Sinnott, R.O. (2001) *Real-time systems development with SDL and next generation validation tools.* In: IEEE Real-Time Embedded System Workshop, 3 Dec 2001, London, England.

http://eprints.gla.ac.uk/7291/

Deposited on: 4 November 2009

Real-Time Systems Development with SDL and Next Generation Validation Tools[1]

Dr Richard O. Sinnott
Ericsson Software Technology AB
Ölandsgatan 1-3
37123 Karlskrona
Sweden
Richard.Sinnott@epk.ericsson.se

Abstract

The language SDL has long been applied in the development of various kinds of systems. Real-time systems are one application area where SDL has been applied extensively. Whilst SDL allows for certain modelling aspects of real-time systems to be represented, the language and its associated tool support have certain drawbacks for modelling and reasoning about such systems. In this paper we highlight the limitations of SDL and its associated tool support in this domain and present language extensions and next generation real-time system tool support to help overcome them. The applicability of the extensions and tools is demonstrated through a case study based upon a multimedia binding object used to support a configuration of time dependent information producers and consumers realising the so called lip-synchronisation algorithm.

Keywords: SDL, Validation, Real-Time Systems, Timed Automaton.

1 Introduction

The Specification and Description Language (SDL) [1] is arguably the most successful formal technique used today with widespread usage throughout the software, telecommunications and automotive industries. Part of the reasons for its general adoption are its intuitive graphical notation and excellent tool support. The tool support typically offers capabilities to analyse, design, implement and subsequently test systems, often using combinations of interrelated notations together with SDL such as Message Sequence Charts (MSC) [2] and Tree and Tabular Combined Notation [3].

One of the main perceived benefits of SDL over other notations such as the Uniform Modelling Language [4] (UML) is the ability to model and reason about, e.g. via model checking tools, detailed behavioural specifications, including real-time behaviours. We note here that this is an area that the UML community is currently addressing both within the development of the UML 2.0 specification [5] as well as in proposals such as the Scheduling, Performance and Time in UML [6]. Whilst it is true that SDL through its semantic basis of

---

extended finite state machines does allow for detailed modelling of behaviour and has some language aspects for expressing features of timed systems, these are unfortunately inadequate for real-time systems development. Further, as a natural consequence of the language limitations the associated tools suffer from a lack of precision for dealing with the temporal aspects of specifications and are often unable to enforce or establish the existence of temporal properties. Typical examples of the properties that a real-time specification language and associated real-time tool support should be able to check for include:

- *deadlock properties* where the real-time specification reaches a state where no more transitions are possible and time progresses indefinitely;

- *livelock properties* where the specification is unable to ever receive messages (signals) from the environment due to continuous internal interactions;

- *invariant properties* that must hold for all executions of the model including real-time invariant properties;

- *non-zenoness of runs* where time in the system does not progress beyond a certain value due to continued (non-time dependent) interactions;

As well as these more classical real-time properties, more general properties should also be supported, e.g. *non-linear properties* such as signal X should be followed by signal Y within a maximum of Z time units.

To achieve this, a precise notion of time in SDL and language features that allow for various timing aspects to be both modelled and subsequently validated by associated tools is required. The European project Interval [7] is currently investigating this area. This paper provides an outline of several of the key SDL language extensions as well as an overview of the associated tools being developed. We present both the language extensions and tools through their application to a real-time case study based on a multimedia binding object supporting a configuration of real-time information producers that collectively realise the so called lip-synchronisation algorithm.

The rest of the paper is structured as follows. Section 2 provides an outline of the existing timing features of SDL and how they are handled by current toolsets, together with their associated limitations. Section 3 provides an outline of the case study used to introduce the language extensions and associated tools. Section 4, then provides an outline of the key features of the SDL specification realising the case study. Finally, in section 5 we draw some conclusions on the work and indicate future directions for both the language extensions and tool development.

2   Problems of SDL for Real-Time Specification

The language SDL contains various features which can be used to model aspects of timed systems. Specifically, the specifier is able to describe temporally dependent behaviour through using: **timers**, enabling conditions and continuous signals, where the latter two features can be used for timing purposes through referencing the time variable **now**.

