Verification of Quantitative Temporal Properties of SDL Specifications *

Iulian Ober[1 2] and Alain Kerbrat[2]

[1] Institut National Polytechnique/IRIT, 2 Rue Camichel, 31000 Toulouse, France.
iulian.ober@enseeiht.fr
[2] Telelogic, 150 Rue N. Vauquelin, 31106 Toulouse, France.
{iulian.ober,alain.kerbrat}@telelogic.com

Abstract. We describe an approach for the verification of quantitative temporal properties of SDL specifications, which adapts techniques developed for timed automata [2]. With respect to other verification approaches applied to SDL, our approach broadens the class of analyzable specifications and improves the handling of non-deterministic systems, such as open systems communicating with an unspecified environment. Compared to the initial framework of timed automata, the application of these verification techniques to SDL raises two interesting issues, discussed in the paper. They are: expressing the semantics of time in SDL in terms of timed automata concepts, and employing a user friendly automata-based property specification language (GOAL [1]) to express and verify temporal properties. The paper also presents a verification tool prototype for SDL which implements these ideas.

1 Introduction

Automatic verification of behavioral properties is an important feature which can be offered by SDL tools, and which distinguishes SDL from other modeling languages used for specification and design of complex systems. Automatic verification is made possible by the existence of a formal semantics and syntax for the language [13], which can be used as a basis for mathematical reasoning about SDL specifications. Verification is an important task in the development of safety critical systems, which constitute an important share of the systems built with SDL.

Automatic verification capabilities implemented in industrial SDL tools (e.g. [19, 17]), employ techniques derived from model checking. Properties which can be verified range from simple safety properties (e.g. absence of deadlocks, invariance of a logical condition) to more complex safety or liveness properties (e.g. linear properties specified by (Büchi) automata). The value added by SDL compared to other automatic verification approaches is twofold:

* Work supported by the European project INTERVAL IST-1999-11557, *Formal Design, Validation and Testing of Real-Time Telecommunications Systems* (http://www.cordis.lu/ist/projects/99-11557.htm) and by the French RNRT Project PROUST (http://www-verimag.imag.fr/PROUST/).

1. model checking is done on the analysis or design model (a functional model), without the need for building more abstract models in another formalism
2. model checking is brought to the level of non specialists, by using simple and intuitive property specification languages (such as GOAL [1]) and tools.

As SDL is increasingly used for designing and implementing real-time and embedded systems, the verification methods and tools for SDL must be extended to account for quantitative temporal properties. This paper presents the results of a work aiming to adapt analysis techniques initially developed for timed automata [2] to SDL. This work also points out a series of deficiencies of SDL, which diminish its expressivity and the power of the analysis tools based on it. For the identified problems, we suggest some ways to improve the language.

The rest of the paper is structured as follows: §2 presents the state of the art in the specification and verification of temporal behavior using timed automata. We complete this section with a discussion of the limitations of timed automata, and their impact on our SDL approach. In §3 we show how the SDL execution model relates to the timed automata model. We also discuss possible improvements of SDL suggested by this relation. In §4 we examine how GOAL [1] can be used to describe temporal properties of SDL systems. In §5 we present the temporal verification facilities implemented in an extended version of the *Object*GEODE tool [19]. Finally, we draw the conclusions of this work and examine further advancement possibilities.

1.1 Related work

The idea of applying timed automata techniques in higher level formalisms was intensively studied during the past few years. SDL is not the only language to benefit from this trend: proposals for the improvement of both formal description techniques (Lotos, Estelle) and more informal modeling languages (UML) using timed automata are being studied.

For relating SDL and timed automata, [5] propose an intermediate formalism, called IF, in which one can describe a system of asynchronously communicating processes much in the same way as in SDL. The authors propose a methodology in which SDL specifications are translated to IF, and IF models are analyzed using timed automata methods and tools.

[8] presents an approach in which a variant of timed automata, called Timed Finite State Machines (TFMS), are used as intermediary form for generating SDL specifications and timed test cases from timed scenario descriptions. Timed automata techniques are not used for verification of properties, but the authors point out some of the same deficiencies of SDL that we reveal in this paper.

The general problems with expressing time-related behavior, discovered during this work and the work of our project partners are systematized in [6].

2 Specification and verification of temporal behavior

2.1 Reasoning about timed systems

SDL contains constructs for describing time-driven behavior: the designer can use timers or enabling conditions involving the time variable **now** in order to describe such behavior. Thus, the execution of an action may be triggered or conditioned by time.

While these constructs can model infinitely complex behavior, SDL has two major drawbacks when one wants to *verify* temporal properties of timed systems:

1. The formal semantics of the language [12] is loose about time progress: indefinite amounts of time may pass while a process is in a state even if it has a valid input signal waiting in the queue, and actions take indefinite times to execute. The only system component which behaves strictly with respect to time is the underlying agent responsible for keeping track of timers and sending timer expiration signals. With such loose assumptions about the performance of the underlying execution machine, it is difficult to guarantee almost any time-related property about system behavior. This problem was examined in more detail in [6].
2. Timers and conditions on **now** can describe infinitely complex behavior, for which it is difficult to conceive analysis methods and algorithms.

Several formalisms have been proposed for modeling time-conditioned behavior. Among them, we look into timed automata because they are extended versions of finite automata, and thus semantically related to SDL. Timed automata cope with both problems mentioned above:

1. They provide stronger requirements on time progress, which can be constrained by the state of the automaton. Thus, one can specify actions that occur at a specific moment or within a bounded time, unlike in SDL.
2. Time conditions can only have simple forms. In timed automata, the only mechanism to measure time is the clock. An automaton may use several clocks at a time, all of which progress at the same rate and can be initialized and tested separately. Time conditions are represented by conditions on clocks, which can only have the following forms: $x \sim c$ and $x - y \sim c$, where x, y are clocks, $c \in \mathbb{Z}_+$ is a constant, and $\sim$ is one of $\leq$,$<$, $=$, $>$, $\geq$.

These restrictions make it possible to solve analytically a series of problems on timed automata, such as the reachability problem ("given a configuration of the timed automaton, i.e. a state and a set of values of the clocks, is there an execution which leads from the initial state to that configuration"), or various model checking problems.

For timed automata techniques to be applied to SDL, a SDL specification has to conform, in a way, to the restrictions mentioned above. We will discuss the implications of this in §3.

