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

1 Introduction

The ITU–T Specification and Description Language (SDL, [?]) 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 in SDL, or the difficulty to program atomic transactions, to *high-level modeling* issues, like the difficulty to model time non-deterministic system components, or the impossibility to use the standard formal semantics of SDL in 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 different 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 dispose of analysis methods that work on top of this semantic framework, and by which we can verify general timing properties of a SDL 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 allow to solve these problems. Finally, we briefly compare our work with other proposals [?,?].

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, which means 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 SDL code).
2. *Usability* problems relate to the way an SDL model is used in analysis and early design: the modeler must be able to simulate the system or formally verify certain properties. An usability problem is a problem in the definition of the SDL semantics, which makes it practically or theoretically difficult to construct the global state graph of a SDL system (graph that is used by simulation or verification tools).

2.2 Semantic Profiles for SDL

The semantics of SDL, as presented in [**?**], 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 a time that is external, unrelated to the SDL system, is practical for code generation in the sense that actual implementations of the system conform to it. However, for simulation and verification, this semantics of time is utterly impractical: timer extents do not have any significance, any timer that gets in a queue may stay there for an indeterminate amount of time.

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, possibly causing an explosion of the state space. Moreover, since no control is ensured over time progress many undesirable execution paths can be obtained during this exhaustive simulation. We have here a *usability* problem, as characterized in section 2.

In practice, 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 SDL systems 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 SDL 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* of SDL. The idea of defining multiple semantic profiles of a language is not new: the UML community is on the way to defining several profiles for UML [**?**], each one fit 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 the different usages of a SDL model: code generation, 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 a 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 semantics, and 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 [**?**] supports such a parameter suggests it is useful). There is no need to have two whole different 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 was already undertaken by SDL tools, which are highly parameterized. The parameters used by tools such as *Object*GEODE [**?**] and TAU [**?**] 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 review some of them here.

3.1 Interruptive Timers

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

SDL timer timeouts are always received as asynchronous messages. For general-purpose time dependent code this is usually fine, but it may be difficult to write real timeout emergency procedures using timers. To ensure that a piece of code is executed immediately as a consequence of a timeout, the SDL programmer must first make sure that the agent handling the timer is idle when the timer is received. If this is not the case, then the process may consume the asynchronous timer message from the message queue only when it finishes its current job, which may be too late.

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 the system time.

Our proposal goes towards the introduction of the notion of interruptive timer in the language. An interruptive timer is a timer that 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 may be achieved in SDL by directly inserting system calls in the SDL code. However, these are patterns that are very common in real-time systems, and SDL could benefit from native constructs for expressing atomicity and mutual exclusion.

Additionally, inserting system calls in SDL for achieving atomicity and mutual exclusion 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. The 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 benefic.

4 What Is Missing on the Specification Side?

On the specification side things are more critical. As mentioned in 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.
2. when two timers are set at the same time, the timer set 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, and eventually to state space explosion.

A semantic profile for simulation must give the simulator some control over the progress of time. 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.

These means of controlling the time progress in simulation are limited. There are cases when the user needs to control the simulation time 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 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 (a sequence 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. So in order to express high-level specifications, one needs to use programming features.

There exists already several approaches to introducing execution time assumptions in SDL specifications. The *Object*GEODE Simulator [**?**] uses a syntactic extension by which one can associate an execution time (interval) to an action. [**?**] uses a more elaborate approach in which execution times are dynamically calculated with the help of queuing machines, so that they are depending on the amount of work and on the charge of the system.

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 ([**?**,**?**]) 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 [**?**] 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 an SDL semantic profile for simulation and verification.

4.4 Flexible Channel Specifications

SDL defines channels as reliable means for transporting messages: a channel never looses 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 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 resulted description of the protocol cannot be used in simulation: the simulator will never cover the parts that handle signal loss.

In practice, when one needs to model a channel which losses messages, or which delays messages by a rule, he 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, which differ 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 looses 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 law followed by 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 an SDL 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 dispose of powerful analysis methods that allow to derive interesting timing information (such as the minimal/maximal time span between two events) and to verify timing properties of SDL systems.

The framework presented here is not a direct solution to the problems described in Sections 3 and 4. Instead, it may constitute the *underlying semantics* for other temporized extensions of SDL (such as [**?**,**?**]), 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 directly 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 [**?**,**?**] (timed automata), and [**?**] (timed automata with urgencies).

As stated in Section 4.1, in order for a semantics to be usable in simulation and verification, the simulator has to have control over the system time. In SDL, the 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 simulation state, and time does not progress while the simulator executes a system transition (that is, **now** is not modified during a transition). Note that we talk about simulation states and simulation transitions, which may differ from SDL states and transitions: for example, if transitions are not atomic but SDL statements are, there will be a simulator state between each of the SDL statements on a SDL transition, and there will be a simulator transition for each individual SDL statement.