With regard to timers, the SDL 2000 standard [SDL] states that:

> *A timer is an object that is owned by an agent that causes a timer signal stimulus to occur at a specified time. When an inactive timer is set, a time value is associated with the timer. Provided there is no reset or other setting of this timer before the system time (**now**) reaches this time value, a signal with the same name as the timer is put in the input port of the agent.*

It also states that:

> *The **now** expression is an expression which accesses the system clock variable to determine the absolute system time... Whether two occurences of **now** in the same transition give the same value is system dependent. However, it always holds that: **now** <= **now**;*

The problem with this definition as far as the modelling and subsequent validations of real-time systems with SDL is concerned, stems from the notion of system time in SDL. Specifically, time as given by the system clock (**now**) is something external to the specification and to all intent and purposes, independent from the specification itself. For example the system clock cannot be reset within the specification, nor does it progress in an orderly fashion as one would expect a clock to. Rather, the only means for any form of control over the system clock is through the usage of timers. A typical assumption on the progress of time in SDL, as has been adopted by most tool vendors [13,14] is to assume that time only progresses when the system is in a stable inactive state, i.e. where no signals can be sent or consumed. With regard to real-time systems development, this is of limited use since with this approach when no timers are currently set then time, in effect, does not progress. Further, since timer expiry results in an input signal being placed in the (possibly non-empty) input queue of the associated agent, these signals can be in the queue any arbitrary time before they are consumed. If the expired timer was to stop delivering plutonium, then such a delayed treatment is unlikely to be satisfactory.

Timed behaviour can also be modelled via enabling conditions and continous signals as shown in Figure 1. An enabling condition referring to the system time (**now**) can be attached to an input signal as shown on the left of Figure 1.

ready

inpX

now <= 10

idle

ready

now >= 10

outX

idle

Figure 1: Usage of Time Enabling Conditions and Timed Continuous Signals

The intention of such a construct is to limit the consumption of a signal so that it can only be consumed when some specific time constraint is satisfied, e.g. here signal *inpX* can only be consumed before the global time reaches 10. A continuous signal as shown on the right of Figure 1, can also refer to **now**, where the intention is that when some time constraint is satisfied in a state, the behaviour of the agent can evolve without environmental interaction, i.e. the behaviour following the continuous signal can occur (outputting signal *outX*).

These constructs also suffer from the external notion of time in SDL. Further, usage of **now** in such systems often causes unexpected and unwanted results in SDL tools. For our purposes here we broadly classify two SDL tool types: *simulators* and *model checkers*. With simulators, the behaviour of the model is most often explored manually. Typically, the user sends a signal into the model from the environment and observes (traces) its path through the system. Such an approach is often used for requirements capture, e.g. through ensuring that a given system satisfies a given use case. Model checkers however, perform more rigorous checks on the system behaviour. These tools allow for the state space of the system to be explored to check for the presence or absence of particular behaviours or system properties more generally, e.g. the properties described in section 1 may typically only be discovered via model checking tools. Indeed, such rigorous model checking capabilities are one of the key advantages of formal methods versus non-formal approaches.

Whilst it is possible to model and simulate systems where the value of **now** is changing (progressing) as timers are set and subsequently fire, the usage of tools for performing more rigorous model checking are adversely influenced by such a weak model of time. Thus, since time can in principle progress in every system state, dealing with **now** results in the well known problem of state space explosion. Thus every time **now** increases results in a new system state and since **now** can have a potentially infinite number of values, the state space explodes immediately. To overcome this, model checking tools effectively ignore the concrete value of **now**.