2.2 Timed automata

There are several, more or less restrictive ways of defining timed automata. We consider the following definition which includes the notion of transition urgency[1]:

Definition 1 (timed automaton). *A* timed automaton *is a tuple* $A = (\Sigma, \mathcal{X}, Q, q_0, E, \mathbf{inv})$ *where:*

1. *Σ is a finite set of transition labels.*
2. *$\mathcal{X}$ is a finite set of clocks.*
3. *Q is a finite set of discrete states.*
4. *q_0 is a distinguished state of Q called* initial state.
5. *E is a set of transition edges between the states from Q, each edge $e \in E$ having the following components: $e = (q, \zeta, u, a, X, q')$. $q, q' \in Q$ are the source and destination states. ζ, also denoted $guard(e)$ is the guard of the transition and it is a conjunction of* atomic conditions *on the clocks $\mathcal{X}$ of A. An atomic condition has one of the following two forms: $x \sim \mathbf{c}$ or $x - y \sim \mathbf{c}$ where $x, y \in \mathcal{X}$, $\sim \in \{<, \leq, >, \geq\}$ and $\mathbf{c} \in \mathbb{Z}_+$ is a constant. We will denote $\mathcal{CP}(\mathcal{X})$ the set of conjunctions of atomic conditions over the clocks of $\mathcal{X}$. $u \in \{eager,\ lazy,\ delayable\}$ is an attribute called the* urgency *of the transition. $a \in \Sigma$ is the label of the transition edge e. $X \subseteq \mathcal{X}$ is the set of clocks reset during the transition e, also denoted $reset(e)$.*
6. *$\mathbf{inv} : Q \longrightarrow \mathcal{CP}(\mathcal{X})$ is a function that associates to each state q of A a conjunction of atomic conditions on the clocks from $\mathcal{X}$ called the* invariant *of the state q.*

Timed automata as labeled transition systems. A semantics is given to timed automata by associating an (infinite) labeled transition system (LTS) G_A to each timed automaton A. This infinite LTS, called the *semantic graph* of the timed automaton, is defined by the following:

1. The nodes of G_A are called *configurations* or *dynamic states* of A. They are pairs $(q, \mathbf{v})$ where $q \in Q$ is a discrete state and $\mathbf{v}$ is a *valuation* of the clocks of the automaton, $\mathbf{v} : \mathcal{X} \to \mathbb{R}_+$. In order for a configuration to be consistent, the $\mathbf{v}$ must satisfy the invariant $\mathbf{inv}(q)$.
2. The edges of G_A correspond to transitions of A from one configuration to another. There are two kinds of transitions allowed in a state $(q, \mathbf{v})$:
   - **Discrete transitions** happen when a transition edge $e = (q, \zeta, u, a, X, q')$ is taken. e is *enabled* in $(q, \mathbf{v})$ iff $\mathbf{v}$ satisfies the condition ζ. When the transition e is taken, the system moves to state $(q', \mathbf{v}')$ where $\mathbf{v}'(x) = \mathbf{v}(x), \forall x \in \mathcal{X} \setminus X$ and $\mathbf{v}'(x) = 0, \forall x \in X$. The transition is denoted by $(q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')$.
   - **Time transitions** happen when an amount $\delta \in \mathbb{R}$, $\delta > 0$ of time elapses without any discrete transition being fired in the meantime. A time transition moves the system from state $(q, \mathbf{v})$ to state $(q, \mathbf{v}+\delta)$ where $\mathbf{v} + \delta$ denotes the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = \mathbf{v}(x) + \delta, \forall x \in \mathcal{X}$. The

[1] Urgencies for timed automata transitions were proposed in [3, 4]

transition is denoted $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v} + \delta)$. Time is considered to progress only if the automaton is prepared for that, i.e. the time transition is *enabled* iff the following conditions hold:

(a) $\forall \delta' \in (0, \delta],\ \mathbf{v} + \delta' \in \mathbf{inv}(q)$
(b) $\forall \delta' \in [0, \delta)$, there is no *eager* transition $e = (q, \zeta, u, a, X, q')$ enabled in $(q, \mathbf{v} + \delta')$ (i.e. $u = eager$ and $\mathbf{v} + \delta' \in \zeta$).
(c) $\forall \delta',\ \delta''$ such that $0 \leq \delta' < \delta'' \leq \delta$, there is no *delayable* transition $e = (q, \zeta, u, a, X, q')$ enabled in $(q, \mathbf{v} + \delta')$ and disabled in $(q, \mathbf{v} + \delta'')$ (i.e. $u = delayable$ and $\mathbf{v} + \delta' \in \zeta$ and $\mathbf{v} + \delta'' \notin \zeta$).

It is considered that G_A contains only those vertices which are reachable from the initial configuration of the system, which is $(q_0, \mathbf{v})$ with $\mathbf{v}(x) = 0,\ \forall x \in \mathcal{X}$.

Decidable problems for timed automata. An interesting problem concerning an automaton A is whether a particular state $(q, \mathbf{v})$ is reachable from the initial state $(q_0, 0)$. This is called the reachability problem of timed automata. It is important in that many properties of a timed automaton (e.g. invariance properties) may be solved if reachability is solved.

[2] proves the decidability of the reachability problem for timed automata[2], by constructing a finite abstraction (the *region graph*) from the potentially infinite semantic graph G_A, and showing that it preserves reachability. The same abstraction can be used to solve other problems, such as satisfaction of certain temporal logic formulas.

The simulation graph. For verification problems involving only reachability or linear properties (see §2.3), there are more efficient analysis methods than the region graph mentioned above. The *simulation graph* is one of them, which we use for verifying temporal properties of SDL specifications.

First, note that in a configuration $(q, \mathbf{v})$ of the automaton A, $\mathbf{v}$ can be regarded as a point in the space $\mathbb{R}^{|\mathcal{X}|}$. Then, the set of valuations $\mathbf{v}$ which satisfy a condition $\zeta \in \mathcal{CP}(\mathcal{X})$ form a convex polyhedron in $\mathbb{R}^{|\mathcal{X}|}$ (which can be identified with the condition ζ and denoted identically). Finally, let $\mathcal{NCP}(\mathcal{X})$ denote the set of non-convex polyhedra in $\mathbb{R}^{|\mathcal{X}|}$, i.e. finite unions of convex polyhedra from $\mathcal{CP}(\mathcal{X})$. We will call *zone* an element from $\mathcal{NCP}(\mathcal{X})$.

