INTERVAL

Formal Design, Validation and Testing of Real-Time Telecommunications Systems

IST-1999-11557

Title : **Definition of the Timed Extensions**

Author(s) : Marius BOZGA, Zhen Ru DAI, Jens GRABOWSKI, Susanne GRAF, Dieter HOGREFE, Beat KOCH, Laurent MOUNIER, Helmut NEUKIRCHEN, Iulian OBER, Jean-Luc ROUX, Richard SINNOTT, Daniel VINCENT, Vassilis VELENTZAS

Editor : ITM LÜBECK

Date : 29 June 2001

Identifier : IST/11557/WP1/03.01/ITM/D13.1

Document Version : 3

Status : Proposed

Confidentiality : Public

Abstract : *(10 lines max.)*

This document defines the timed extensions for the SDL, MSC and TTCN languages. The basis was provided by the preliminary document M12 that was presented and discussed with the Interest Group (October 2000) and with the Study Group 10/Q7 at the last ITU-T meeting (November 2000). The chapter on SDL timed extensions introduces flexible time semantics and the assume/assert annotations that will be contributed to the definition of the new ITU-T/Z.108 standard (September 2001). Regarding MSC timed extensions, these are very limited as the notation already contains various facilities. An aspect to be further elaborated is the definition of an SDL-MSC compliance relation. For TTCN notation, timestamps for time critical events will be proposed to the TTCN-3 expert group at ETSI.

ITM LÜBECK (D), TELELOGIC (F), SOLINET (D), ERICSSON (S),
FRANCE TELECOM R&D (F), TELETEL (GR).

History

Date	Version	Comments
10 Oct 2000	1	First draft
26 Oct 2000	1	Updating after Interest Group meeting
06 Nov 2000	1	Partners contributions for SDL, MSC and TTCN
03 June 20001	2	Extensions as described in the accepted conference papers (SDL Forum 2001)
29 June 2001	3	Version delivered to CEC

Table of Contents

1. Introduction to the Deliverable

The purpose of this deliverable is to provide an account of the time extensions proposed by the INTERVAL project for the languages MSC 2000, SDL 2000 and TTCN-3. The extensions documented here are based on those given in the INTERVAL milestone M12 [1] and refinements of them, which have been accepted for publication at the SDL forum in June 2001 in Copenhagen [2],[3], and for presentation at the German workshop Formale Beschreibungstechniken (FBT 2001) in June inBruchsal [4]. These accepted publications form the core of the present deliverable.

The extensions considered in [2] are concerned with real time extensions of the SDL 2000 language and how the proposed extensions impact upon both the specification and subsequent validation of real time systems. The paper presents a precise picture of the concepts, for both modelling and analysis, that are needed to extend SDL for real-time engineering. It proposes a concrete syntax for these extensions and provides justifications and examples for its usefulness. It gives an intuitive idea how these new concepts can be integrated into the formal semantics of SDL 2000.

The focus of [3] is on MSC 2000 and TTCN-3 and their inter-relationship with regard to computer aided test case generation (CATG). The paper presents how generation of TTCN-3 test cases can benefit from the timed features introduced in SDL 2000 and MSC 2000. The translation of the MSC 2000 time concepts into TTCN-3 is described in detail. Since real time testing requires a test equipment which is fast enough to test an implementation, time constraints for the tester are necessary. It is shown how TTCN-3 can be used to benchmark the test equipment itself.

The extended abstract [4] gives directions on how TTCN-3 can be used for the specification of real-time tests. The main idea is to separate functional and non-functional requirements in the test specification. The described approach requires just a few extensions to TTCN-3 by reusing the log concept for recording timestamps of time critical events. Real-time requirements are expressed by the relationship of points in time. The collected timestamps can be evaluated during or after the test run.

Besides being introduced to the international community via conferences and workshops, these extensions are also in the process of being submitted as official proposals to the appropriate standardisation bodies responsible for maintaining the relevant standards and will likely appear (in some form) in future releases. The details of these submissions to standardisation will be documented in INTERVAL deliverable D43. The precise form of the extensions, e.g. the concrete syntax that will be proposed to standardisation and the underlying formal semantics, together with consistency issues between the three languages will be documented in the subsequent deliverable D13.2.

The language extensions defined for SDL 2000,MSC 2000 and TTCN-3, are currently being appraised through the development of prototype tools which are applied to the real-time protocols studied in WP3. The results obtained from the detailed investigation of the language extensions and prototype tools will be documented in the upcoming INTERVAL deliverable

D32. It is expected that the feedback gained from the prototypes and the applications will lead to a stable situation regarding all timed extensions by next October.

2. References

[0] INTERVAL Report: *Interest Group Meeting on Timed Extensions for SDL, MSC, TTCN*. ETSI/MTS, Sophia Antipolis. Reference IST/11557/WP1/11.00/TTT/012, November 2000.

[1] INTERVAL Deliverable M12: *Preliminary Specification of Timed Extensions*. Reference IST/11557/WP1/11.00/ITM/M12, November 2000.

[2] Marius Bozga, Susanne Graf, Laurent Mounier, Iulian Ober, Jean-Luc Roux, Daniel Vincent: *Timed Extensions for SDL*. In Proceedings of 10th SDL Forum, Copenhagen, June 2001. Lecture Notes in Computer Science series 2078, pp.223-240, Springer Verlag.

[3] Dieter Hogrefe, Beat Koch, Helmut Neukirchen: *Some Implications of MSC, SDL and TTCN Time Extensions for Computer Aided Test Generation*. In Proceedings of 10th SDL Forum, Copenhagen, June 2001. Lecture Notes in Computer Science series 2078, pp. 168-181, Springer Verlag.

[4] Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen: *Real-time test specification with TTCN-3*. Extended paper abstract accepted for presentation at the Workshop Formale Beschreibungstechniken (FBT 2001), Bruchsal, June 2001.

3. Documents part of the Deliverable

The papers referenced [2], [3] and [4] above, form the contents of this deliverable. They are reproduced in the following pages.

Timed Extensions for SDL*

Marius Bozga[1], Susanne Graf[1], Laurent Mounier[1], Iulian Ober[2], Jean-Luc Roux[2], and Daniel Vincent[3]

[1] VERIMAG, Centre Equation, 2 avenue de Vignate, F-38610 Gières
[2] Telelogic Technologies, 150 rue Nicolas Vauquelin, F-31106 Toulouse Cedex
[3] France-Telecom R&D, DTL, 2 avenue Pierre Marzin, F-22307 Lannion Cedex

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

1 Introduction

The ITU–T Specification and Description Language (SDL, [10]) is increasingly used in the development of real-time and embedded systems. For example many recent telecommunication protocols (such as RMTP-II [18] or PGM [17]) integrate such real-time features in their architecture, and these non-functional aspects are essential in the expected behaviour of the application. This kind of systems imposes particular constraints on the development language, and SDL is a suitable choice in many respects: it is formal, it is supported by powerful development environments integrating advanced facilities (like simulation, model checking, test generation, code generation, etc.), and thus it can cover several phases of the software development, ranging from analysis to implementation and on-target deployment.

It appears however that several important needs for a real-time systems developer are not covered by SDL. These problems range from pure *programming issues*, like the lack of useful primitives commonly available in real-time operating systems, to *specification issues*, like the difficulty to describe in a appropriate way the assumptions under which the system is supposed to be executed. Clearly, the needs are not

* This work is supported by the INTERVAL IST-11557 European project on timed extensions for SDL, MSC and TTCN.

the same for both uses of the language, and, in many cases, the *programming side* has been given priority in the supporting tools to the detriment of the *specification side*.

Several proposals already exist to extend SDL with real-time features. We can mention for example the work carried out on performance evaluation [8, 13, 15, 12], on schedulability analysis [4], or on real-time requirements [11]. In this paper we are more concerned with the use of SDL as a specification language for real-time systems and its application for formal validation. In particular, one of the important questions we address is what kind of real-time features should be modeled in SDL and at which level of abstraction.

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

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

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

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

The remainder of the paper is organized as follows: in section 2, we give an overview on the problems occurring when using SDL for real-time systems, concerning both programming and specification aspects, in section 3 we propose extensions allowing to improve SDL as a real-time specification language and in section 4 we propose necessary programming concepts. All new concepts are illustrated by examples illus-

trating their use and proposed syntax. Finally, in section 5 we draw some conclusions and give some perspectives.

2 Real-time SDL: what is missing?

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

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

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

2.1 Real-time semantics

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

A simulator that would use the semantics of time as described in Z.100, would not be able to make any assumption on the way time progresses, and therefore many unrealistic executions will be present in the resulting graph. As a result, the simulator would not guarantee elementary properties like *when a timer expires, it will be consumed by the concerned process in a* reasonable *amount of time* (whatever the notion of reasonable is). Even worse, according to some previous version of Z.100, *when two timers are set on the same transition (for example as two consecutive assignments), the timer with the lower delay is not always consumed first.*

In practice, existing simulation and verification tools have foreseen means for limited control over time progress. However, the control over time they propose is in general quite limited and moreover these annotations are tool dependent whereas they are totally part of the specification in the sense that they describe assumptions on the

system environment. The fact that designers and design languages neglect a clear description of relevant properties of the environment in which the system should be executed, is a frequent source of errors.