As an example of this discrepancy between simulation and model checking usage of SDL specifications, consider the SDL fragments in Figure 2. The first fragment shows a timer being set for time **now**+2. Once this timer fires, i.e. signal *t* is included in the input queue of the associated agent and subsequently consumed, one would expect that the system time (**now**) to be greater than or equal to 2 depending on the initial value of **now**. When this system is simulated this results in the continuous signal expression being satisfied and signal *outX* being sent. However, model checking tools do not result in signal *outX* being sent since the explicit value of **now** is ignored when model checking. A similar **now** problem occurs in the second SDL fragment. Simulation tools show that the signal *outX* is sent with a timestamp equal to 3 (or more depending on the value of **now** when the timer is set), whereas model checking tools show the signal sent with a time stamp of 0.
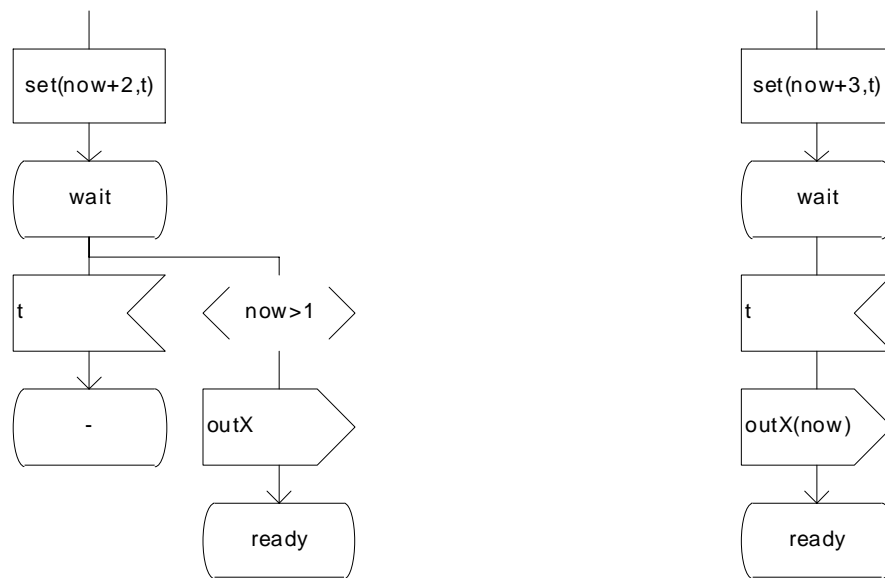

Figure 2: Problems of Timing Based on NOW

As well as these direct problems of dealing with time in SDL, a fundamental aspect of modern real-time distributed systems that makes them especially complex to model and reason about, is their very lack of a global system clock. Thus it is typically the case that temporal synchronisation between distributed components is necessary where the simplifying assumption of reading and synchronising on a global clock is infeasible or impossible. It is also often the case that this temporal coordination of the components is the key area where SDL and its associated model checking tools should be applied, i.e. this is the most complex design area where unforeseen errors such as deadlocks, livelocks etc caused by the temporal coordination of the components, are likely to be introduced.

To overcome these limitations, it is necessary to both extend the SDL language with appropriate timing features as well as developing model checking tools that incorporate features for exploiting these timing informations. We introduce these SDL language features

and how they are supported by associated tools via a case study based on a multimedia binding object supporting a configuration of time dependent information producers and consumers.

3   Multimedia Binding Object Case Study

One of the classic scenarios for describing real-time issues is the lip synchronisation algorithm [8,9,10]. In this system, we assume that we have two (or potentially more) producers of multimedia flows of information, e.g. an audio flow and a video flow. The goal of this algorithm is to ensure that the two flows of information obey strict timing considerations, i.e. they are synchronised so that the video image of somebody speaking and the audio flow of the associated voice are kept within certain time bounds. In addition, the flows of information themselves have strict timing requirements that apply to them.

One way in which this algorithm can be applied is through a multimedia binding object [11,12] which is responsible for the management of the production and consumption of the information flows. Such a multimedia binding object configuration is presented in Figure 3.
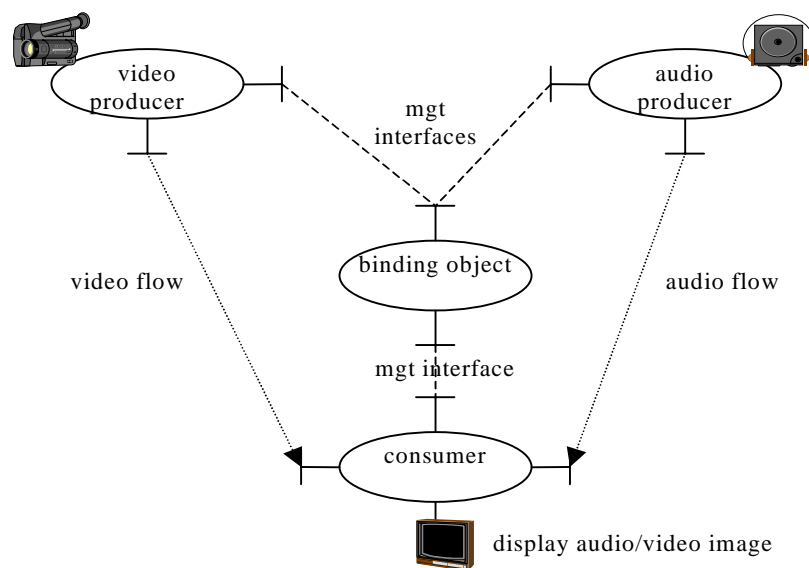