The *simulation graph* of A has couples (q, S) as vertices, where $q \in Q$ is a discrete state and $S \in \mathcal{NCP}(\mathcal{X})$ is a zone. The following operations are defined on (q, S) pairs:

$$\begin{aligned}
\textsf{time-succ}((q, S)) &= (q, \{\mathbf{v}' \mid \exists \mathbf{v} \in S,\ \delta \in \mathbb{R}.\ (q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v}')\}) \\
\textsf{disc-succ}(e, (q, S)) &= (q', \{\mathbf{v}' \mid \exists \mathbf{v} \in S.\ (q, \mathbf{v}) \xrightarrow{e} (q', \mathbf{v}')\})
\end{aligned}$$

where e is an edge between q and q'.

[2] [2] refers to timed automata without urgencies. The proof can be extended to timed automata with urgencies, by showing that time restrictions imposed by urgencies preserve the form of the regions.

It can be proved that if S is a zone, $\mathsf{time\text{-}succ}((q, S))$ and $\mathsf{disc\text{-}succ}(e, (q, S))$ also yield zones[3]. The simulation graph of the automaton A is the smallest graph $SG(A)$ such that:

1. $\mathsf{time\text{-}succ}((q_0, 0))$ is a node of $SG(A)$
2. for every node (q, S) of $SG(A)$ and every discrete transition edge e from q to q', if $(q', S') = \mathsf{time\text{-}succ}(\mathsf{disc\text{-}succ}(e, (q, S)))$ and $S' \neq \emptyset$ then (q', S') is also a node of $SG(A)$ and $(q, S) \xrightarrow{e} (q', S')$ is an edge of $SG(A)$.

As a zone S is a finite reunion of regions (from the *region graph* [2]), it can be easily proved that the simulation graph is always finite. Moreover it contains all and only the reachable states of A, and every run of A is contained in a path from $SG(A)$.

2.3 Temporal properties

Like for modeling systems, several formalisms have been proposed for expressing temporal properties of systems (and in particular quantitative temporal properties). Complexity limits apply on these formalisms: there are properties which cannot, in general, be algorithmically verified on a timed automaton.

We present here some classes of properties which have been studied for timed automata, and can be solved using the *simulation graph*. The effective method for solving them is discussed later on in the case of SDL.

Absence of deadlocks. A deadlock in a timed automaton is a situation in which the automaton, arrived at a state $(q, \mathbf{v})$, cannot take any more discrete transitions (there is no delay δ, edge e, and state $(q', \mathbf{v}')$, such that $(q, \mathbf{v}) \xrightarrow{\delta}\xrightarrow{e} (q', \mathbf{v}')$.

Non-zenoness of runs. An infinite run of a timed automaton is called *zeno* if all along the run, time remains below a fixed finite limit. A *zeno* run corresponds to the situation where the automaton makes an infinite number of actions (transitions) within a bounded amount of time.

Invariance properties. An invariance property is a condition on the state and clocks of an automaton, which holds along all possible executions of the automaton. Only particular forms of invariants can be verified using the simulation graph: propositional logic formulas having as atoms either conditions on the discrete state q of the automaton, or simple atomic conditions on the clock values (conditions from $\mathcal{CP}(\mathcal{X})$).

Timed linear properties. More complex properties than the ones mentioned above are sometimes useful to express and verify. Take for example the following property: the action a of the system is always followed by b within at most d time units.

This property belongs to a class called *linear properties*, which are, informally defined, assertions about the set of possible executions of a system. Such properties generally express things like: there is an execution of the system which has a certain form, or all executions have a certain form.

[3] for timed automata *without urgencies*, convexity is also preserved, and therefore all zones of the simulation graph are from $\mathcal{CP}(\mathcal{X})$. See [18].

Linear properties may be expressed in various (equivalent) ways: as linear logic formulas, as finite automata – if they make assertions about finite executions of a system, as Büchi automata – if they make assertions about infinite executions of a system. We are interested in properties expressed through (finite or Büchi) automata, since we use GOAL [1], an automata-based property specification language, in connection with SDL.

The verification of properties expressed through timed Büchi automata (TBA), a simple extension of timed automata with Büchi acceptance conditions, has been studied in [18]. This verification problem is decidable using the *simulation graph*. We use a decision method similar to the one described in [18] to verify GOAL properties on SDL.

2.4 Decidability limits of timed models

Other models, more general than timed automata, have been studied. From a practical modeling point of view, for example, it is interesting to generalize the variation laws for the clocks, such that:

- clocks may vary with different relative speeds (*multirate automata*),
- a clock may vary with different speeds in different states (*integrator automata*), etc.

However, such models are usually decidable only under strong restrictions. For example, *multirate automata* are decidable only if clocks are not compared with one another, and *integrator automata* are decidable only under strong structural restrictions (for a synthesis of decidability results, see [10] and the work cited therein). Thus, timed automata give in a way a *complexity limit* up to which "general" timed models are decidable.

Semi-decision procedures can sometimes been developed for undecidable models such as those mentioned above. Such procedures are important from a practical point of view, in situations where timed automata are not expressive enough and only a generalized model can capture the behavior of a system. Nevertheless, these semi-decision procedures are usually complex and difficult to apply to large models. Therefore, in our work we have restricted to basic timed automata and equivalent SDL models.

3 A timed automata model for SDL

In order to apply analysis techniques developed for timed automata to SDL specifications, we select the SDL constructs that can be mapped to timed automata constructs (clocks, operations and tests on clocks).

3.1 The semantics of SDL

The execution model for SDL is given by the formal semantics of the language in [12]. The model of an SDL system is a multi-agent abstract state machine

(ASM). A multi-agent ASM, which is formed of a (dynamic) set of agents and a (dynamic) set of programs, has in turn its behavior defined in terms of *states* and *transitions* (modulo partial ordering, see [9]). Thus the semantic model of an SDL system is a labeled transition system (LTS) formed with the aforementioned *states* and *transitions*.

The reachability problem for SDL is undecidable, since the language does not impose any restrictions on variables and message queues (and therefore two-counter machines – which are undecidable in general [11] – may be simulated by an SDL system). However, this difficulty is overpassed in verification tools by imposing additional restrictions on the range of variables and message queue length.

However, free-formed expression containing **now** are allowed in enabling conditions, continuous signals, decision discriminants, and generally in any expression appearing in an SDL statement. This can also lead to non-decidability of the reachability problem[4].

This latter problem is more abruptly avoided by SDL verification tools. In most SDL verification tools (including *Object*GEODE and Telelogic TAU), in verification mode **now** is always considered to be 0, and bookkeeping is correctly done only for timers with relative expiration delays (i.e. set with a deadline $\mathbf{now} + d$, where d is an integer not depending on **now**).

This method rules out from the verification process all SDL systems in which **now** is used for other things than setting relative timers.