2.2 How to note non-functional aspects?

The development of a complex real-time protocol usually needs to consider several preliminary stages, during which some abstract or incomplete descriptions are produced. In order to properly validate (and document) these early designs, general assumptions on both the "environment" of the system and on its "non-functional" aspects have to be taken into account. For example, such assumptions concern:

- the expected duration of some internal task (which might be either informal or fully specified),
- the periodicity of some inputs triggered by the environment,
- or even the expected behaviour of the communication channels used within the system (these channels may be reliable or not, assumptions may be made on communication times, etc.).

Of course, some of these assumptions can already be partly included in the specification, either directly in SDL (e.g., using timers for explicit waiting) or using some separate formalisms offered by the verification tools (like the GOAL language [2] proposed in *Object*Geode to specify external observers). However, none of these two solutions is satisfactory: the first one leads to a specification in which external and non-functional assumptions do not appear as such, and this is obviously not desirable for code generation (these timers need not to be implemented), whereas the second one is restricted to a particular tool. Our objective is to provide a more suitable framework, based on standardized *annotations* on SDL specifications and compatible with the real-time semantics we propose.

2.3 High level synchronisations and other real-time primitives

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

Several synchronisation mechanisms that are commonly employed in real-time systems should be usable as concept at the SDL level. In particular, SDL timers are rather limited: the only available primitives are **set** and **reset** operations, the **active** function (which allows to determine if a given timer is running or not), and timeouts are always transmitted in the form of signals in the input buffer of the process.

2.4 Deployment information

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

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

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

- First of all, there exist time dependent properties which are not architecture dependent: for example, often the safety of a protocol may depend on the relative values of a set of timers, the expiration of which are used as implicit signal between processes. This is a common use of timers which cannot be expressed in the present SDL time semantics and still does not need any architecture indication.
- In a system where time is consumed either in communications or in requests to external systems (like a distant data base or anywhere else in the environment), the execution times will not depend on the mapping of processes on processors whenever the execution time of all activities consuming a negligible amount of time can be safely simplified to zero. Also in the case where time is consumed within the system, but within a single process per processor, analysis of execution time is still possible in the same way.
- In the case where time is consumed in several parts of the system, it would be sufficient to indicate which parts of the system are executed in parallel and which ones are not. The distinction between block agents, process agents and sub-agent gives some limited possibility to indicate such an architectural information.

Going one step further, scheduling policies also can influence the properties of the system in critical hard real-time systems. Moreover, there exists important advances in the synthesis of schedulers [3], where scheduling policies are expressed mainly in terms of dynamic priorities.

3 Extending specification aspects

As we mentioned above, what is missing in SDL to improve real-time systems *specification* is both a flexible time semantics together with more facilities to express

some "non functional" parts of the system or its environment. Our proposal is to solve these two problems uniformly – from the user point of view – by offering the possibility to *annotate* the specification in a standardized way. In particular, we propose to distinguish between two types of annotations:

- *assumptions*, which express *a priori* knowledge or hypotheses about the environment or the underlying execution system (system architecture, scheduling policy, etc). The use of assumptions is twofold: first, they might be necessary for the verification of properties which do not hold otherwise. Second, they might be used for code generation, both to guide some implementation choices or to add specific code in order to check their correctness at run-time or during the test phase.
- *assertions*, which express expected (local) properties on the system components. Such properties have to be proved on the specification, during the verification phase, possibly taking into account some of the assumptions.

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

3.1 Urgencies

A very abstract – and still very powerful – manner for making realistic assumptions on the time environment of a system is by means of transitions urgencies [5]: a transition is "urgent" if it is enabled and will be taken or disabled before time progresses. Three types of urgency assumptions (*eager*, *lazy* and *delayable*) are enough to control the progress of time with respect to the progress of the system:

- **eager** transitions are urgent as soon as they are enabled: they are assumed to be executed "as soon as possible"; in a simulation state *time does not progress* as long as there are enabled eager transitions.
- **lazy** transitions are never urgent: enabled **lazy** transitions do not inhibit time progress in any simulation state.
- **delayable** transitions are a combination of eager and lazy transitions: they become urgent when time progress disables them. They are supposed to be executed within some interval of time in which they are enabled. A delayable transition usually has an enabling condition depending on time, such as $\mathbf{now} \leq x$ or $\mathbf{now} - x \leq y$ (where x and y are numerical values) and *time may progress* in all simulation state in which this transition is enabled *as long as* $\mathbf{now} \leq x$ (or $\mathbf{now} - x \leq y$). When the extreme point of the interval is reached the transition becomes urgent.
  Notice that the attribute "delayable" is not primitive: a delayable transition with a guard $\mathbf{now} < x$ (for instance) can always been replaced by two transitions, a lazy one with the same guard and an eager one with guard $\mathbf{now} = x$. In particular, in SDL specifications in which explicit time guards (others than timeouts) are not used, explicit delayable transitions are not useful. However, whenever a task or a communication is assumed to take some time specified by an interval this is expressed by a delayable transition in the semantics model.

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

- one would like to consider some of the inputs as lazy (which is the standard point of view). When such a transition is enabled, the system can choose to react immediately or to wait. In the case of an *external* input laziness denotes the absence of knowledge about the possible arrival time of the input; for *internal* inputs laziness can be used to interpret internal action durations or internal propagation delays as unconstraint or unknown.
- On the other hand, one would like to consider some of the inputs as eager to express that the system cannot ignore them when they are enabled and must react immediately without waiting. This can be used to guarantee an immediate response to *critical* events such as timeout expirations or other prioritary inputs. Some care must be taken here since one can easily write system specifications in which time is blocked "forever" because at least one eager transition is always enabled (this is called a *Zeno behaviour*). This is particularly problematic when all transitions are assumed eager.

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

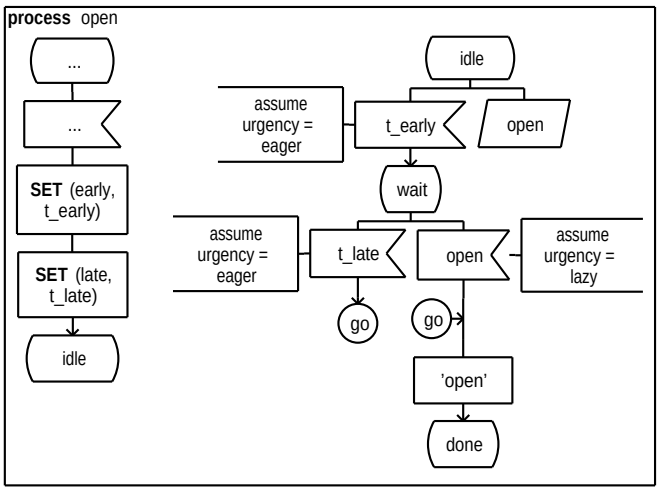

Fig. 1. Urgency assumptions - an example.

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

Timer semantics Timers are the most used primitive to observe the time progress. The semantics of timer expirations has been revisited in the last version of Z.100. The behaviour of a timer, can be sketched by the automaton given in Figure 2. Basically, it switches between *inactive* and *active* states depending on the set and reset actions performed on it. When active, once the expiration time is reached, it will *expire* and the timeout signal becomes available to the corresponding process instance. Previous version of Z.100 considered this expire transition as *lazy*, which means that one cannot make any assumption on the maximal time elapsed since its last setting. In other words, nothing can prevent the automaton from remaining in the *running* state after the expiration time.

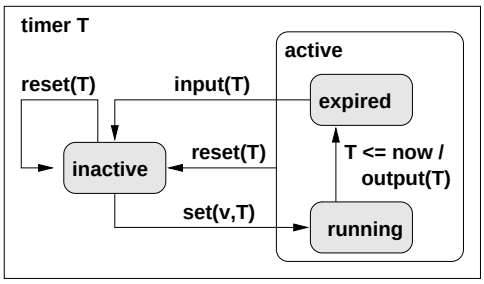

Fig. 2. Timer behaviour.

The current semantics introduces a more reliable timer concept ensuring the availability of the timeout signals *exactly* at expiration time. This can be done simply by considering the implicit expiration transition as *eager*. Note that this cannot be captured by the urgency annotation described before since it relies on an implicit semantic choice which cannot be explicited at the specification level. Also, this is

not too restrictive: it still allows the situation in which the timeout is really consumed at a later point of time as the consumption time depends on the urgency of the consuming transition.