Figure 3: Multimedia Binding Object Configuration

Here the binding object is used to configure and manage the production and consumption of the audio and video information flows. Typical examples of the functionality required by the management interfaces is the ability to start, stop, suspend, resume the production or consumption of the information flows. In addition, it is useful to be able to tell producers or consumers of flows to send or consume faster or slower.

Before the producers and consumers send and receive the flows of information, it is necessary that they agree upon the timing aspects of flows. This is achieved through negotiations with the binding object and the creation of a binding contract. The binding

contract itself can include numerous different types of real-time aspects which can be negotiated, examples of which are:

- maximum and minimum rates of production and consumption and the related features of delays and throughputs;
- maximum acceptable loss, i.e. how many consecutive lost information items a consumer can stand before the audio/video quality deteriorates too much;
- maximum acceptable bound and unbound jitter rates.

With regard to this last bullet point, jitter may be considered as the upper and lower limit on the time window at which a consumer can accept an information item, e.g. a frame. For example, if a frame is expected every τ seconds with an allowed variation of $\Delta\tau$ then a frame should arrive within the range $\tau - \Delta\tau$ to $\tau + \Delta\tau$. There are two main cases of jitter: bounded jitter and unbounded jitter. The distinction between the two cases is dependent upon whether the arrival time of the last frame influences the arrival time of the next frame. In unbounded jitter, if frames are expected every τ seconds with a variation of $\Delta\tau$ then should frames consistently arrive early, but within the allowed time range, then the flows will eventually drift out of synchronisation. For example if τ was 30 time units say and $\Delta\tau$ was 5 time units and frames arrived every 29 time units, then after five frames had arrived, all subsequent frames would be outside the allowed range, i.e. the next frame would be expected at 180 but would arrive at 174 time units which would exceed the maximum unbounded jitter rate of 5.

In bounded jitter if a frame arrives early but within the allowed time variation, then the arrival time of the next frame is time τ after that arrival time. Hence using the above numbers, if a frame arrives at time 29 then the next one would be expected at time 59 and not 60. From this, it can be seen that bounded jitter does not allow flows to drift out of synchronisation. Of course, both forms of jitter are affected by network latency and loss.

Once the parties (producers, consumers and binding object) have agreed upon the contents of the binding contract, this is then used to influence and enforce the interactions of the involved objects. Thus for example, it should not be possible for a consumer to request (via the binding object) that a producer produces at a rate which exceeds that agreed upon within the binding contract.

4   SDL Language Extension and Tool Support

To ensure that the timing aspects of a given binding are maintained by the involved parties, it is necessary to enforce the occurence of certain time dependent actions. As discussed previously in section 2, SDL is severely limited with regard to both modelling and subsequent enforcement of timing aspects since a simple global and external timing model is

adopted. One way that timing enforcement can be achieved however is through introducing *action urgencies* into the SDL language and the usage of clocks to monitor time progress.

Action urgencies, which stem from the theory of timed automaton [17], offer an abstraction that can be used to influence the behaviour of a system as time progresses. A transition can be regarded as urgent if it will be taken or disabled before time progresses. There exist three main types of transition urgencies which can be used to control progress of time with respect to progress of the system behaviour: *eager transitions, lazy transitions* and *delayable transitions*. Eager transitions are urgent as soon as they are enabled, i.e. they are to be executed as soon as possible and time should not progress as long as an eager transition is enabled. Lazy transitions on the other hand, do not inhibit time progress in any system state. Delayable transitions are a combination of both eager and lazy transitions in that they become urgent only when time progress would otherwise disable them.