By defining a relation between the SDL execution model and timed automata, we aim to broaden the class of systems on which behavior properties can be verified, to include systems in which **now** is used not only for setting relative timers. The class of verifiable SDL systems will still be limited by the complexity limits of timed automata, which are however more loose than the one mentioned above.

3.2 Mapping SDL constructs to timed automata constructs

Among the SDL constructs which can be used to specify time-related behavior, there are two categories which can be represented through timed automata constructs:

Timers. Timers in SDL can be set to a deadline which can be computed in an arbitrary way. Absolute timers (set with a constant deadline d) and relative timers (set with a deadline of $\mathbf{now} + d$) are the most common. However, timers may be set with deadlines computed by arbitrarily complex expressions, such as $2 * \mathbf{now}$ or $\mathbf{now} * \mathbf{now}$ (these expressions are type-incorrect, but equivalent expressions involving type casts may be written).

Absolute and relative timers can be mapped to timed automata constructs, while timers set with general expressions cannot. The mapping of absolute and relative timers is based on the observation that each timer can be represented

[4] This is proved by the fact that an *integrator automaton* (§2.4) can be simulated by an SDL system.

by a clock, which is tested for equality with the deadline in all states following the arming of the timer (see the mapping details in §3.4).

Relative delays measured with now. Free-formed expressions involving the SDL global clock **now** cannot be mapped, in general, to timed automata constructs. There is however a case in which they can: if y is a variable, and the assignment $y := \textbf{now}$ appears on a transition, from that point on, the value of the difference $\textbf{now} - y$ can be simulated by a TA clock (x_y).

The assignment $y := \textbf{now}$ equates to resetting x_y to 0. An enabling condition, continuous signal or decision discriminant which compares $\textbf{now} - x$ with an integer constant is equivalent to a TA guard testing the clock x_y (see the mapping details in §3.4).

3.3 Expressivity considerations

Although we impose the above mentioned restrictions on the statements of SDL specifications, they are looser than the ones imposed by current verification tools. In this section we argue, based on an example, that from a temporal point of view our restricted SDL is strictly *more expressive* than the restricted SDL analyzed by current tools.

By allowing inequalities involving **now** in enabling conditions, continuous signals and decision discriminants, the SDL modeler has means to specify time-non-deterministic behavior. Such behavior cannot otherwise be specified using only timers. There are at least two cases in which time non-determinism must be modeled in the SDL specification:

1. open systems – in which the behavior of the environment is not completely specified but may conform to some conditions (e.g. a certain signal is sent within given time bounds, etc.).
2. incompletely specified systems – in which the timing of the system depends on an unspecified component (such as the underlying hardware), for which some timing information is available (e.g. minimum/maximum execution times, etc.).

One more construct is necessary for the accurate expression of non-deterministic behavior: a notion of *urgency*, similar to the one defined in §2.2 for timed automata. The *urgency* specifies the degree of non-determinism of a transition: *eager* transitions are deterministically triggered as soon as they are enabled (i.e. they block the progress of time) while *lazy* and *delayable* transitions are non-deterministically triggered after they are enabled, but they do not block the progress of time (*delayable* transitions still block time progress if they reach the upper limit of their enabling condition, beyond which they are disabled).

In the example below (Fig.2-3), urgency is attached to SDL transitions as a formal comment.

Example. We consider the SDL specification of the exchange level of the SpaceWire protocol stack. SpaceWire [16] is a protocol for high-speed data links, used by the European Space Agency for handling payload data on-board a

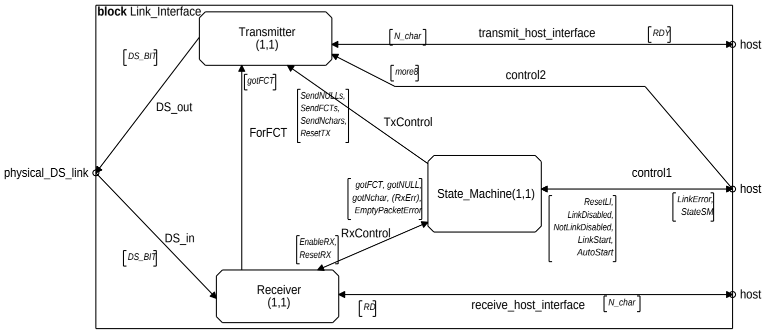

Fig. 1. The structure of a SpaceWire Link Interface

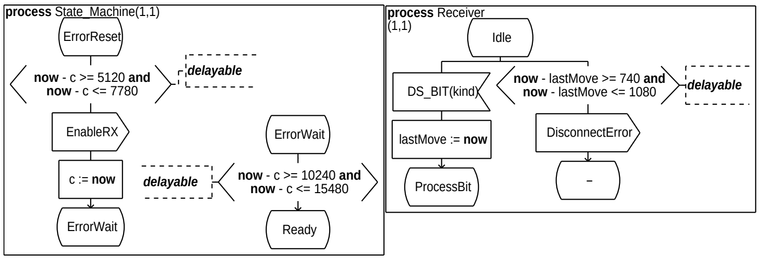

Fig. 2. Non-deterministic behavior of the SpaceWire Link Interface

spacecraft. The SpaceWire standard covers all the protocol levels from the physical links up to the transport layer. The Exchange Level referred in the sequel corresponds to the Data Link layer in the OSI stack.

The Exchange Level is built on top of an *unreliable serial full-duplex point-to-point* link, and offers functionality like:

- Error detection through parity checking. No re-transmission functionality is provided.
- Disconnection detection based on timeouts.
- Flow control using credit counters for avoiding buffer overflow.

The SDL specification of the exchange level of SpaceWire consists in a SDL block describing a SpaceWire Link Interface (see Fig. 1). The block contains three processes: a Transmitter, a Receiver and a State Machine, which correspond to the components described in the informal version of the SpaceWire standard [16].

The SpaceWire standard stipulates the timing requirements a link interface must satisfy during functioning. Examples of requirements taken from the standard are:

- the disconnect timeout of 850 ns nominal shall be between 740 ns and 1080 ns.

– after reset, the link interface remains in the *ErrorReset* state for a period of $5.12\mu s$ to $7.78\mu s$.
– during the reset cycle, the link interface remains in the *ErrorWait* state for 10.24μs to 15.48μs.

Fig. 2 contains excerpts from the behavior specification of the SDL processes composing the Link Interface. It shows how the three timing requirements mentioned above are modeled using *delayable* transitions triggered by continuous signals involving inequalities on **now**.