Example 2. A second generic process is illustrated in Figure 3. The informal actions *action_1*, *action_2*, *action_3* must be executed *precisely* at moments, respectively *T0-d1*, *T0-d2*, *T0-d3*, where *T0* is given as parameter to the process. Let us assume first that the informal actions correspond to external commands which need just to be initiated by the process e.g, actioning some external devices. That means it is reasonable to assume that the process is essentially idle, wiating for the timer expiration, and reacts imediately. Unfortunately, even if the SDL description of the process seems intuitive and concise, based on the standard semantics of SDL, no assumption can be made about the time at which the actions will be executed. Considering the timeout transitions as lazy, the actions are executed *not earlier than* the required moment. The eagerness of the timeout consumption transitions expresses the assumption that actions are executed at the moment at which the timer expirations are available.

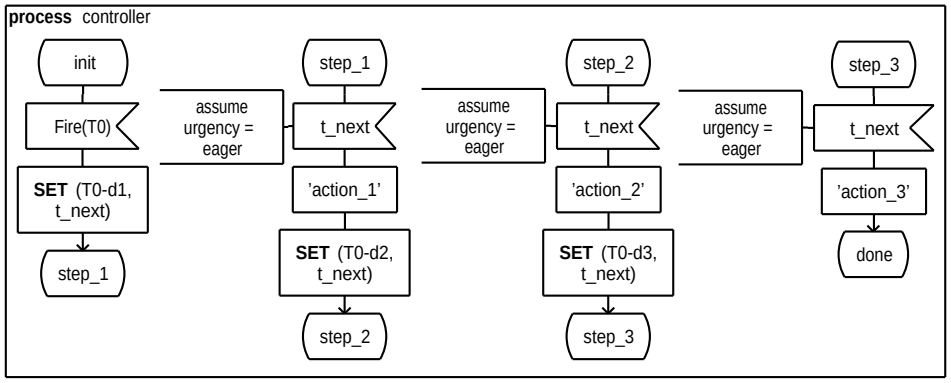

Fig. 3. Timeout urgencies - an example.

3.2 Durations

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

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

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

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

3.3 Periodicity

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

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

Example 3. A typical producer/consumer system is illustrated in Figure 4. The producer reacts to external *request* signals and sends *data* signals to the *consumer*. When the *consumer* receives *data*, it sends an acknowledgment back to the *producer*. Several annotations are used here. First, we assume that *request* signals come from the environment at the given rate of one signal every 10 or 11 time units. Also, both data production and consumption times are supposed to be not negligible, the former being between 5 and 7 and the second between 4 and 5 time units.

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

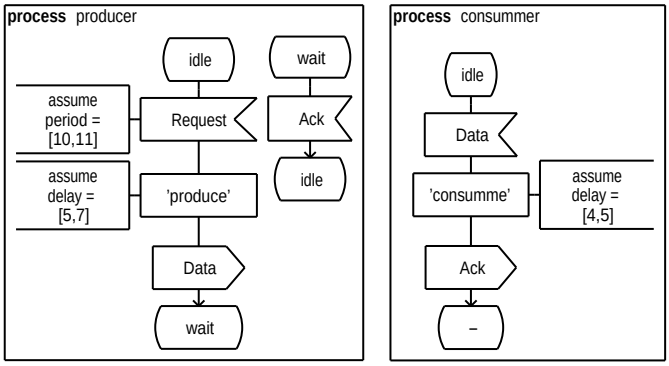

Fig. 4. Delay and periodicity assumptions.

3.4 Flexible channel specifications

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

Thus, by flexible channel specification we mean typically that messages can be lost, reordered or delayed to some extent, leading to a well-defined set of channel types. It is possible to describe any of these channel types by means of additional SDL processes, but it is not necessarily desirable to do so, for several reasons. First of all, these processes serve only for simulation and not for implementation. Also, in some (rare) cases it may be problematic as the transported signals do not carry the pid of the original sender anymore. Finally, the fact to have a predefined set of channel attributes is an advantage for simulation and verification tools as this knowledge can be directly exploited by appropriate techniques and lead to more efficient algorithms (notice for example that an interesting set of properties is decidable for finite state machines communicating through lossy channels whereas they are undecidable in case of communication through reliable channels [1]).

We propose annotations on channels allowing to specify a propagation delay in a similar way as execution times and distinguish between fully reliable (which never lose messages) and unreliable channels (where arbitrary losses are possible), and between ordered and unordered ones. Here again, for performance evaluation purposes, it is possible to extend these annotations to include a probability law which specifies a distribution of message losses or a degree of reordering.

Example 4. Figure 5 gives an example of use of annotations to denote propagation delays and reliability of channels. Notice that the default option for a channel is to be ordered and reliable in conformance to the standard SDL semantics.

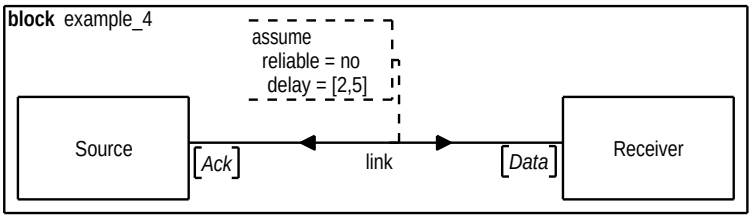

Fig. 5. Channel assumptions.

4 Extending programming aspects

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

4.1 Extension of timer concepts

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

Interruptive timers and signals SDL timeouts are always received in the form of asynchronous signals. For general-purpose time dependent code this is usually fine, but it is difficult to write real emergency procedures using asynchronous timeouts. To ensure that a piece of code is executed immediately, or in a specified interval, after the expiration of a timeout, the SDL designer must first make sure that the corresponding agent (process) is idle when the timeout message is received, otherwise, the agent may consume the timeout message from the input queue only when it finishes its current job, which may be too late. This obliges to artificially restructure the system, whereas in the implementation this task can be left for implementation where it even may be useless because of the existence of interrupts. Therefore, SDL needs a notion of emergency timer, whose expiration is taken into account *immediately* 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. What we need is an extension of this exception mechanism to be triggerable by system time.

We propose to define interruptive timers using an optional attribute (called *interruptive*). The behaviour of interruptive timers will rely on an extension of the exception mechanism already existing in SDL. More precisely, when an interruptive timer expires, instead of sending a timeout signal via the input port of the process, it will raise a timeout-exception in the concerned process. The handling of this exception is left to the user. However, special care is required to clarify what happens

if an interruptive timer wants to interrupt a transition which is required atomic. In particular, when an interruptive timer expires in a service, while another service (within the same process) is running and executes a time-consuming job.

Example 5. In Figure 6 we illustrate the use of an interruptive timer. When set, it receives as parameter the maximal allowed execution time T. Then, the process enters a loop executing some time-consuming job *"decode"*. Two cases are possible: either the entire job is finished in time, or the maximally allowed time is reached before, when the process is processing a *"decode"*. In the second case, an interruptive timer *alarm* is needed in order to break the normal execution flow, i.e, to abort the execution of *decode* and to stop the process immediately.

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

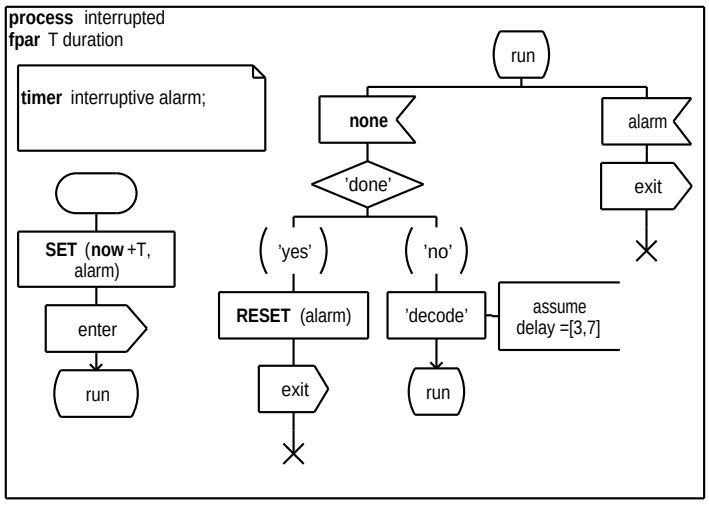

Fig. 6. Interruptive timer example.

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

Such a primitive reduces the current distinction made in SDL between timers and other variables. In particular, the value of the timer can be passed via communi-

cation signals from a process to another, used to compute the remaining duration until its expiration, or used to set another timer depending on it, etc.

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

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

4.2 Atomicity

There are no atomicity and synchronization primitives foreseen in SDL, the idea being that these are implementation details which should not be mentioned at SDL level. The fact is that in implementation oriented SDL descriptions, they are necessary and several SDL users have expressed the need for native SDL constructs for synchronization [9], especially to achieve atomicity and mutual exclusion. The reason is that such constructs are often used in the specification and the implementation of real-time systems such as those developed with SDL.

