. Therefore, *time may progress with an indefinite amount*, if the other enabled transitions allow it too.
3. **delayable** transitions, which have priority over time progress only when time progress would disable them. Time progress may disable a transition if the transition has an enabling condition depending on time (i.e. on the value of **now**). Therefore, a delayable transition will usually have an enabling condition depending on now, such as $\mathbf{now} \leq x$ or $\mathbf{now} - x \leq y$ (where x and y may be integer variables or constants). Then, *time may progress* in the simulation state *until* $\mathbf{now} = x$ (or $\mathbf{now} - x = y$).

With this semantics, the simulator can control the progress of the system time by identifying the urgency of the simulator transitions enabled in a certain state.

The source of this information on urgencies differ from case to case, depending on the concrete timed extensions introduced at SDL level. We can imagine an extension of SDL in which the user is allowed to directly associate urgencies to transitions, like in the example in Section 6. But urgency information can also be derived from other kinds of timed annotations, as we will see in Section 7.

We have implemented transition urgencies in IF [7, 8], a specification language developed at VERIMAG for prototyping semantic variations of the constructs of an SDL-like language. We have also implemented the extensions in the *Object*GEODE Simulator [15]. In both cases we have good results in terms of both what we can express with them and what analyses we can perform on annotated models.

However, such extensions are not very close to the level of abstraction of SDL, and modelers may find it difficult to produce the urgency annotations and the related information. As we mentioned already, our extensions are rather thought to be the semantic basis for more user-level constructs, such as those introduced in [10, 16]. Section 7 is dedicated to showing how such user-level extensions are projected on our semantic framework, and what are the advantages of using this framework.

6 Example: the Bounded Retransmission Protocol

We illustrate here on a simple example some of the specification problems of SDL that have been identified in this paper, and we show how they can be solved using the semantic framework proposed in the previous section.

6.1 Specification of the protocol

The example we propose is the so-called "Bounded Retransmission Protocol" (BRP), which provides a file transfer service through an unreliable medium between two entities, a `Transmitter` and a `Receiver`. More precisely, each file is split into several packets and each packet is transmitted in sequence using the well-known alternating bit protocol. However, in case of packet loss, only a *bounded* number of retransmission are performed, and thus the file delivery is not guaranteed. In this situation, both entities should abort the current file transfer, and proceed with the next file. This protocol has been used as a running example for several verification tools[14, 9, 12], and we consider here a simple version mainly focussed on its timing behaviour, which is in general not treated in the above approaches.

This protocol (figure 1) is composed of a `Transmitter` and a `Receiver` process, whose description in SDL can be found in Figure 1.

The `Transmitter` first waits for a transfer request issued by the environment (`PUT(p)`, where `p` is the number of packets). When a transfer request is issued, it starts transmitting packets `(m,b)` one by one, where `m` indicates whether the packet is a `first`, `middle` or `last` element of the file, and `b` is the alternating bit. After each transmission of a packet, the `Transmitter` starts a timer `t_repeat` and waits either for an acknowledgment issued by the receiver, or for the expiration of `t_repeat`. If a correct acknowledgment is received, it resets `t_repeat` and proceeds to the next packet, unless it was the last one. However, if `t_repeat` expires, the *same* packet is resent up to `max_retry` attempts (`t_repeat` being restarted after each resent). If none of these transmission succeeds (no correct ack is received), the `Transmitter` aborts the current file transfer and reports the failure to its upper layer. After a transfer abortion the `Transmitter` starts a timer `t_abort` and waits for its expiration before processing the next file.

The `Receiver` continuously waits for packet receptions. When a `first` packet is received, it initializes its alternating bit, starts a timer `r_abort` and sends back an acknowledgment to the `Transmitter`. Each subsequent packet is acknowledged (according to the "alternating bit" policy), and the timer `r_abort` is restarted upon each reception of a *new* packet. When a `last` packet is received, the `Receiver` considers that the entire file has been correctly transmitted: it delivers it to its upper layer (`GET(p)`), stops its timer, and waits for a new file. However, if the timer `r_abort` expires, the `Receiver` assumes that the next packet has been repeatedly lost that the transfer has been aborted by the `Transmitter`. It informs its upper layer (`ABORT`), and waits for a new file.