Such non-deterministic behavior cannot be specified in SDL using timers, which have a deterministic deadline. Moreover, existing industrial SDL verification tools work in discrete time and make simplified assumptions on the progress of time such as: time progresses only when there is no transition that can be triggered, and it always progresses up to the deadline of the next timer which it to expire [19]. In consequence, many time lines corresponding to realizable system executions are not explored by "classical" SDL verification tools.

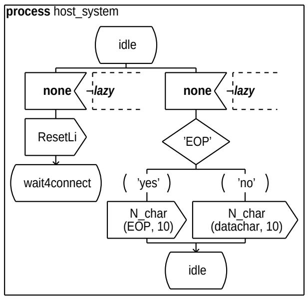

Fig. 3. Non-deterministic behavior of the host systems

The non-deterministic behavior can be modeled using inequalities on time as shown in Fig. 2. As we will see later, the analysis techniques we employ allow to explore all realizable time lines described by such specifications.

The behavior of the environment of most open SDL systems is also non-deterministic. For example, a component of the environment of a SpaceWireLink Interface is the host system. The host system may fail and reset the link at random moments, and it transmits characters over the link also at random.

For verification purposes, the above properties of the host system may be specified in SDL. Fig. 3 shows how this is done using *lazy* transitions (which, we remind, may let time pass with any amount before they are triggered). Modeling the host system like this leads to a more complete exploration of the realizable behaviors of the Link Interface.

3.4 Formalizing the mapping of SDL to timed automata

We take here only a brief look at how the relation between SDL systems (with the structural constraints described before) and timed automata can be defined formally. This provides an insight of the internal workings of the verification tool described in §5. Since the point is to define a mapping that is usable by this tool, which is an extended version of the *Object*GEODE verification tool, we start from the semantics of SDL implemented in the original tool [19] rather than from the standard semantics [12].

The *Object*GEODE discrete-time semantics of SDL. For a given SDL system $\mathcal{S}$, the *Object*GEODE verification tool builds a labeled transition system $\mathcal{G}_\mathcal{S}$ corresponding to the state space of $\mathcal{S}$. $\mathcal{G}_\mathcal{S}$ is used to verify behavioral properties.

The nodes of $\mathcal{G}_\mathcal{S}$ are global states of the SDL system, comprising the discrete state of each process instance, procedure or service, the values of all variables, the content of all queues, as well as the relative delay until expiry for each active timer in the system. The predefined variable **now** is not part of the system state, since this would cause $\mathcal{G}_\mathcal{S}$ to be infinite. (A consequence is that certain expressions involving **now** are wrongly interpreted in $\mathcal{G}_\mathcal{S}$.) $\mathcal{G}_\mathcal{S}$ may contain the following kinds of transitions between states:

1. **internal discrete transition.** $q_1 \xrightarrow{t} q_2$ iff there is a discrete transition identified by t enabled in the state q_1 and which takes the system into the state q_2. A discrete transition is caused by an input, a priority input, a continuous signal, a signal save/discard, etc.
2. **feed discrete transition.** $q_1 \xrightarrow{t} q_2$ iff t is an input transition, and the signal that causes it may be sent by the (unspecified) environment of the system.
3. **time transition.** $q_1 \xrightarrow{time(c)} q_2$ iff there is no internal discrete transition and no timeout transition enabled in q_1, and the next timer to expire has c time units until expiring. q_2 is equal to q_1 except for the delays of active timers, which are decreased by c.
4. **timeout transition.** $q_1 \xrightarrow{timeout\ t} q_2$ iff the timer t is active in q_1 and its relative delay until expiry is 0. q_2 is equal to q_1 except for the active status of t and the queue to which the signal t is appended.

Continuous time and loose time passage conditions. As mentioned in §3.3, the main advantage of using timed automata techniques in SDL tools is that time can be given a less constrained semantics, so that more of the realistic time lines of SDL system execution can be explored by a verification tool.

The time lines included in the $\mathcal{G}_\mathcal{S}$ graph built by *Object*GEODE are restricted by the conditions in which *time* transitions may be taken. We can define a continuous-time semantics by relaxing time passage conditions, thus obtaining a larger LTS, $\mathcal{G}_\mathcal{S}^\mathcal{T}$, called the *timed semantic graph* of the system $\mathcal{S}$.

$\mathcal{G}_\mathcal{S}^\mathcal{T}$ is defined similarly to $\mathcal{G}_\mathcal{S}$, except that the value of **now** is made part of the system state (i.e. the vertices of $\mathcal{G}_\mathcal{S}^\mathcal{T}$). We denote by q.**now** the value of **now** in the state q. For timers, what is kept in the states of $\mathcal{G}_\mathcal{S}^\mathcal{T}$ is not the relative delay until expiry (like in $\mathcal{G}_\mathcal{S}$) but the absolute deadline. Transitions are redefined in $\mathcal{G}_\mathcal{S}^\mathcal{T}$ in the following way:

1. **time transition.** $q_1 \xrightarrow{time(c)} q_2$ iff $c \in \mathbb{R}$, $c > 0$, and:
   (a) All components of q_2 are equal to the corresponding components of q_1, except for the value of **now**, for which $q_2.\mathbf{now} = q_1.\mathbf{now} + c$. In this case we denote q_2 by $q_1 + c$.

(b) $\forall \delta', \delta''.\ 0 \leq \delta' < \delta'' \leq c$, there is no discrete *eager* transition and no *timeout* transition enabled in $q_1 + \delta'$, and there is no *delayable* transition enabled in $q_1 + \delta'$ and disabled in $q_1 + \delta''$.

2. **timeout transition.** $q_1 \overset{timeout\ t}{\longrightarrow} q_2$ iff the timer t is active in q_1 and its deadline is equal to $q_1.\mathbf{now}$. q_2 is equal to q_1 except for the active status of t and the queue to which the signal t is appended.

Note that the *urgency* attributes of the transitions enabled in the source state q_1 are used in this definition to constrain time passage, like in timed automata (see §2.2). Therefore, all transition need to have a (default) urgency. Internal discrete transitions and *timer* transitions are *eager* by default, *feed* transitions are *lazy* by default (since it is logical to put only loose conditions on the behavior of the unspecified environment which sends *feeds*).

With the new conditions on time progress, it can be easily seen that $\mathcal{G}_\mathcal{S} \subseteq \mathcal{G}_\mathcal{S}^\mathcal{T}$ (the inclusion refers to both vertices – modulo the **now** part – and edges) .

Relation to timed automata. The mapping to timed automata is given by defining a timed automaton $\mathcal{A}_\mathcal{S}$ which has the property that its *semantic graph* (see §2.2) is strongly equivalent to the *timed semantic graph* $\mathcal{G}_\mathcal{S}^\mathcal{T}$ of $\mathcal{S}$. We will not go into all the details of this construction. A similar construction is described in [14] for an extended version of SDL, and the strong equivalence property is formally proved.