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

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

Example 6. For example, a semaphore may be specified in SDL as a process type exporting two (empty) procedures P and V, implementing the usual operations on semaphores. The specification of such a semaphore type is shown in figure 7. The only prerequisite for this implementation to work is that the atomicity of P and V are preserved for a same instance of semaphore. This prerequisite is ensured by the execution semantics of SDL. Moreover, concurrent wait operations P which arrive

after the semaphore is already blocked on a wait are implicitly sequentialized by the save operation from the *busy* state.

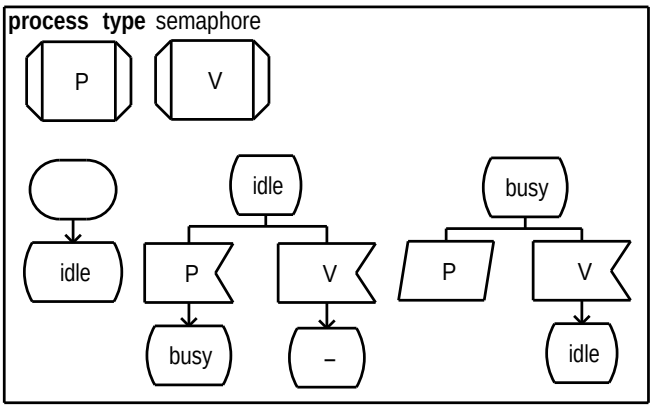

Fig. 7. Semaphore example.

5 Conclusion and perspectives

We have defined a number of real-time extensions of SDL in order to obtain a specification and modeling language really adequate for real-time systems based on general ideas and reflections we have proposed in [7]. The proposed extensions include:

- a more flexible time-progress semantics for SDL, including as particular cases both the time progress semantics of Z100 and the time-progress semantics used in verification tools;
- powerful and flexible annotations allowing to express non-functional aspects: both very abstract ones necessary at an early design stage, and concrete ones useful for code generation, code annotation and performance analysis;
- some primitives which allow to make specifications simpler, easier to understand, and to better separate the implementation oriented and specification oriented features;
- implementation oriented features like emergency timeouts which are mandatory for the use of SDL as a real-time modeling language.

These extensions have been submitted as a first draft to the ITU-T (Study Group 10) within the framework of Question 7 *Time expressiveness and performance annotations on ITU-T modeling languages*. All these extensions are based on a unified and sound semantic framework which can be integrated smoothly into the existing semantics of SDL in which time progress is mainly "unconstraint". In particular,

almost all of them require only local restrictions of the current non-deterministic time semantics.

We have already started to extend existing SDL tools in order to deal with these extensions and they already proved their usefulness in real applications. We have implemented a translation of SDL with our proposed time extensions into communicating timed automata with urgencies which are the input language of the IF toolset [6]. IF has been developed at VERIMAG for the purpose of prototyping timed extensions of SDL-like languages. In particular, it includes a model-checker based on the Kronos-tool [19] allowing to verify quantitative time requirements. In addition, the proposed timed extensions have been implemented also in the *Object*Geode simulator [14]. For both tools we obtain good results, both in what we can express and in what analysis we can perform on annotated models. As an example, we are currently modeling real-time multicast protocols [17, 18] using these annotations.

The main approaches which need to be compared with ours are QSDL[8, 13] and UML related real time extensions (such as UML-RT[16]). QSDL (Queuing SDL) is a performance analysis oriented extension of SDL using a resource mapping and annotations of tasks in terms of "computation power" from which execution and waiting times are computed depending on some scheduling policy which can be user defined. The underlying semantics considers all non annotated transitions as taking zero time and all transitions are considered urgent, respectively delayable if they may take variable amount of time. In so far, this framework is compatible with ours, but is exclusively performance analysis oriented.

The foreseen real-time extensions of UML are all of a very syntactic nature. It is not clear if there will be a semantic time model at all. However, it is interesting to note that non-functional annotations concerning Quality of Service (such as throughput of channels, execution times,...) are planned and can take likewise the form of *requirements* and *assumptions*.

The proposed extensions are not completely satisfactory from the user point of view. More extensions may well appear to be useful. In particular, there is a strong need for an expressive notion of *deployment diagram*, allowing to define the mapping of processes to resources, scheduling policies and QoS annotations. The proposed annotations on the SDL level will then have 3 different sources: some of them may be user defined, especially at an early design stage; some of them will be generated from information extracted from such a deployment diagram (which goes far beyond the resource mapping of the QSDL proposal); finally, some will be obtained from analysis results of other parts of the system, especially in a compositional verification approach, which is the only one able to deal with large specifications.

References

1. P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In A .Hu and M. Vardi, editors, *Proceedings of CAV'98 (Vancouver, Canada)*, volume 1427 of *LNCS*, pages 305–318. Springer, June 1998.
2. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *Proceedings of SDL FORUM'95*. Elsevier, 1995.

3. K. Altisen, G. Gößler, and J. Sifakis. A Methodology for the Construction of Scheduled Systems. In Mathai Joseph, editor, *Proceedings of FTRTFT 2000*, number 1926 in LNCS, pages 106–120. Springer-Verlag, September 2000.
4. J.M. Alvarez, M. Diaz, L.M. Llopis, E. Pimentel, and J.M. Troya. Integrating Schedulability Analysis and SDL in an Object-Oriented Methodology for Embedded Real-Time Systems. In R. Dsoulli, G.v. Bochmann, and Y. Lahav, editors, *Proceedings of SDL-FORUM'99 (Montreal, Canada)*, pages 241–256. Elsevier, June 1999.
5. S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *International Symposium: Compositionality - The Significant Difference (Holstein, Germany)*, volume 1536 of *LNCS*. Springer, September 1997.
6. M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 (Toulouse, France)*, volume 1708 of *LNCS*, pages 307–327. Springer, September 1999.
7. M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for Real-Time: What is Missing ? In *Proceedings of SAM'00: 2nd Workshop on SDL and MSC (Grenoble, France)*, pages 108–122. IMAG, June 2000.
8. M. Diefenbruch, E. Heck, J. Hintelmann, and B. Müller-Clostermann. Performance evaluation of SDL systems adjunct by queueing models. In R. Braek and A. Sarma, editors, *Proceedings of SDL Formu'95*. Elsevier Science B.V., 1995.
9. Interval Consortium. Requirement Analysis Report. Technical Report D11, Interval Deliverable, October 2000.
10. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, November 1999.
11. S. Leue. Specifying Real-Time Requirements for SDL Specifications – A Temporal Logic-Based Approach. In *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing, and Verification PSTV'95*. Chapmann & Hall, 1995.
12. M. Malek. PerfSDL: Interface to Protocol Performance Analysis by means of Simulation. In R. Dsoulli, G.v. Bochmann, , and Y. Lahav, editors, *Proceedings of SDL-FORUM'99 (Montreal, Canada)*, pages 441–455, 1999.
13. A. Mitschele-Thiel and B. Müller-Clostermann. Performance Engineering of SDL/MSC Systems. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in SDL and MSC (Erlangen, Germany)*, February 1998.
14. I. Ober, B. Coulette, and A. Kerbrat. Timed SDL Simulation and Verification - Extending SDL with Timed Automata Concepts. Technical report, Telelogic Technologies Toulouse, 2000.
15. J.-L. Roux. SDL Performance Analysis with ObjectGeode. In A. Mitschele-Thiel, B. Müller-Clostermann, and R. Reed, editors, *Workshop on Performance and Time in SDL and MSC (Erlangen, Germany)*, February 1998.
16. B. Selic and J. Rumbaugh. Using Uml for Modeling Complex Real-Time Systems. Whitepaper, Rational Software Corp., March 1998.
17. T. Speakman, D. Farinacci, S. Lin, and A. Tweely. PGM Reliable Transport Protocol Specification. Internet draft, IETF, 1999.
18. B. Whetten, M. Basavaiah, S. Paul, T. Montgomery, N. Rastogi, J. Conlan, and T. Yeh. The Reliable Multicast Transport Protocol, version 2 (RMTP-II). Internet draft, IETF, 1998.
19. S. Yovine. KRONOS: A Verification Tool for Real-Time Systems. *Software Tools for Technology Transfer*, 1(1+2):123–133, December 1997.

Some Implications of MSC, SDL and TTCN Time Extensions for Computer-aided Test Generation

Dieter Hogrefe, Beat Koch, and Helmut Neukirchen

Institute for Telematics, University of Lübeck
Ratzeburger Allee 160, D-23538 Lübeck, Germany
Tel.: +49 451 500 3721 Fax: +49 451 500 3722
{hogrefe,bkoch,neukirchen}@itm.mu-luebeck.de

Abstract. The purpose of this paper is to describe how computer-aided test generation methods can benefit from the time features and extensions to MSC, SDL and TTCN which are either already available or currently under study in the EC Interval project. The implications for currently available test generation tools are shown and proposals for their improvement are made. The transformation of MSC-2000 time concepts into TTCN-3 code is described in detail.