We note that the current SDL semantics treats all transitions as lazy since it places no constraints on time progress. Most tools however implement an eager semantics where transitions are fired as soon as they are enabled without letting time progress.

Whilst a single external clocking model could be applied together with action urgency, more flexibility, realism and expressive power is gained from assuming a local clocking model. What is required is that clocks can be added to a given model and constraints expressed on the times recorded by those clocks can be checked in conjunction with considering urgencies.

We demonstrate the application of these concepts as they have currently been realised in the next generation validation tools under development in the Interval project [7]. For clarity, we consider the representation of an audio producer object and a video producer object that can produce data at three rates: fast, medium and slow. For simplicity and brevity we assume that these values have been agreed upon via interaction between the producer/consumer objects and the binding object. In addition we assume the data content being sent is required and in the correct format etc.

Since we wish to model and reason about both intra-flow and inter-flow (for lip-synchronisation) jitter, for modelling purposes we also require some fluctuation in the rates of production and subsequent consumption. Specifically we assume the following rates of production in milliseconds:

production rate	audio producer	video producer
fast	90-100	100-110
medium	95-105	105-115
slow	100-110	110-120

Table 1: Rates of Production

Thus based on these values and assuming that they have been agreed upon by the consumer, the agreed consumption rates for the audio flow lie between 90-110ms and between 100-120ms for the video flow respectively.

Once these values have been agreed upon in the binding contract, the binding object sends a message (signal) to the producers and consumers telling them to start with the production and consumption of the information flows. Once this message arrives, the various objects introduce new clocks into the system which are then used for controlling the rates of production and consumption. The producers start production at the medium rate. In the case of the audio producer, this is depicted in Figure 4.
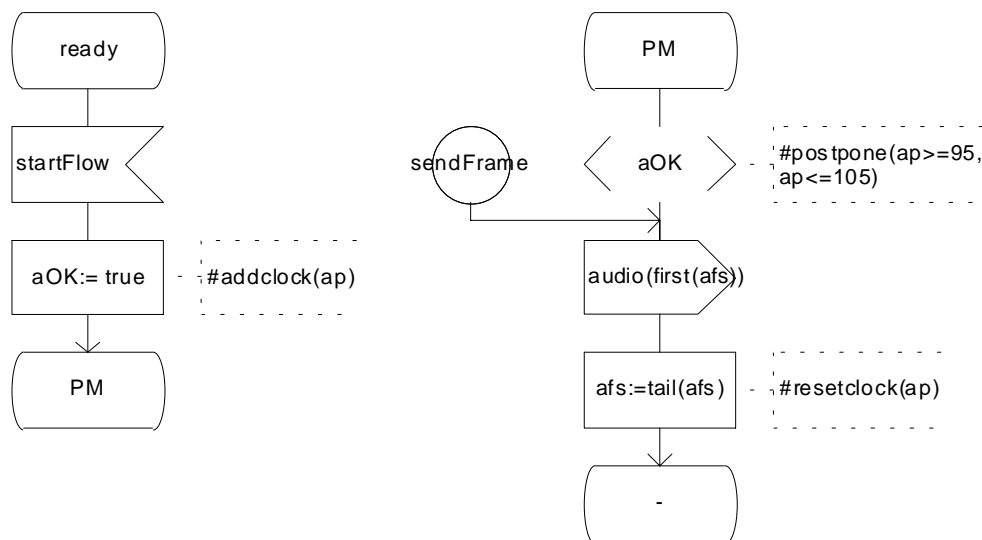

Figure 4: Modelling a Producer Object

Here upon reception of the signal (*startFlow*) an audio flow production variable (*aOK*) is assigned the value true and a new audio production clock (*ap*) is added to the system via the new command *addclock*. Further, to ensure that the audio production occurs at the agreed time, a continuous signal is used with an attached action urgency using the *postpone* command. We note that the syntax used here (*postpone*) corresponds to the delayable action urgency discussed previously. This could not be used for syntactic reasons with the existing tool upon which the prototype is based. The postpone command given here is only satisfied once the clock *ap* has progressed to some value between 95-105. Note however, that once the clock reaches 105 time units then the action then becomes urgent and hence must happen before time is allowed to progress or will be disabled indefinitely. Once the timing condition is satisfied, an audio signal is sent to the consumer with the associated data (the contents of which are not specified here) which is then removed from those to be sent. The clock is then reset to zero and continues until it once again reaches a value such that it satisfies the action urgency condition, namely that it is once again between 95-105.