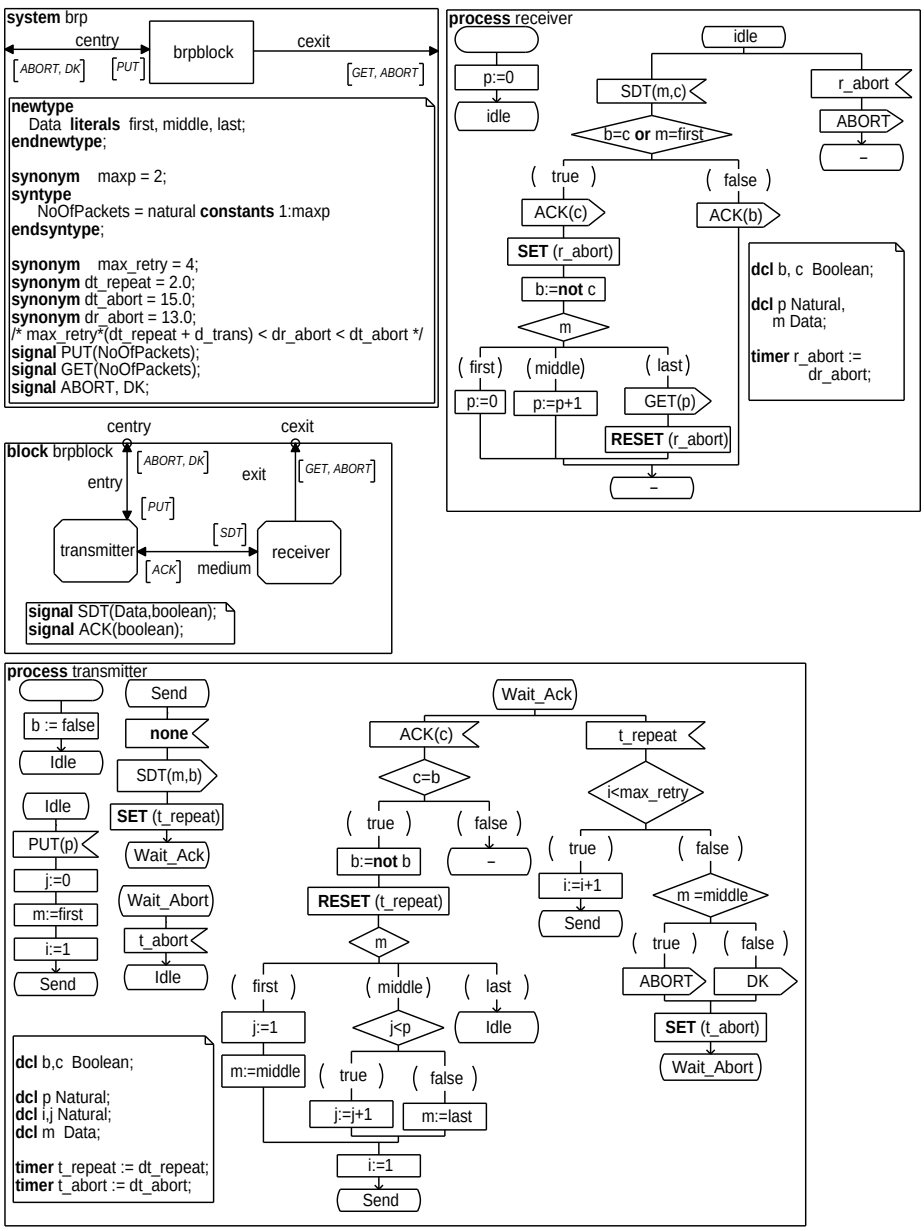

Fig. 1. The Bounded Retransmission Protocol in SDL

6.2 Modeling of the timed behaviour

One of the main correctness criterion of this protocol is that both the `Transmitter` and the `Receiver` should decide coherently on the abortion of a file transfers. However, this can be achieved only when precise constraints are fulfilled between timers values and action durations. In particular:

- if the timer `t_repeat` expires (repeatedly) too late, the timer `r_abort` may expire before the `Transmitter` has actually aborted the transmission of the file, the `Receiver` will consider that the current transfer has been aborted and consider following packets of the same file as packets of a new file;
- if the timer `r_abort` expires too late, that is, when the `Transmitter` has aborted the current transfer *and* received the timeout of `t_abort`, it will proceed to the next file after an abortion before this abortion has been detected by the `Receiver`;

As stated in section 4, if this specification is simulated following the Z.100 time semantics, nothing can be ensured concerning the relative expiration time of the different timers in `Transmitter` and `Receiver`. Therefore, even if the timers are set to correct values, incorrect execution scenarios will be observed, leading to the conclusion that the specification is not correct. However, this conclusion does only hold if *no* assumption about the relative speed of the clocks in the two communicating processes can be made.

On the other hand, simulating this specification using the current default time semantics of *Object*GEODE (i.e., each transition takes 0 time and is considered **eager**) is also not satisfying since it excludes realistic scenarios. For instance, using this semantics, the timer `r_abort` never expires *before* the reception of an expected packet (expiration will take place only after the packet loss). Thus, this too deterministic time behaviour will only lead to partial validation results.

These two limitations can be avoided using the notion of transition urgency introduced in section 5. More precisely, **lazy** and **delayable** transitions are used to specify some parts of the system supposed to take a certain amount of time to execute, or those occurrence is only controlled by the environment (they may occur at a specified or unspecified frequency). All other transitions (and in particular timeout expirations) are supposed to be **eager**. In the BRP specifications the non **eager** transitions are the following:

- The transfer requests (`PUT(p)`) issued by the environment, which may occur at an unspecified rate, and which should therefore be declared as **lazy**;
- The packet transmission (`Sdt(m,b)`), which is supposed to take a non deterministic amount of time within a given interval to model the transmission delay, and which should be declared as delayable. (Note that the delay required to transmit the acknowledgments are omitted here, but they could have been introduced similarly).
- A bounded difference of clock speeds in the two concurrent processes could be simulated by using timers expiring within some interval, instead of an exact time.

Figure 2 gives a correct specification of the `Transmitter` process including the urgencies annotations.

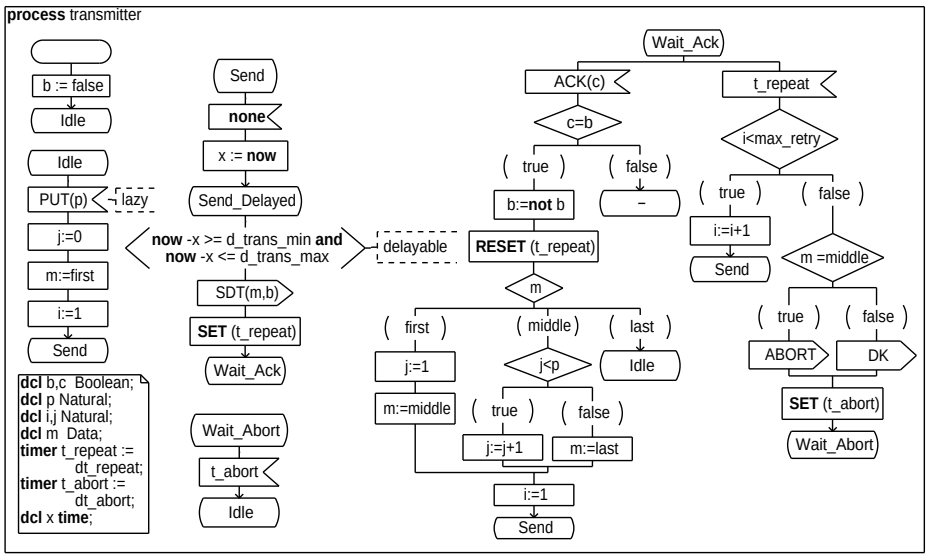

Fig. 2. The specification of `Transmitter` using urgencies

7 Related work

We present here two existing proposals for extending SDL with time-related constructs, and we show how the semantic framework introduced in Section 5 is fully compatible with both of them.