Moreover, the progress of time in a simulator state is controlled (bounded) by the transitions that may 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* until the transition (or another enabled transition) is taken.
2. **lazy** transitions, which do not have priority over time. 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 SDL timed extensions introduced at user level. We can imagine an extension of SDL in which the user puts the urgency information directly in the SDL model, like in the example in Section 6. Urgency information may also be derived from other kinds of timed annotations, as we will see in Section 7.

Transition urgencies were implemented in IF [**?**,**?**], a specification language developed at VERIMAG for prototyping semantic variations of the constructs of a SDL-like language. We have also implemented the extensions in the *Object*GEODE Simulator [**?**], with 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 [**?**,**?**]. Section 7 is dedicated to showing how such user-level extensions are projected on our semantic framework, and what advantages we acquire by 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 splitted 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 transfer, and proceed to the next file. This protocol has been used as a running example for several verification tools[?,?,?], and we consider here a simple version mainly focussed on its timing behaviour.

The SDL specification of this protocol (figure 1) is composed of a `Transmitter` and a `Receiver` process, briefly described below:

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 sending each packet `(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 sent of a packet, the `Transmitter` starts a timer `s_repeat` and waits either from an acknowledgement issued by the receiver, or for the expiration of `s_repeat`. If a correct acknowledgement is received, it resets `s_repeat` and proceeds to the next packet, unless it was the last one. However, if `s_repeat` expires, the *same* packet is resent up to `max_retry` attempts (`s_repeat` being restarted after each resent). If none of these resent succeeds, then the `Transmitter` aborts the current file transfer and reports the failure to its upper layer. This is done either using an `ABORT` message when the current packet was a `middle` one, or using a `DK` (don't know) message when the current packet is a `first` or a `last` one (since in this case the `Receiver` may have either correctly received the entire file, or no received any packet at all). Finally, after a transfer abortion the `Transmitter` starts a timer `s_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 initialises its alternating bit, starts a timer `r_abort` and sends back an acknowledgement 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 an expected packet lates to arrive, then the timer `r_abort` expires and the `Receiver` can assume that the transfer has been aborted. It informs its upper layer (`ABORT`), and waits for a new file.

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 to abort the *same* file transfers. However, this is achieved only when precise constraints are fulfilled between timers values and action durations. In particular:

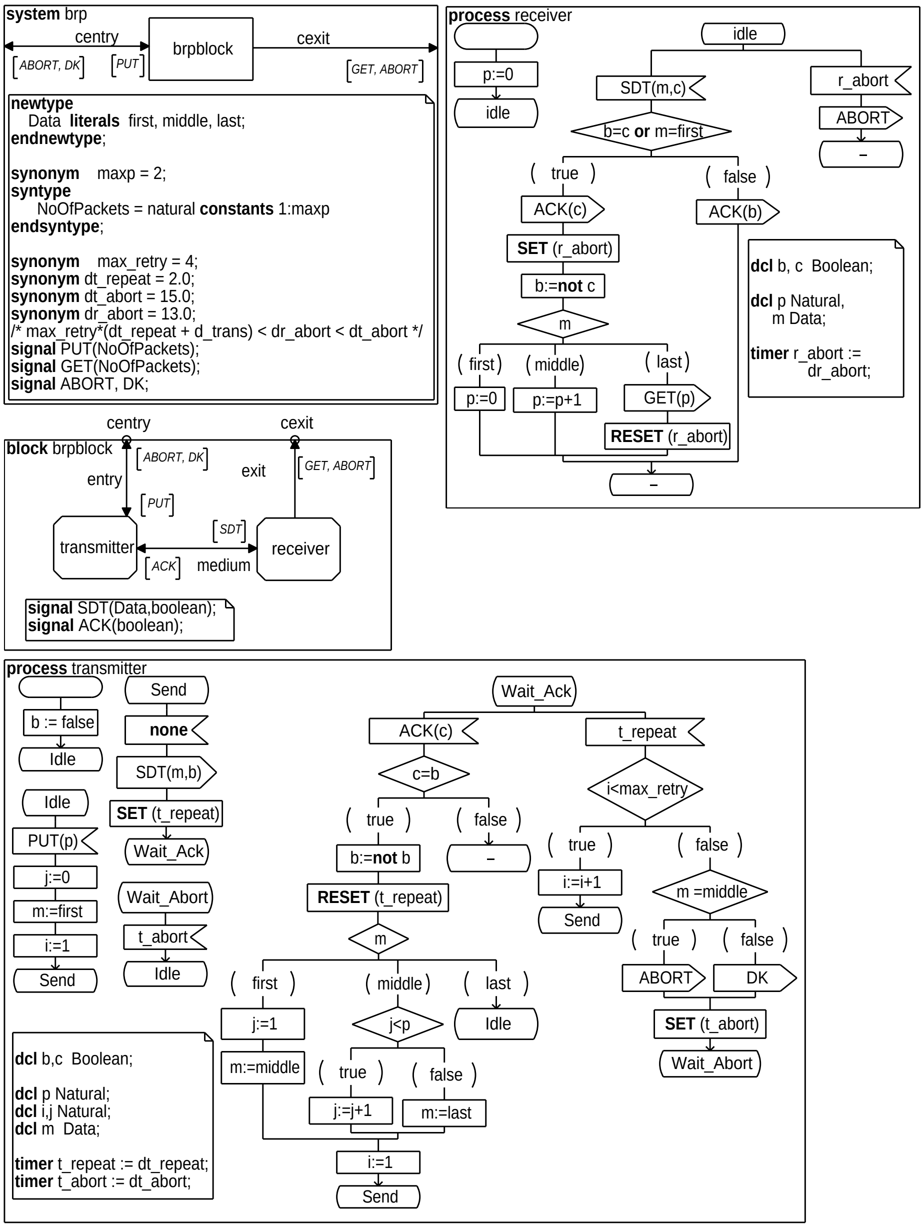

Fig. 1. The Bounded Retransmission Protocol in SDL

- if the timer `r_abort` expires too early, then the `Receiver` will consider that the current transfer is aborted whereas other packets of the same file may still arrive;
- if the timer `t_abort` expires too early, then the `Transmitter` will proceed to the next file after an abortion before this abortion was detected by the `Receiver` (which may never detect it later);
- finally, if the timer `t_abort` is set to a value smaller than `max_retry` times the transmission delay, then the `Transmitter` will always abort the current transfer ...

As stated in section 4, if this specification is simulated following the Z.100 time semantics, no guaranties are ensured about the relative expiration times of the different timers. Therefore, even if the timers are set to correct values, many incorrect (and irealistic !) execution scenarios will be observed, preventing any validation result.

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` can never expire *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 acknowledgements are omitted here, but they could have been introduced similarly).

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

7 Comparison with related works

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.

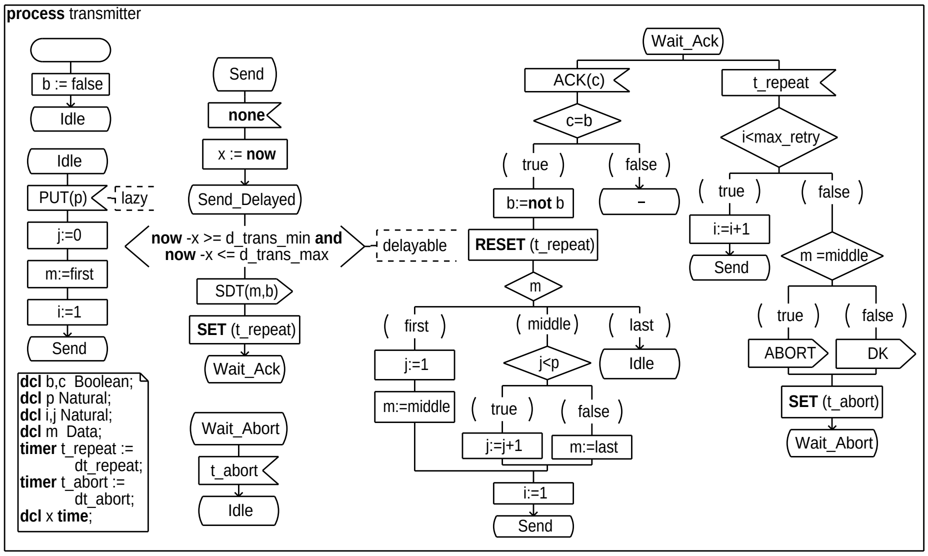

Fig. 2. The specification of `Transmitter` using urgencies

7.1 The *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. The transition shown in Fig. 3 may replace the transition outgoing from the state `Send`, in the initial specification of the BRP protocol (Fig. 1).

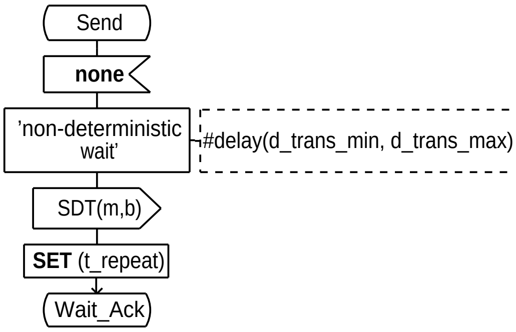

Fig. 3. The BRP `Transmitter` delay modeled using the *Object*GEODE performance evaluation extensions

In *Object*GEODE, execution durations on actions are specified statically, by a time interval and a probability distribution (not considerd here). The actions that have no duration specified, 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` 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, [?]) 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 [?], 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 [**?**]

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 Conclusions

9 conclusion

In this paper we have identified some of the major weaknesses of SDL for the development of real-time systems, in particular or 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 [**?**,**?**,**?**]. It now remains to propose adequate user-level extensions of SDL based on this underlying semantics, which could come from industial users facing the problems that we have described.

Finaly, 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.