1 Introduction

Computer-aided test generation (CATG) from system specifications has been an active field of research for many years [1, 5, 10, 24]. This research has resulted in the development of a number of test generation tools [2, 8, 9]. Today, two industrial-strength, commercially available CATG applications exist [6, 18]. These tools take formal system specifications with the 1992 edition of the *Specification and Description Language (SDL-92)* and test purpose descriptions with the 1996 version of *Message Sequence Charts (MSC-96)* as input and produce test suites based on the second edition of the *Tree and Tabular Combined Notation (TTCN-2)* [14].

Meanwhile, the standards of both SDL and MSC have been updated (SDL-2000 [16], MSC-2000 [15]) and a thoroughly new version of TTCN has been standardized (TTCN-3 [7]). In addition, the European Commission has set up the *Interval* project [21] to prototype an SDL, MSC and TTCN-based tool chain for the development and testing of systems with real-time constraints. During the first project stage, the Interval consortium identifies constructs which are suitable for capturing, specifying, modelling and testing real-time requirements. Based on these constructs, the consortium proposes time extensions to the formal languages as ITU-T recommendations. In the second project stage, tools will be developed which include the new time constructs.

Taking existing test generation tools as reference implementations, this paper evaluates the implications of existing and proposed time extensions to CATG. It is structured as follows: Section 2 shows when and why time constructs are

needed during testing. Section 3 contains an overview of the timer concepts in MSC, SDL and TTCN. In Section 4, timer support of the test generation tools TestComposer and Autolink is discussed. Section 5 is the main part of this paper. It examines first how the test generation process may be improved through the use of SDL-2000 together with the proposed extensions. Second, the benefits of using MSC-2000 for test purpose description are shown and a concrete mapping of MSC-2000 time concepts to TTCN-3 is presented. Section 6 concludes this paper.

2 Timer in Test Purpose Descriptions

Timers in test sequences have one of the following purposes:

- assuring that test cases end even if they are blocked due to unexpected behavior of the system under test (SUT);
- checking constraints on the response time of the SUT;
- delaying the sending of messages to the SUT in order to
  - allow the SUT to get into a state where it can receive the next signal (if the tester is too fast);
  - check the reaction of the SUT if a signal is delayed too long (invalid behavior specification);
  - check that the SUT does not send any signal for a given amount of time.

To guarantee the conclusion of a test case, one or more *global timers* are used. In case of a single-tester test architecture, one timer is started at the beginning of test case execution. Its duration is chosen to be longer than the expected execution time of the test case. At the end of each possible test sequence, the timer is reset. In the exception handling section of the test case, the timeout of the global timer is caught and handled. If a distributed test system is used, a global timer is started within each test component participating in the test execution. In case of a timeout in any test component, the other test components have to be notified in order to let them conclude the test execution gracefully.

Time constraints are checked through the use of one or a pair of *guarding timers*. Guarding timers are started when a signal is sent to the SUT. If a lower bound is specified in the time constraint, one timer has to expire before the response signal from the SUT is received. The second timer — which checks the upper bound of the time constraint — is reset immediately upon reception of the response signal. Premature reception of the response signal or the expiration of the second timer are caught in the exception handling section of the test case and result in a FAIL verdict.

A *delaying timer* is specified by inserting a timer start operation immediately followed by a timeout event into the test sequence.

3 Timer in Formal Languages

The formal languages MSC, SDL and TTCN all contain timer support. In this Section, an overview of the timer concepts of these languages in given.

3.1 MSC-96

Timer support in MSC-96 [13] is very basic: there exist events to *set* and *reset* a timer, and a *timeout* event. Timer events are identified by a mandatory timer name and an optional timer instance name. The specification of a timer duration is optional; if it is specified, it has no semantics. Pairs of timer set and reset/timeout events must be specified on the same MSC instance.

3.2 MSC-2000

MSC-2000 [15] supports the same basic timer events as MSC-96, with some changes and refinements. First of all, the *set* event has been renamed to *start-timer* and *reset* is now called *stoptimer*. If a duration is specified, then it must be done in the form of an interval with an optional lower bound (default value: zero) and an optional upper bound (default value: infinite). This means that the timer can expire within the specified period.

In addition to the basic timer concepts, MSC-2000 also provides a timed semantics for constraining and measuring the time of events. (However, a formal semantics for MSC-2000 is still missing.) Using the external data language approach introduced in MSC-2000, variables of type *Time* may be declared. There are two operators to measure time and store it in time variables: one to determine the absolute time at the moment of the execution of a given event, and one to determine the amount of time which passes between two events. It is also possible to specify time constraints: the lower and upper bound of a time interval between a pair of events may be defined in order to specify the allowed delay between those events. For a single event, the absolute time of occurrence can be constrained, too.

An extension to the MSC-2000 standard has been proposed by the Interval consortium to ITU-T Study Group 10 in [20]. A new symbol is proposed to express periodic occurrence of repetitive events which are folded into a loop.

3.3 SDL-2000

In SDL-2000 [16], timer declarations are mandatory. As part of the declaration, a constant default duration may be defined. With the *set* statement, a timer is activated. With the *reset* statement, a timer is put back to the inactive state. If an active timer expires, a signal with the same name as the timer is put into the input queue of the process which contains the timer. This corresponds to the *timeout* event in the MSC language. Whereas timers in SDL-2000 and MSC-2000 are basically equivalent, the new time constraint concept of MSC-2000 has no equivalent in standard SDL-2000.

Timer handling has been a weak point of SDL since its first introduction and there has been no improvement with the publication of SDL-2000. Therefore, several timer and time semantics related extensions to the SDL standard have been proposed by research groups [19] and Interval consortium members [3, 4, 11]. The latter proposals include

– the addition of cyclic timers which are automatically restarted after expiration;
– mechanisms to read a timer value;
– the introduction of interruptive signals and timeouts.

Furthermore, a real-time semantics is introduced. This semantics allows to assign urgencies to transitions and to model time progression caused by actions which are annotated with a corresponding assumption on time consumption.

3.4 TTCN-2

In TTCN-2 [12], there are three timer operations: the common *START* and *CANCEL* operations to activate and deactivate a timer, as well as the *READ-TIMER* operation which returns the amount of time which has passed since a timer has been activated. Timer expiration is caught with the *TIMEOUT* event.

There are several problems with the implementation of timers in TTCN-2. First, timers have to be declared at test suite level. According to the standard, a full set of timers must be allocated for each test component, potentially wasting scarce hardware resources. The second problem concerns the applicability of TTCN-2 to the testing of real-time time constraints: timeout events are stored in a list until they match an alternative in the test sequence. As a consequence, timeout events may remain unnoticed for some time. This in turn may lead to incorrect test execution and verdict.

Moreover, due to the snapshot semantics of TTCN, it has to be noted that when using the existing timer concepts, a coherent and valid test verdict for real-time tests can only be found if the test equipment is reasonably fast. Since the snapshot semantics may summarize time-critical events arriving at different queues into one snapshot, important timing or ordering information might get lost. In this case, it is not decidable whether a violation of real-time constraints has occurred or not. The test verdict will rather depend on the question of how the triggering events of an alternative are ordered in the TTCN dynamic behavior description.

To solve this problem, a refinement of the standard snapshot semantics is proposed in [25]. Instead of testing time constraints using standard timers, additional columns for earliest and latest execution times of TTCN events are proposed. Since this way of specifying time constraints is orthogonal to the evaluation of alternatives, the test verdict does not depend on the speed of the tester or the ordering of alternatives.

Nevertheless, even with [25] the test system has to be fast enough in order to avoid the overflow of input queues. Therefore, sufficient processing capabilities of the tester are in any case a necessary prerequisite of real-time testing.

3.5 TTCN-3

TTCN-3 [7] renames some of the timer operations of TTCN-2: the keyword to deactivate a timer is now *stop* and the elapsed time of an active timer can

be queried with the *read* operation. In addition, the *running* operation returns *true* if a given timer is running, *false* otherwise. The *start* operation and *timeout* event remain unchanged. Timer functionality is also included in the synchronous *call* operation. A timeout value may be provided as an optional parameter to this operation. If a timeout occurs, it may be handled as an exception with the *catch* operation.

No concrete proposals have been published so far regarding the extension of time concepts in TTCN-3. However, since TTCN-3 uses the same snapshot semantics as TTCN-2, the weakness of this semantics concerning real-time testing still holds for TTCN-3. A forthcoming proposal to overcome this problem is currently under study by the Interval consortium. Rather than using the standard timers to test real-time requirements, it is intended to separate the description of functional requirements (e.g. signal reception and "functional" timeouts) and non-functional (i.e. real-time) constraints. Since these extensions are currently under study, the test cases given in this paper are written using standard TTCN-3 notation.

4 Timer in Current Test Generation Tools

At the time of writing this paper, there are two major test generation tools on the market which take SDL-92 and MSC-96 specifications as input and produce TTCN-2 as output: TestComposer [18] and Autolink [6]. In this Section, the current status of these tools with respect to timer support is presented.