7.1 *Object*GEODE Performance Evaluation Extensions

The *Object*GEODE Simulator implements a series of SDL extensions, for modeling timing properties of systems. The modeler has the possibility to split the system among multiple processors, to give priorities to processes, and to declare execution durations on actions.

We can use these extensions to specify the process `Transmitter` from our example in Section 6. Namely, we use the *Object*GEODE extensions to model the non-deterministic waiting time before the transmission of signal `SDT`, as shown in Fig. 3: this transition may replace the transition outgoing from the state `Send`, in the initial specification of the BRP protocol (Fig. 1).

In *Object*GEODE, execution durations on actions are specified statically, by a time interval and a probability distribution (not considered here). Actions that have no specified duration are considered to take 0 time. The semantics of time consuming actions is the following: when an agent reaches a time consuming action, it enters an implicit state in which it stays for a time period complying to the specified interval. While the agent is in that state, only agents executed by other physical processors may execute. The other agents executed by the same physical processor as the blocked agent are blocked too. When time elapses, the agent exits the implicit state and executes the action in 0 time.

In our example, the simulator executes all the actions described in the process `Transmitter` before the informal task *'non-deterministic wait'*. Then, the simulator puts the `Transmitter`

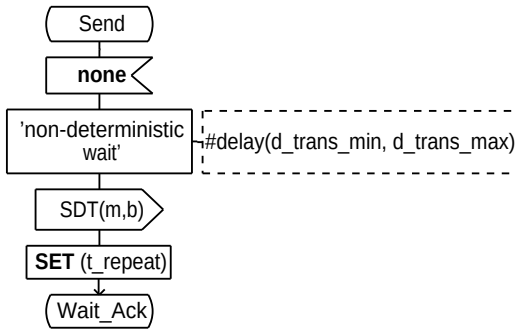

Fig. 3. The BRP `Transmitter` delay modeled using the *Object*GEODE performance evaluation extensions

in an implicit state, where it stays for a period of `d_trans_min` to `d_trans_max` time units. At the end of this period, the `Transmitter` exits the implicit state, and the simulator executes the output of `SDT` in 0 time.

This semantics can be captured using urgencies. Associating a delay to an action is equivalent to splitting the initial transition with an implicit intermediate state and an implicit delayable transition, as shown in Fig. 2.

The advantage of using our semantic framework for expressing execution times is that our analysis methods allow to consider both lower and upper limits simultaneously during simulation. The analysis methods we have developed on our model work with time intervals, and we can compute the minimal/maximal time span between two arbitrary occurrences in the system.

7.2 QSDL

Queuing SDL (QSDL, [10]) is an extension of SDL with constructs for modeling timing properties of systems, developed at the University of Essen, Germany. QSDL was developed for doing performance modeling and analysis on SDL systems.

The tool supporting QSDL, QUEST [11], implements a discrete-time semantics that resembles the semantics implemented in *Object*GEODE and TAU. Time passes in simulation states, normal transition actions take 0 time to execute. Additionally, QSDL introduces a new SDL statement, which takes time and which may be put on transitions: REQUEST. Like in *Object*GEODE, described in the previous section, this time consuming action introduces in fact an implicit simulation state, in which the calling agent stays for as long as the REQUEST takes.

The difference between QSDL and the *Object*GEODE performance evaluation extensions comes from the fact that the execution time of a REQUEST is not specified statically. QSDL uses the concept of queuing machine to compute the dynamic execution time of a REQUEST. Queuing machines represent computing resources shared between several agents of an SDL system, for which the agents compete.