Modelling different rates of production can be achieved by using different states with continuous signals having the appropriate action urgencies attached as shown in Figure 5. These states can then be reached via signal reception from the binding object, e.g. upon demand from the consumer, the binding object requests that the producer send faster or slower (moving to states PF or PS respectively).
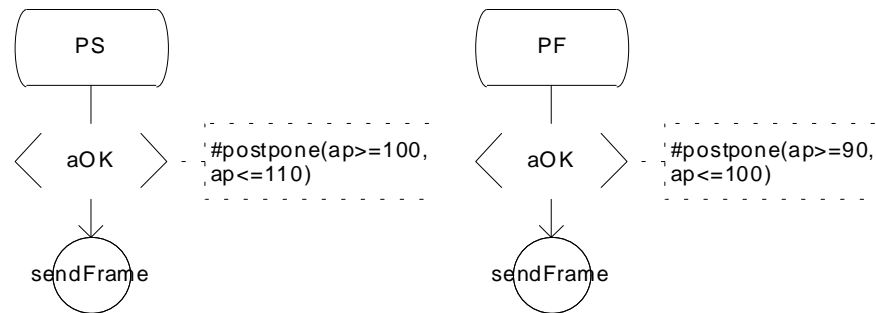

Figure 5: Modelling Different Rates of Production

The video producer can be specified similarly. For the consumer of the flows it is required that the audio and video flows are consumed at rates 90-110ms and 100-120ms respectively. This can be represented as shown in Figure 6.
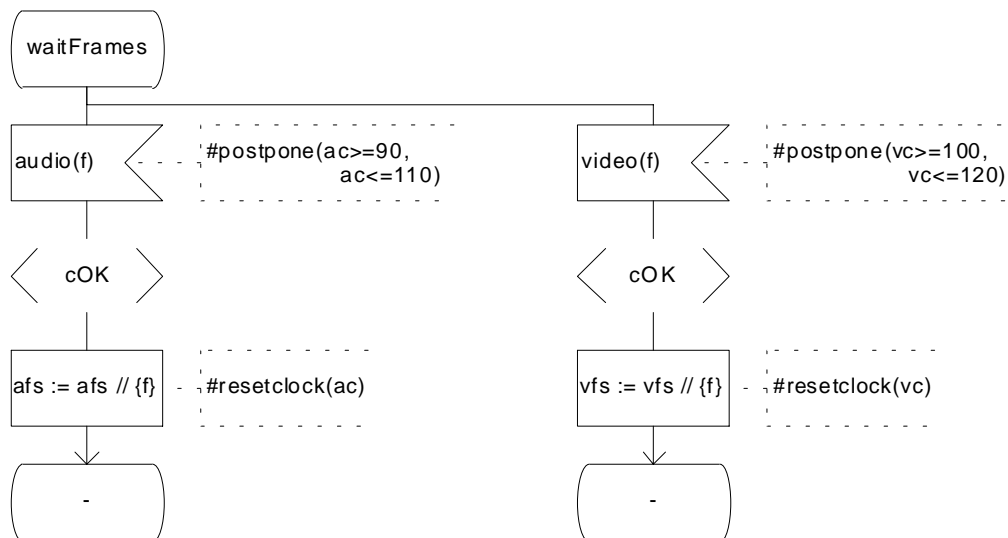

Figure 6: Modelling the Consumer of the Information Flow

Here the audio and video input signals are constrained to only be consumed at times that satisfy the attached action urgencies and provided the consumer is able to consume, i.e. the consumption variable (*cOK*) has been set to true. This variable and the associated clocks for audio and video consumption (*ac* and *vc* respectively) are introduced after the reception of the signal to start consuming from the binding object. For brevity this is omitted. Once consumed the audio and video frames are stored and can be subsequently displayed at a given display rate (not shown here).