4.1 TestComposer

TestComposer automatically generates four types of timers during the computation of test cases:

- a timer TAC is set whenever a test component waits for a response from the SUT. A fail verdict is assigned in case of a timeout. With respect to timer purposes introduced in Section 2, TAC corresponds to a guarding timer;
- the timer $TWAIT$ is another guarding timer: it checks that time to execute an implicit send does not exceed a predefined amount of time;
- the timer $TNOAC$ is set to check that the SUT does not send a message to the tester for a specific amount of time. $TNOAC$ is a delaying timer;
- $TEMPTY$ is a delaying timer which is used to force a timeout in the SUT.

4.2 Autolink

Autolink generates the declaration of a global timer *T_Global* automatically. Depending on the test architecture, timer statements for *T_Global* are added to the test case behavior description and the top-level test steps of all parallel test components.

Guarding and delaying timers can be specified by the user in test purpose MSCs with timer set, reset and timeout events; these events are translated into corresponding TTCN-2 statements during test generation.

4.3 Discussion

With the methods available in the current test generation tools, the common cases for using timers in test cases can be handled fairly well. However, both tools do not offer optimal timer support. On the one hand, unnecessary timer events may be generated with the fully automatic method in TestComposer. These events have to be removed from the test suite manually. On the other hand, while Autolink offers complete flexibility regarding timers, the manual specification with MSC-96 may be laborious. This is especially true if an SUT response has to fall within a time interval: with the MSC-96 notation used by Autolink, two timers must be drawn, which increases the effort to specify the test purpose MSC and reduce its readability (see Figure 1). Neither tool supports the reading of timer values.

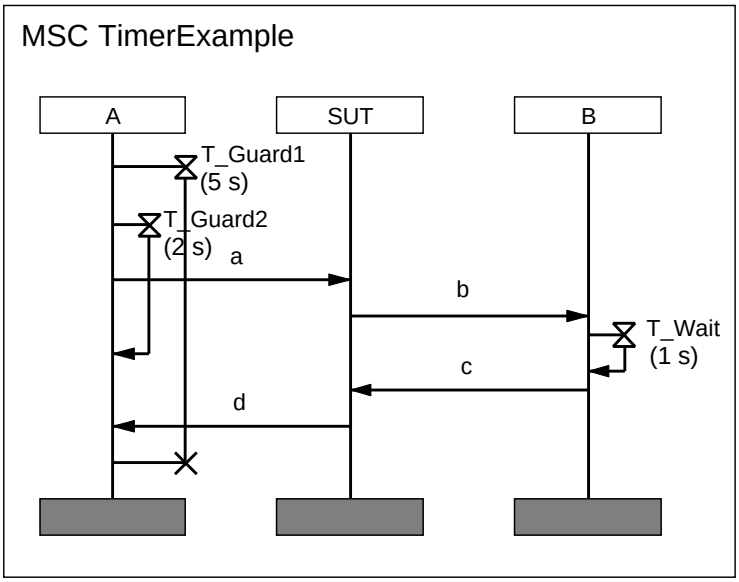

Fig. 1. Timer specification for Autolink with MSC-96

5 Improving the Test Generation Process

Both TestComposer and Autolink are test-purpose-based test generation tools. This means that they need a formal description of the test purpose which they can transform into a TTCN test case. The transformation is done either by direct translation from MSC to TTCN or by performing a state space exploration of

an SDL specification. Test purpose descriptions may be provided in the form of MSC-96 diagrams for both tools.

CATG tools may benefit from the use of formal languages with time extensions in a number of ways:

- reduction of the state space during exploration-based test generation with timed SDL;
- automatic generation of time requirements for test equipment with timed SDL;
- improvement of the capabilities to efficiently describe timing constraints in test purpose descriptions by using MSC-2000.

5.1 Test Case Generation with Timed SDL

The extensions proposed by the Interval consortium for SDL are mainly intended for verifying and validating a specification with respect to time properties. Nevertheless, automatic test generation benefits for two reasons from such time annotations.

First, the state space of a timed SDL model can be reduced in comparison to an untimed specification. The reason is that an untimed specification allows a lot of unrealistic scenarios which cannot occur in practice, because it contains paths where time does not progress at all. By using a real-time semantics and a timed SDL model, the state space can be reduced to the realistic scenarios. As CATG is mainly based on representing observable events of paths allowed by an SDL model, unrealistic test cases can be avoided.

Second, if additional timing information is given for all symbols contained in an SDL transition, the exact moment when observable events are allowed to take place can be determined. Test cases which take this information into account can be derived automatically. However, this topic is subject to further study. If additional timing information is not available for a whole transition, it is still possible to specify real-time requirements using MSC-2000. The usage of MSC-2000 for test description is shown in Section 5.3.

5.2 Generation of Time Requirements for Test Equipment

TTCN assumes that the test equipment is always fast enough to test the IUT. While this assumption is legitimate if only time non-critical functional behavior is tested, it may not hold for real-time applications. The processing speed of the tester may not be fast enough to keep track with the test events that happen at the PCOs. As an example, during the development of the GSM test suite at ETSI, there were various occasions where the possible lack of sufficient speed of the test devices had to be taken into account. In some cases, this problem was resolved by letting the tester respond to a signal from the SUT before it even receives the signal, just by assuming that the signal will arrive eventually. If the tester had to wait for the reception of the signal, it would be not fast enough to respond to it. While such workarounds are possible, they are problematic,

because the order of test events has to be changed. As a consequence, the prose test purpose description does no longer correspond to the formal description.

In general, it seems more feasible to require some speed of the tester and treat these requirements as part of the test suite. If this approach is taken, time constraints for the tester have to be defined somehow. This means that the tester is required to execute test events within a certain time interval in order to test the SUT accurately and successfully.

A very detailed idea about the timing behavior of the SUT is required to determine the time intervals between test events. The test designer or a test generation tools need to know at which points in time the SUT may be stimulated or events from the SUT may be observed. Traditionally, this timing information has not been part of the SDL specification. However, if such timing information is added to the SDL specification, the minimal time interval between test events can be derived which the test equipment must be able to process. Based on this information, it is possible to generate benchmarks for the tester. An example for a tester benchmark is given below using the TTCN-3 notation. This benchmark checks whether the test equipment is fast enough to send two consecutive messages within a duration specified by *required_time*:

```
1:  timer T;
2:  T.start(required_time);
3:  A.send(a);
4:  A.send(b);
5:  if (T.running)
6:  {
7:    T.stop;
8:    verdict.set(pass);
9:  }
10: else
11: {
12:   verdict.set(fail);
13:   MyComponent.stop;
14: }
```

5.3 Using MSC-2000 for Test Purpose Description

Figure 2 shows the test description of Figure 1 in MSC-2000 notation. The use of the time interval notation instead of four separate timer symbols (two set events, one reset and one timeout) to specify two guarding timers makes the diagram much more readable. Given a time interval where the start event is a send to the SUT and the end event is a receive from the SUT, the test generation tool has to perform the following actions:

1. Check the time interval to establish the number of timers which are needed to represent the interval with TTCN. If just one boundary value is specified, only one timer is needed. If both a minimal and a maximal time point are

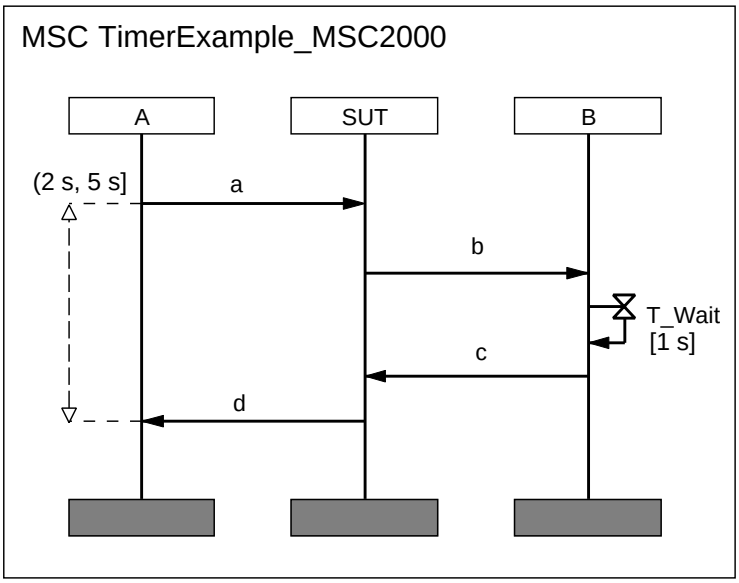

Fig. 2. Time constraint specification with MSC-2000

specified, two TTCN timers are needed. Since no timer name is specified with the time interval notation, the tool has to select the timer names by itself. Preferably, the user should be able to define timer name templates such as *T_Guard_Min* and *T_Guard_Max*. The tool then must check if timers with these name are in use already. If they are, new timers (e.g., *T_Guard_Min_2* and *T_Guard_Max_2*) must be declared. In order to minimize the number of timers which have to be declared, the test configuration has to be taken into account in this step.