The idea is that the states/transitions of $\mathcal{A}_\mathcal{S}$ are defined similarly to those of $\mathcal{G}_\mathcal{S}$ (the "untimed" semantic graph of $\mathcal{S}$), with the following differences:

1. **Time transitions** from $\mathcal{G}_\mathcal{S}$ are deleted in $\mathcal{A}_\mathcal{S}$. Time passage in $\mathcal{A}_\mathcal{S}$ is represented by the modification of the clocks, which are defined below.
   As a consequence, the delay until expiry of each active timer (kept in the states of $\mathcal{G}_\mathcal{S}$ and updated by *time* transitions) will never be updated in the states of $\mathcal{A}_\mathcal{S}$ (i.e. will remain equal to the initial delay of the timer, until a timer transition is triggered, see below).
2. A **timer transition** $q_1 \overset{timer\ t}{\longrightarrow} q_2$ leaves each state q_1 of $\mathcal{A}_\mathcal{S}$ in which t is active, regardless of the delay of t.
3. A **discrete transition** $q_1 \overset{t}{\longrightarrow} q_2$ leaves each state q_1 of $\mathcal{A}_\mathcal{S}$ in which t could be triggered provided a condition on **now** (having the form defined in §3.2) is satisfied.

$\mathcal{A}_\mathcal{S}$ further has a set of clocks $\mathcal{X}$, which contains:

1. A clock $x_{t,p}$ for each timer t of a process instance p which can be activated during an execution of the system. Defined like this, the set $\mathcal{X}$ is potentially infinite. However, our implementation described in §5 handles dynamic clock creation and deletion, and therefore can partially analyze systems in which the set of timers poses problems.
2. A clock $x_{y,p}$ for each variable y of a process instance p which is used in assignments $y := \mathbf{now}$ and in tests over $\mathbf{now} - y$. Moreover, these variables y are not kept in the states of $\mathcal{A}_\mathcal{S}$, and assignments like $y := \mathbf{now}$ are represented in $\mathcal{A}_\mathcal{S}$ by the reset of $x_{y,p}$ on the corresponding transition.

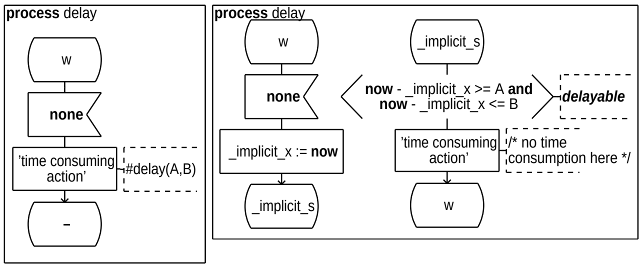

Fig. 4. Implicit transformation of time consuming actions

To simulate the temporal behaviors described by $\mathcal{G}_{\mathcal{S}}^{\tau}$, the transitions of $\mathcal{A}_{\mathcal{S}}$ have correspondingly defined guards:

1. A **timer transition** $q_1 \xrightarrow{timer\ t} q_2$ has the guard $\zeta = [x_t = v_t(q_1)]$, where x_t is the clock in $\mathcal{X}$ corresponding to t and $v_t(q_1)$ is the relative delay until expiry of t, which is part of the state q_1.
2. A **discrete transition** $q_1 \xrightarrow{t} q_2$ for which the transition t is guarded with a condition of the form $\mathbf{now} - y \sim \mathbf{c}$ in the original SDL system $\mathcal{S}$, has in $\mathcal{A}_{\mathcal{S}}$ the guard $\zeta = (x_y \sim \mathbf{c})$. SDL transitions guarded with conjunctions of conditions over **now** translate similarly.

With this definition, it can be proved that the *semantic graph of the automaton $\mathcal{A}_{\mathcal{S}}$ is strongly equivalent to the timed semantic graph $\mathcal{G}_{\mathcal{S}}$ of $\mathcal{S}$.*

In the verification tool presented in §5, we use the timed automaton defined here as semantic model for an SDL system. However, the automaton $\mathcal{A}_{\mathcal{S}}$ is built *on the fly*, at the same time with its *simulation graph*, which is actually used for verification.

3.5 Time consuming SDL tasks

The methods described in the previous sections can be used to analyze SDL systems in which transition execution times for specific statements are given by the designer. Previously, several extensions have been proposed for the modeling time consuming tasks in SDL [15, 7].

We consider the extensions proposed in [15], in which a delay can be associated to each SDL action statement. The delay is either a precise value A or an interval $[A, B]$. The semantics described in [15] is such that when the system reaches the time-consuming action, it stays blocked for a time $\delta = A$ (or $\delta \in [A, B]$) after which it executes the action in zero-time.

Time consuming tasks can be translated in an equivalent form in our model, by using an implicit state, an additional clock and a *delayable* transition exiting the implicit state to model the waiting conditions (given by A or $[A, B]$). The translation of time consuming tasks into delayable transitions is shown in Fig. 4. This translation is done implicitly by our verification tool (§5).

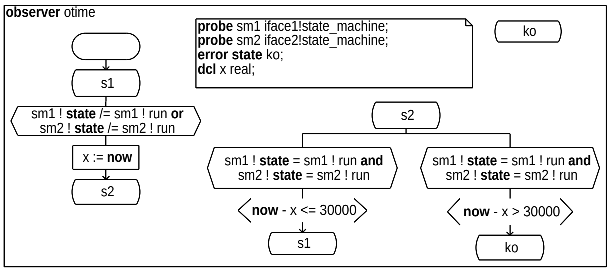

Fig. 5. GOAL observer for checking a timed safety property

4 Specification of temporal properties in GOAL

There are several classes of properties which can be verified for SDL systems. While simple properties, like absence of deadlocks or invariance of a logical condition do not need a complex formalism to be expressed in (propositional logic with an appropriate set of atomic propositions suffices), there are more complex properties which cannot be specified by these means.

Linear properties mentioned in §2.3 are from this class. The discrete-time verification tool of *Object*GEODE employs an automata-like language, GOAL, for the specification and verification of such properties. An introduction to GOAL can be found in [1].