To understand the way in which time constraints imposed on the system can be used to influence the behaviour of the system, we consider one particular trace of the system consisting of the sending and consumption of four frames: audio, video, audio, video respectively. The clock constraints for these interactions are exhibited are shown in Table 1, where * implies that this event occurs.

ap (95-105)	vp (105-115)	ac (90-110)	vc (100-120)	global time
0-105*	0-105	0-105	0-105	0-105
0-15	95-110	95-110*	95-110	95-110
0-20	95-115*	0-20	95-115	95-115
0-25	0-15	0-25	105-120*	105-120
0-105*	0-105	0-105	0-105	105-210
0-30	75-110	80-110*	70-110	190-220
0-40	75-115*	0-40	70-115	190-230
0-50	0-30	0-50	90-120*	210-240
0-105	0-105	0-105	0-105	210-315

Table 2: Audio, Video and Global Clock Values

Initially before any frames have been sent, all that is known is that the value of the various clocks are that they somewhere between 0-105 time units. The upper bound here is based on the knowledge that the sending of an audio frame should occur before 105 time units (and after 95 time units), hence time is constrained by this upper bound. Once an audio frame is sent however, it is known that at least 95 time units has passed (since the frame could not be sent before then), hence the lower bounds of the other clocks in the system must have progressed by at least 95 time units. The next upper bound (110 time units) on these clocks is given by the consumption of an audio frame clock which requires that the audio frame must be consumed before 110 time units. Note that the audio producer clock is reset once the frame is sent, hence its value now lies somewhere between 0-15. These clock values can be understood by considering the possibility that the audio frame was sent at 95 time units, if so then the clock would have a maximum value of 15 time units as the upper bound has progressed to 110 time units. It can be seen that once an audio and a video frame have been sent and consumed (line 5 of table 2), the producer and consumer clocks have the same time constraints as they initially had (0-105ms) since they have all been reset, the effective global time has now progressed to somewhere between 105-210ms.

Through this approach of comparing clock upper and lower bounds constraints in conjunction with the overall system states and ensuring the occurence of certain events through an action urgency semantics, real-time properties of SDL systems can be validated via model checking tools. As an example of a property of the system that we would like to validate, consider the (lip-synchronisation) case where we want to ensure that the audio and video consumption clocks never differ by the maximum time for one video frame, i.e. they

should never differ by more than 120ms. One way in which this property is expressible is using an automaton related language called GOAL [18].

The GOAL language allows for aspects of an SDL model to be observed to check for the presence or absence of certain behaviours. GOAL observers themselves have states which can be declared as normal, success or error states. GOAL transitions themselves are triggered by events occurring in the SDL model. The required property as expressed through the GOAL language, for clarity showing only parts of the audio frame consumption is shown in Figure 7. For brevity we omit the parts of the GOAL specification showing where the different upper (*oa1,ov1*) and lower (*oa2,ov2*) bounds on the audio and video clocks are introduced into the system.
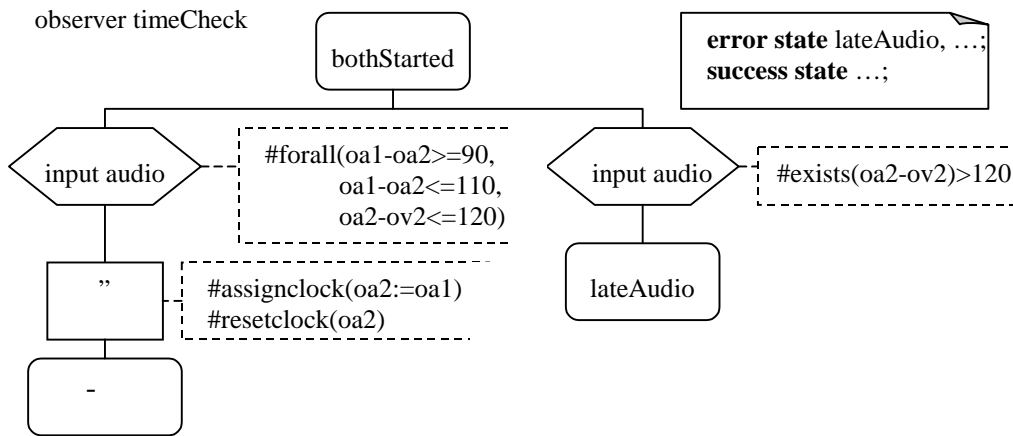