For projecting the QSDL extensions on our semantic framework which uses transition urgencies to control time progress, we would need to model QSDL queuing machines by SDL automata annotated with urgencies. The task is not trivial, because the behavior of a queuing machine depends on a series of parameters:

1. *the speed.* The absolute amount of work, which is a parameter of the REQUEST, is first divided by the speed of the machine, to obtain an amount of work relative to the machine
2. *the number of processors.* A machine may have from one to an infinity of processors. Perfect parallelism is assumed (i.e. if a machine has n requests to process simultaneously, m processors, and a speed s, the rate at which each request is processed is $r = \frac{m}{n}s$ if $n \geq m$ and $r = s$ if $n < m$).
3. *the scheduling policy.* In case of multiple, competing requests, the scheduling policy determines which requests are serviced and which are put on hold. QSDL defines the following scheduling policies: FIFO with three variants (non-preemptive, priority non-preemptive, and priority preemptive), Processor Time Sharing, Infinite Processors, Random non-preemptive, and LIFO priority preemptive. For details, see [11]

Contrary to our approach which relies on static hypothesis, QSDL introduces a dynamic calculus of action durations depending on the system architecture and the scheduling policies used. However, both extensions are fully compatible and can be advantageously combined to improve the development process.

8 Conclusion

In this paper we have identified some of the major weaknesses of SDL for the development of real-time systems, in particular on the specification side. To fill these gaps, our proposal relies first on introducing a new timed semantics for SDL, based on timed automata with urgencies. The benefits of using such a semantics are numerous: it is simple and intuitive, it relies on a well-defined and well-studied theoretical model, it allows to express most of the real-time primitives, and several verification tools have been already developed upon this model with very good results in practice [13, 19, 15]. It now remains to propose adequate user-level extensions of SDL based on this underlying semantics, which could come from industrial users facing the problems that we have described.

Finally, since several semantics may be necessary to cover different needs (simulation, validation, code generation, performance evaluation, etc.), we suggest to adopt multiple *semantic profiles* for SDL, each one being dedicated to some particular steps of the development process. Introducing such profiles in the SDL standard has the advantage that all existing profiles would be concentrated together, and that *compliance relationships* between profiles could be defined formally. A theoretical basis for defining profiles and inter-profile compliance relationships does not exist and is hard to develop, but the current practice of SDL tools, which employ the notion of profile implicitly and without discipline, demands it.

References

1. Languages for telecommunications applications - specification and description language (SDL). ITU-T Recommendation Z.100, 1996.
2. Requirements for UML profiles. OMG document ad/99-12-32, December 1999. OMG ADTF Green Paper.
3. R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, (104):2–34, 1993.
4. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.
5. 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. Dssouli, G.v. Bochmann, and Y. Lahav, editors, *Proceedings of* SDL *Forum'99*, Montreal, Canada, 1999. Elsevier.
6. 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.
7. M. Bozga, J.C. 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.v. Bochmann, and Y. Lahav, editors, *Proceedings of* SDL *Forum'99*, Montreal, Canada, 1999. Elsevier.
8. M. Bozga, S. Graf, L. Mounier, and J. Sifakis. The intermediate representation IF. Technical report, Verimag, 1998.
9. P.R. D'Argenio, J-P. Katoen, T. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! Technical report, Univ. of Twente, 1997. Report CTIT 97-03.
10. 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 *Forum'95*. Elsevier Science B.V., 1995.
11. M. Diefenbruch, J. Hintelmann, and B. Müller-Clostermann. *Quest User Manual*. University of Essen, Dept. of Mathematics and Computer Science, Essen, Germany, March 1998.
12. J-F. Groote and J. van de Pool. A bounded retransmission protocol for large data packets. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 536–550. Springer-Verlag, 1996.
13. K.G. Larsen, P. Petterson, and W. Yi. UPPAAL: Status & Developments. In O. Grumberg, editor, *Proceedings of CAV'97*, volume 1254 of *LNCS*. Springer, June 1997.
14. R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 98–113. University of Maribor, Slovenia, June 1996. Also available as INRIA Research Report RR-2965.
15. I. Ober, B. Coulette, and A. Kerbrat. Timed SDL simulation and verification: Extending SDL with timed automata concepts. Technical report, Telelogic Technologies Toulouse, 2000.
16. J.-L. Roux. SDL performance analysis with *Object*GEODE. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in* SDL *and* MSC, Erlangen, Germany, February 1998.
17. Telelogic A.B., Malmö, Sweden. T*elelogic* TAU SDL *Suite Reference Manuals*, 1999.
18. VERILOG, Toulouse, France. ObjectGEODE *4.1 Reference Manuals*, 1999.
19. S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.