2. If the time interval contains expressions with measurements (see Section 5.3), replace the time patterns with the corresponding variable identifier. If necessary, convert time values to seconds.
3. Create the appropriate TTCN timer statements. This step depends on the number of time points specified in the MSC. In the examples below, TTCN-3 is produced from the MSC in Figure 2, assuming that there is a test component handling just PCO *A*. A similar transformation can be done for TTCN-2.

If only the lower boundary value is specified, create the following statements:

```
1:  timer T_Guard_Min;
2:  A.send(a); T_Guard_Min.start(2);
3:  alt {
4:  []  T_Guard_Min.timeout;
5:  []  A.receive(d)
```

```
 6:      {  verdict.set(fail);
 7:         MyComponent.stop;
 8:      }
 9:  }
10:  A.receive(d);
```

The case where d is received prematurely by PCO A (lines 5 to 8) may as well be handled in a default. If only the upper boundary value is specified, create the following statements:

```
 1:  timer T_Guard_Max;
 2:  A.send(a); T_Guard_Max.start(5);
 3:  alt {
 4:  []  A.receive(d)
 5:      {  T_Guard_Max.stop; }
 6:  []  T_Guard_Max.timeout
 7:      {  verdict.set(fail);
 8:         MyComponent.stop;
 9:      }
10:  }
```

If the lower and the upper boundary values are specified, create the following statements:

```
 1:  timer T_Guard_Min;
 2:  timer T_Guard_Max;
 3:  A.send(a); T_Guard_Min.start(2); T_Guard_Max.start(5);
 4:  alt {
 5:  []  T_Guard_Min.timeout;
 6:  []  A.receive(d)
 7:      {  verdict.set(fail);
 8:         MyComponent.stop;
 9:      }
10:  }
11:  alt {
12:  []  A.receive(d)
13:      {  T_Guard_Max.stop; }
14:  []  T_Guard_Max.timeout
15:      {  verdict.set(fail);
16:         MyComponent.stop;
17:      }
18:  }
```

The case where d is received prematurely by PCO A (lines 6 to 9) may as well be handled in a default. TTCN-3 has a special notation to put a timeout value on a procedure call. If the start event of a timer interval in an MSC is a method call (cf. Figure 3), then the following code should be generated in order to guard the call by an upper time bound of e.g. 3 seconds:

```
1:  A.call(x, 3);
2:  {
3:    [] A.getreply(x);
4:    [] A.catch(timeout);
5:       {  verdict.set(fail);
6:          MyComponent.stop;
7:       }
8:  }
```

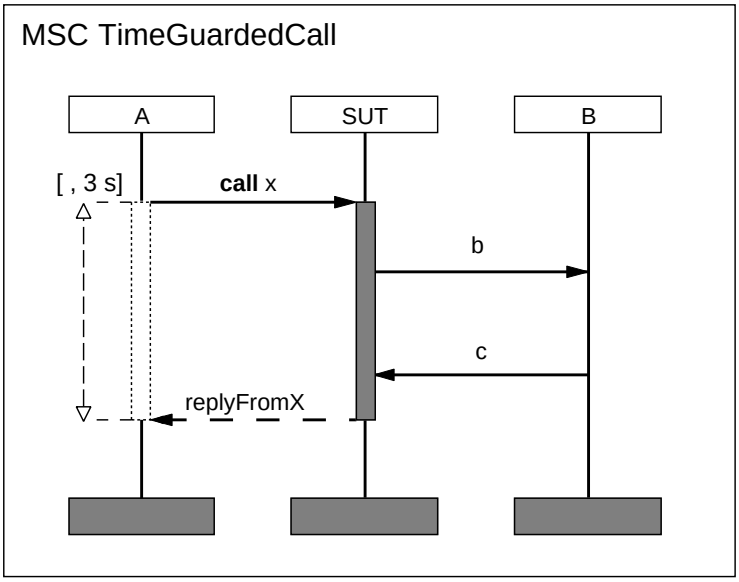

Fig. 3. Time constraint for a method call in MSC-2000

Time Measurement. In MSC-2000, time can be measured and stored in variables. These measurements can be reused, e.g. to specify time intervals. Figure 4 shows the two kinds of time measurements provided by language: *&t1* is a relative measurement; the time which passes between the sending of *a* and the reception of *d* is stored in a variable *t1* of type *Time*. *@t2* is an absolute measurement, which means that the value of an existing global clock is stored in a variable *t2* of type *Time*. The global clock is started when the first event in the MSC is executed.

Transforming a relative time measurement into TTCN is straight-forward. The test generation tool needs to declare a special timer used for the measurement. This timer is activated after the first event in the MSC has occurred. After

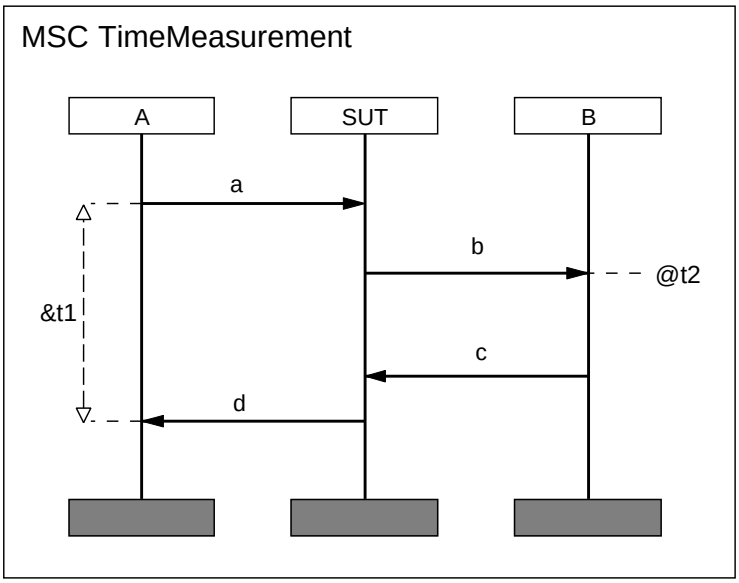

Fig. 4. Time measurements in MSC-2000

the second event, the timer is read and deactivated. The only problem is the fact that to start a timer in TTCN, a duration has to be defined, which in this case is not known in advance. As a solution, the test designer has to provide a maximum value for time measurement. Most likely, this value is available anyway because a timeout period has to be defined for a global test case timer. From the MSC in Figure 4, the tool should generate the following TTCN-3 statements:

```
1:  timer T_Measure := 10000;
2:  var float t1;
3:  A.send(a); T_Measure.start;
4:  A.receive(d); t1 := T_Measure.read;
5:  T_Measure.stop;
```

To measure an absolute time value, a global timer has to be started at the beginning of the test case. The measurement can then be done by using the *read* operation on the global timer. Below is the TTCN-3 code generated for the measurement of *t2*:

```
1:  timer T_Global = 10000;
2:  var float t2;
3:  T_Global.start;
4:  B.receive(b); t2 := T_Global.read;
5:  B.send(c);
```

MSC-2000 also allows to measure or to constrain the amount of time which passes between a pair of events on different instances. This kind of time interval cannot be represented with standard TTCN-3 timers, since the start and read or timeout operations of the same timer cannot be distributed between different parallel test components. If there is more than one test component, coordination messages might be used to synchronize the parallel test components concerning the two relevant events. However in this case, the time needed to transmit these coordination messages has to be taken into account.

6 Conclusion

In this paper, the current state of SDL, MSC, TTCN and test generation tools with regard to timer support has been presented. Commercially available test generation tools already allow to generate TTCN-2 test suites which reflect time requirements expressed by standard MSC-96 timers. However, due to the simple timer concepts of MSC-96, the specification of time constraints in test purpose descriptions may be quite laborious. It has been shown that the use of the MSC-2000 time interval notation can facilitate the specification of time constraints for events on the same instance. A translation of MSC-2000 time constructs into TTCN-3 code has also been presented. The mapping of MSC-96 timers to MSC-2000 time constructs and the transformation of TTCN-2 to TTCN-3 is straightforward. Therefore it will be possible to automatically generate TTCN-3 test cases which test the conformance to such MSC-2000 time constraints.

Nevertheless, many challenges remain. The testing of time constraints in a distributed test architecture has not been solved yet. Currently, it is neither possible to derive test cases in an automated way nor to test real-time requirements if several parallel test components observing time critical events are involved. Care has to be taken to synchronize these parallel test components not only in a functional manner but also with regard to their local clocks.

It has also not been shown yet that it is possible to generate real-time tests from time extended SDL models. The accuracy of automatically derived test cases depends on how exhaustively an SDL model is enriched with time annotations. Research by the Interval consortium will show whether this is feasible. Moreover, as an underlying basis, existing test theory has to be extended in the area of real-time testing.