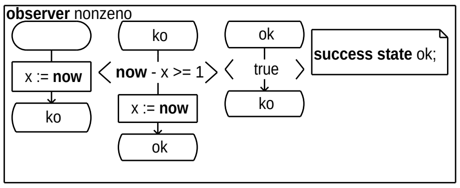

Fig. 6. GOAL observer for checking non-zenoness in *liveness* mode

Briefly, a GOAL observer has states, which are classified as **normal, error**, or **success** states, and transitions which are triggered by events occurring in the SDL system. The observer is executed synchronously with an SDL system. There are two verification modes possible:

1. **safety** verification, in which the GOAL observer is considered a finite automaton, whose accepting states are the **error** and **success** states. The verification tool looks for particular executions which lead to these states.
2. **liveness** verification, in which the GOAL observer is considered a Büchi automaton whose accepting states are the **success** states. The verification tool looks for *infinite* executions of the SDL system which are not accepted by this Büchi automaton (i.e. do not pass infinitely often through a **success** state), and which correspond to lack of progress ad infinitum.

The GOAL language may be used for verifying timing properties of the SDL system, by using the value of **now** in the same way it is used in SDL (§3). The difference between time driven transitions in GOAL and SDL is that GOAL transitions can only have *lazy* urgency. Allowing them to be *eager* or *delayable* would cut out potentially realizable system behaviors from the simulation graph, by blocking time passage.

Fig.5 shows how a GOAL observer may be used to verify that a SpaceWire link is always re-established in at most $30\mu s$ after a fault (safety property). Furthermore, Fig.6 shows an observer which can be used in liveness mode to verify that an SDL system has *no zeno runs* (on each infinite zeno run, the observer will eventually remain in the state ko forever).

5 The temporal property verification tool

In this section we describe a tool for verifying temporal properties of SDL systems, based on building the timed *simulation graph*. This tool is derived from the *Object*GEODE SDL Simulator [19], and therefore supports all the features of SDL–96 in addition to the extensions presented in the previous sections.

5.1 The simulation graph

The tool handles SDL specifications according to the timed automata model for SDL described in §3. Verification is based on building the simulation graph of the timed automaton $\mathcal{A}_\mathcal{S}$ corresponding to the SDL system $\mathcal{S}$.

The timed automaton $\mathcal{A}_\mathcal{S}$ itself is not explicitly built by the verification tool. The rules for building $\mathcal{A}_\mathcal{S}$ described in §3.4 are instead used for constructing the simulation graph of $\mathcal{A}_\mathcal{S}$ directly.

Simulator states. The dynamic states handled by the verification tool are pairs (q, S). q is the discrete state of the SDL system (containing the components described in the definition of $\mathcal{A}_\mathcal{S}$ (§3.4), i.e. no value kept for **now**, initial relative deadline kept for timers). S is a convex polyhedron over the clocks of $\mathcal{A}_\mathcal{S}$, which represents the clock values reachable via the scenario executed so far. The tool eliminates non-active clocks from S, such as clocks corresponding to inactive timers, so that S keeps only constraints on clocks which are relevant in the state q.

The discrete state and the clock constraints of the GOAL observers executed in parallel are part of q and S. Thus $\mathcal{A}_\mathcal{S}$ is in fact the synchronous product (actually the product synchronized only on observable events) of the timed automaton corresponding to the system $\mathcal{S}$ and the timed automata corresponding to observers.

Simulator transitions. When the simulator executes a transition t in a state (q, S), it performs several steps corresponding to the computation of $\mathsf{time\text{-}succ}(\mathsf{disc\text{-}succ}(t, (q, S)))$ from the definition of the simulation graph (§2.2). The steps are detailed below.

We note one difference with the functioning of the standard *Object*GEODE Simulator: in case of an SDL model executed in parallel with a GOAL observer, there may be more than one successor of the state (q, S) by the transition t. The cause is that, by executing t on the SDL model first, the simulator reaches another state (q', S') (see steps 1 and 2 detailed below), on which *more than one* GOAL *transition of a same observer may be fireable*. A simple example is the observer in Fig.5: consider the two transitions exiting state s2. If the SpaceWire link is restored in a state (q', S') which has non-void intersection with both guards **now** - x <= 30000 and **now** - x >= 30000, there will be two successors of the initial simulator state on the same SDL transition, but on two different GOAL transitions.

The steps of the execution of the transition t are:

1. The zone S of the state (q, S) is intersected with the guard of t.
2. The actions of transition t are executed on the contents of the discrete state q. If there are clocks reset by t – i.e. assignments of the form $x :=$ **now** – the zone resulted from step 1 is modified accordingly. The result (q', S') of these two steps corresponds to disc-succ$(t, (q, S))$.
3. For each GOAL observer, the fireable transitions are determined. There may be several fireable transitions $g_1, ..., g_k$. However, the parts of the simulator state (q', S') on which these transitions are fireable must not overlap – i.e. $guard(g_j) \cap S'$ must be disjoint for $j = \overline{1, k}$. This condition ensures that the behavior of the observer is deterministic for every actual system state $(q', \mathbf{v}) \in (q', S')$. The determinism condition is essential for GOAL observers, see [1].
4. For each GOAL transition g_j, there will be a successor state (q'_j, S'_j) computed in two steps similar to steps 1 and 2 above:
   (a) q'_j is computed from q' by applying the transformations described by the transition g_j.
   (b) S'_j is obtained from $S' \cap guard(g_j)$, by applying the clock resets described in g_j, if there are such resets.

   Additionally, if $S' \setminus \bigcup_{j=1}^{k} guard(g_j) \neq \emptyset$, no GOAL transition may be taken on this part of the state (called residual state). Therefore there will be a successor $(q', S' \setminus \bigcup_{j=1}^{k} guard(g_j))$.
5. If there are additional GOAL observers executing in parallel, they further split each state (q'_j, S'_j) (and the residual state) in substates.
6. Lastly, on each state (q', S') among the states resulted from the previous steps, the possible temporal successors $(q'', S'') =$ time-succ(q', S') are computed. The states (q'', S'') are the possible successors of (q, S) by t.

5.2 Verification of properties

The simulation graph built by our verification tool as shown above is a finite symbolic representation for the *reachable states* of the SDL system (and associated GOAL observers). It can therefore be used for the verification of linear properties, such as those described in §2.3.

Absence of deadlocks in a SDL system is verified with no attached GOAL observer. Thus for each state and each fireable SDL transition there is only one successor. A state (q, S) contains a deadlock if one of the following conditions holds:

1. (q, S) has no successor in the simulation graph, or
2. (q, S) has the successors $(q'_1, S'_1), ..., (q'_k, S'_k)$ by the transitions $t_1, ..., t_k$, but $S \setminus \bigcup_{j=1}^{k} guard(t_j) \neq \emptyset$ (meaning that there is an actual SDL system state $(q, \mathbf{v}) \in (q, S)$ on which no transition among $t_1, ..., t_k$ is fireable).