Due to the problems introduced by the snapshot semantics of TTCN, standard timers should not be used to test real-time constraints where a high resolution of timing information is required. Therefore in the testing domain, the next step which will be done in the Interval project is to present a real-time extensions for TTCN-3 which allows deterministic real-time testing.

Acknowledgements

Part of this work has been sponsored by the European Commission under contract IST-1999-11557.

References

1. C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. Automatic executable test case generation for extended finite state machine protocols protocols. In IWTCS'97 [17], pages 75–90.
2. C. Bourhfir, R. Dssouli, E. Aboulhamid, and N. Rico. A test case generation tool for conformance testing of SDL systems. In SDL'99 [23], pages 405–419.
3. M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for real-time: what is missing? In *SAM 2000 – 2^{nd} Workshop on SDL and MSC*, pages 108–122, Grenoble, France, June 2000.
4. M. Bozga, S. Graf, L. Mounier, I. Ober, J.-L. Roux, and D. Vincent. Timed extensions for SDL. In *SDL Forum 2001*, Copenhagen, Denmark, June 2001.
5. M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In *Proceedings of IWPTS '95 (8th Int. Workshop on Protocol Test Systems*, pages 151–166, Evry, France, September 1995.
6. A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, and M. Schmitt. Towards the industrial use of validation techniques and automatic test generation methods for SDL specifications. In SDL'97 [22], pages 245–259.
7. ETSI, Sophia Antipolis, France. *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; TTCN-3: Core Language*, v1.0.10 edition, November 2000. DES/MTS-00063-1.
8. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.
9. J. Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts*. PhD thesis, University of Bern, Bern, Switzerland, February 1994.
10. J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs. In *SDL'93: Using Objects*, pages 253–265, Darmstadt, Germany, October 1993. Elsevier Science Publishers B.V.
11. S. Graf. Timed extensions for SDL, November 2000. Delayed Contribution No. 13 to ITU-T Study Group 10, Questions 6&7.
12. ISO/IEC. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework*, 1994. International ISO/IEC multipart standard No. 9646.
13. ITU-T, Geneva, Switzerland. *Message Sequence Charts*, 1996. ITU-T Recommendation Z.120.
14. ITU-T, Geneva, Switzerland. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation*, 1997. ITU-T Recommendation X.293-ISO/IEC 9646-3.
15. ITU-T, Geneva, Switzerland. *Message Sequence Charts*, November 1999. ITU-T Recommendation Z.120.
16. ITU-T, Geneva, Switzerland. *Specification and Description Language (SDL)*, 1999. ITU-T Recommendation Z.100.
17. *Testing of Communicating Systems*, Cheju Island, Korea, September 1997. Chapman & Hall.
18. A. Kerbrat, T. Jéron, and R. Groz. Automated test generation from SDL specifications. In SDL'99 [23], pages 135–151.

19. A. Mitschele-Thiel. *Systems Engineering with SDL – Developing Performance-Critical Communication Systems*. Wiley, Chichester, England, 2001.
20. H. Neukirchen. Corrections and extensions to Z.120, November 2000. Delayed Contribution No. 9 to ITU-T Study Group 10, Question 9.
21. Interval Consortium Web Page. `http://www-interval.imag.fr/`, 2000.
22. *SDL'97 – Time for Testing*, Evry, France, September 1997. Elsevier.
23. *SDL'99 – The Next Millennium*, Montréal, Québec, Canada, June 1999. Elsevier.
24. G. v. Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In SDL'97 [22], pages 261–276.
25. Th. Walter and J. Grabowski. Real-time TTCN for testing real-time and multimedia systems. In IWTCS'97 [17].

Real-time test specification with TTCN-3*

Zhen Ru Dai, Jens Grabowski, and Helmut Neukirchen

Institute for Telematics, University of Lübeck
Ratzeburger Allee 160, D-23538 Lübeck, Germany
Tel.: +49 451 500 3721 Fax: +49 451 500 3722
{dai,grabowsk,neukirchen}@itm.mu-luebeck.de

Abstract. In this paper we propose an extension to the third edition of the Tree and Tabular Combined Notation (TTCN-3) to support real-time testing. Our approach is based on the separation of functional and real-time requirements in the test specification. Functional requirements are defined in form of stimuli and expected responses, i.e., in form of functional TTCN-3 test cases, and real-time requirements are expressed by the relationship of points in time. During test execution the points in time are collected in form of time-stamps that can be evaluated during or after the test run. We explain all extensions to TTCN-3 necessary to support the collection and evaluation of time-stamps and show the applicability of our approach by means of an example.

Motivation

Testing (e.g. [1, 6, 7]) is an important issue during the development processes of communication systems and can be divided into two categories: testing of functional and non-functional requirements. An important sub-category of non-functional requirements are hard real-time requirements [2, 8].

Functional testing is well studied and there exists a standardized testing language: TTCN-3 [3, 4]. Real-time testing is still under study. There is no standardized language to specify real-time tests.

In this paper, we present an extension for TTCN-3 in order to use it for real-time testing. Our intention is to present a powerful but easy to use language extension without changing the existing TTCN-3 semantics. We concentrate on the formal test specification of hard real-time requirements, but not on the theoretical foundations of real-time testing. Our approach aims at the specification of hard real-time requirements, such as delay, response time, latency, throughput, jitter etc.

A solution for Real-Time Testing

The main idea is to separate the description of functional and hard real-time requirements. This allows a clear distinction between these two classes of properties.

* This work is funded by the European Commission contract IST-1999-11557 INTERVAL.

The essence of the various real-time requirements can be broken down to the relationship of pairs or tuples of time-stamps. Hence, our description of real-time requirements is based on relating particular points in time to each other. By using mathematical equations, constraints on those time points can be imposed.

In order to obtain such time-stamps, a functional test case has to be instrumented with special log statements. Thus, during test case execution, time-stamps of the relevant events are recorded.

The analysis whether the recorded time-stamps met the real-time requirement can be done either off-line when test execution has finished or on-line during the test run. Off-line evaluation has the advantage of delaying the test execution by just a small amount of time which is needed for time-stamping. However, in certain situations, functional and non-functional behavior may influence each other. Therefore, an on-line evaluation during the test run might be necessary, too. In this case, time needed for evaluation is critical, because it slows down the tester and influences test execution and verdict.

Implementation using TTCN-3

In order to get time-stamps of the interesting events, the TTCN-3 test cases have to be instrumented by log statements which writes time-stamps in a log file. In TTCN-3, a log file may contain information about sending and receiving events including time information. By evaluating the log file with respect to the time-stamps, compliance to real-time requirements can be checked. Related log entries can be identified and be used as input for the evaluation programs that check the real-time requirements.

The evaluation programs have to be specified in a machine processable format. Since we are already in the TTCN-3 domain, it is obvious to express them by using TTCN-3 functions. Depending on whether a real-time property described by a TTCN-3 evaluation function holds for a test case or not, a verdict can be assigned.

Off-line evaluation can be done independently from the execution of a test case. In this case, no modifications to the TTCN-3 language are needed. A shell script could for instance be used to analyse the log file and feed the extracted time-stamps into the evaluation functions. For on-line evaluation, an extensions of the TTCN-3 language is necessary. Means to interleave test execution and test evaluation have to be added in order to allow such a reactive testing.

Case study

As a case study, we chose the INRES protocol [5] to assess our approach. We enhanced INRES with real-time requirements. The functional TTCN-3 test suite was instrumented by log statements in order to get a log file which can be analysed.

In simple cases, it is just necessary to search for the time-stamp keyword and the additional label within the log file in order to get the time of a certain

event. Slightly more complex cases evolve when loops are executed. In this case, the same label will occur several times in the log file. Nevertheless, using the knowledge of the order of the log statements in the test specification allows to find the right tuples of time-stamps.

This simple approach is not possible anymore if e.g. signals can get lost. A simple pattern matching scheme to find adjacent tuples of time-stamps cannot be applied. However, additional information, like package numbers or consecutive numbers which are coded as arbitrary data in available signal data fields, could be utilised to identify matching tuples of time-stamps.

References

1. B. Beizer. *Black-Box Testing*. Wiley, 1995.
2. The ATM Forum Technical Committee. ATM Forum Performance Testing Specification, October 1999.
3. European Telecommunications Standards Institute (ETSI). *Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3 (TTCN-3); Part 1: TTCN-3 Core Language*. ETSI ES 201 873-1, Sophia-Antipolis, France, 2001.
4. J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the design of the new testing language TTCN-3. In *13th IFIP International Workshop on Testing Communicating Systems (Testcom 2000)*, Ottawa, August 2000. Kluwer Academic Publishers.
5. D. Hogrefe. Report on the Validation of the Inres System. Technical Report IAM-95-007, Universität Bern, November 1995.
6. ISO/IEC. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework*, 1994. International ISO/IEC multipart standard No. 9646.
7. G. J. Myers. *The Art of Software Testing*. Wiley, 1979.
8. V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Request for comments 2330: Framework for IP performance metrics, May 1998.