Invariance properties are given by a propositional logic formula that must hold in each state of the system. The formula is formed of atoms α which can test:

1. The *discrete state* of the SDL system, such as the value of a variable, the length of a queue, etc. In this case, the satisfaction of α by (q, S) is decided based on the discrete part q.
2. *Time.* Time may only be tested by means of atomic conditions of the form $\mathbf{now} - y \sim c$, where y is a SDL system variable interpreted as a clock. In this case, (q, S) satisfies α iff the polyhedron $\alpha' = [x_y \sim c]$ is included in the zone S.

Non-zenoness and other linear properties are specified by GOAL observers (see §4). In *safety* mode, the tool detects scenarios leading to states (q, S) in which q designates a **success** or **error** state in at least one GOAL observer. In *liveness* mode, it detects infinite loops which do not pass through states (q, S) in which q designates a **success** state in at least one GOAL observer.

5.3 Interactive commands and presentation of results

The verification tool can be also used in interactive mode, as a simulation tool. The user triggers transitions one by one and examines the reached states, like in normal *Object*GEODE interactive simulation [19]. However, the simulator states the user gets are symbolic states (q, S) of the same form as above.

The discrete part of the state can be examined using the usual *Object*GEODE Simulator mechanisms (watches, the `print` command), while the part S referring to clocks can be examined with an additional command, `clocks`, which prints all or a part of the constraints on clocks which define the polyhedron S. For example, the output of this command during the simulation of the SpaceWiresystem can look like:

```
> clocks
0 <= iface1!state_machine!c <= 10440
0 <= iface2!state_machine!c <= 7780
0 <= iface1!state_machine!c - iface2!state_machine!c <= 2660
```

Since the time span between two events in interactive simulation is not obvious (and sometimes not even possible) to derive from the clock conditions of S, there is a mechanism to measure time spans: the chronometer. A chronometer behaves like a *clock*, except that it is introduced from the simulation console. The command `addclock` creates a new chronometer, which can be consulted after several simulation steps to check the lower and upper limits of the elapsed time interval.

6 Conclusions and Future Work

We have presented a method for verifying quantitative temporal properties on SDL systems, by mapping the semantics of SDL to that of *timed automata* [2], for which powerful timing analysis methods exist. In order for the mapping and analysis to work, the SDL designer must use only a limited set of constructs for modeling timing conditions in the SDL specification: *timers* and *linear conditions* on **now** having a particular form.

However, the restrictions imposed by the mapping are looser than the restrictions imposed by most SDL verification tools, so a larger set of SDL models can be correctly analyzed with the methods presented here. We also argue that the analysis is more precise in the sense that it gives a better coverage of the possible behaviors of the system.

The use of *linear conditions* on **now**, together with a notion of transition *urgency* that we take from timed automata, make it possible to model more precisely systems presenting non-determinism from the temporal point of view. Incompletely specified systems and systems interacting with a non-specified environment are in this category. *Urgencies* have been proposed first for timed automata in [3, 4], and later for SDL in [6].

The verification method presented in the first part of the paper is implemented in a tool discussed in the final part. We examine the practical aspects of modeling temporal properties so that they can be verified by the tool, as well as the internal working of the tool.

From the experience we have with our tool, including the SpaceWire protocol used as example throughout this paper, we can conclude that the tool is able to give a better coverage of the behavior of the system, compared to classical SDL analysis tools. Further experimentation is taking place within the INTERVAL project[5].

Further work is needed in order to improve the tool: in particular cases, the use of transition urgencies may lead to non-convex clock polyhedra. Handling non-convex polyhedra poses theoretical difficulties, as signaled in [18]. Our tool can handle such polyhedra, but the performances are not always satisfactory.

Another direction for our efforts is to include the MSC language in our timed verification method. MSC'2000 offers the possibility to specify timing information, which could be exploited by our tool either in order to check temporal properties specified in MSC, or in order to generate timed MSC traces for simulation scenarios.

Acknowledgements. The authors wish to thank the European Space Agency for providing the SpaceWire specification, on which the example from §3 is based.

[5] *Formal Design, Validation and Testing of Real-Time Telecommunications Systems*, European project IST-1999-11557.

References

1. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL behaviors with GEODE. In R. Braek and A. Sarma, editors, SDL *'95 with* MSC *in* CASE. Elsevier Science B.V., 1995.
2. R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.
3. S. Bornot and J. Sifakis. Relating time progress and deadlines in hybrid systems. In *International Workshop HART'97*, volume 1201 of *LNCS*. Springer-Verlag, 1997.
4. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. Technical report, Verimag, Grenoble, 1998.
5. 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, SDL *'99. The Next Milenium. Proceedings of the 9th* SDL *Forum*, Montreal, Canada, 1999. Elsevier.
6. Marius Bozga, Susanne Graf, Alain Kerbrat, Laurent Mounier, Iulian Ober, and Daniel Vincent. SDL for real-time: What is missing? The 2nd Workshop on SDL and MSC, 2000.
7. 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.
8. A. En-Nouaary, R. Dssouli, and F. Khendek. From timed scenarios to SDL: specification, implementation and testing of real-time systems. In R. Dssouli, G.v. Bochmann, and Y. Lahav, editors, SDL *'99. The Next Milenium. Proceedings of the 9th* SDL *Forum*, Montreal, Canada, 1999. Elsevier.
9. Y. Gurevich. Evolving algebra: The lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
10. Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.
11. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adisson-Wesley, 1979.
12. ITU-T. Annex F (11/00) to recommendation Z .100 – Specification and Description Language (SDL) – Formal definition of SDL– to be published.
13. ITU-T. Recommendation Z.100 (11/99) - Specification and Description Language (SDL).
14. I. Ober. Extending SDL with timed automata concepts. Technical report, VERILOG, 1999.
15. 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. Friedrich-Alexander Universität, Erlangen-Nürnberg.
16. SpaceWire Working Group. SpaceWire: Serial point-to-point links. European Space Agency document UoD-DICE-TN-9201, Issue D, May 2000. http://www.estec.esa.nl/tech/spacewire.
17. TELELOGIC A.B., Malmö, Sweden. T*elelogic* TAU SDL *Suite Reference Manuals*, 1999.
18. Stavros Tripakis. *The Formal Analysis of Timed Systems in Practice*. PhD thesis, Joseph Fourier University, Grenoble, 1998.
19. VERILOG, Toulouse, France. ObjectGEODE *4.1 Reference Manuals*, 1999.