Figure 7: Example Properties to be Satisfied by Model

Here the consumption of an audio frame has the requirement that it should occur 90-110ms after the last audio frame. In addition, the audio consumption clock should never differ by more than 120ms from the video consumption clock. When this is the case, an error state is reached. Other properties can also be specified directly in GOAL, e.g. to ensure that it is never the case that an audio (or video) producer in state *PF* (fast) is asked to produce more quickly. Thus it should never be the case that a request is made which violates the binding contract. In addition to these types of safety properties, the GOAL language can be used to check for liveness properties, e.g. checking for non-zeno runs.

5   Conclusions

This paper has shown the limitations of the existing treatment of time within SDL and its handling by associated tools. We have proposed more useful and powerful time features for SDL based on timed urgencies. These result in system models which next generation real-time tool support can be applied for model checking and to validate numerous temporal properties. Prototypes of these tools have been completed and are being applied to various case studies within the Interval project – this case study being just one.

We note that these language extensions presented in this paper have been deliberately developed to be manageable by existing toolsets, i.e. they are treated as SDL comments and can be ignored by non-real-time SDL toolsets. As such, it is very likely that as work in Interval progresses, they will be modified to be both more intuitive and in a form suitable for the next version of the SDL standard. Further, the extensions shown in this paper represent only a subset of the overall language extensions. More information on these extensions are given in [15,16].

Of course, formally validated models represent only one stage in the overall development of real-time systems. Nevertheless we argue that languages and tools that allow for powerful reasoning of real-time aspects represent a significant step in real-time systems development, i.e. it is at the design level that key decisions on the temporal requirements of the systems are made. The earlier, potential real-time design errors are discovered, the better, i.e. design changes at a later stage of software development are likely to be more costly.

Finally we note that these real-time SDL extensions and tools are just one aspect under investigation in the Interval project. The project is also looking into development of real-time tools and language extensions for MSC 2000 and TTCN-3 to provide a complete real-time development lifecycle from real-time requirements to real-time models and subsequent real-time tests, with visibility and tractability via tool support throughout. More information on Interval can be found at http://www-interval.imag.fr. We also note that an Interest Group has also been set up which can be subscribed via an email to: interval-ig@imag.fr.

5.1   Acknowledgements

6   References

[1] International Telecommunications Union (ITU-T), Recommendation Z.100, Specification and Description Language (SDL), SDL 2000, Geneva Switzerland.

[2] International Telecommunications Union (ITU-T), Recommendation Z.120, Message Sequence Charts (MSC), MSC 2000, Geneva Switzerland.

[3] European Telecommunication Standards Institute (ETSI) - Methods for Testing and Specification (MTS): The Tree and Tabular Combined Notation version 3, TTCN-3: Core Language, DES/MTS-00063-1 v1.0.9, Sophia Antipolis, France.

[4] Object Management Group, Unified Modelling Language Specification, version 1.4 Beta R1, November 2000 .

[5] Object Management Group, UML 2.0 Superstructure Request For Proposal (RFP), Needham MA, USA, ad/2000-08-09.

[6] UML Profile on Scheduling, Performance and Time, http://www.omg.org/ad99-03-13.

[7] INTERVAL project web site, http://www-interval.imag.fr

[8] R.O. Sinnott, Specifying Aspects of Multimedia in LOTOS, Conference on Computational Intelligence & Multimedia Applications, New Delhi, India, 1999.

[9] R.O. Sinnott, Specifying Multimedia Configurations in Z, Conference on Computational Intelligence & Multimedia Applications, New Delhi, India, 1999.

[10] T. Regan, Multimedia in Temporal LOTOS: a Lip-Synchronisation Algorithm, Proceedings of PSTV XIII, pages 127-142, eds A. Danthine, G. Leduc, P. Wolper, 1993.

[11] R.O. Sinnott, K.J. Turner, Specifying Multimedia Binding Objects in Z, Trends in Distributed Systems: CORBA and Beyond, LNCS vol. 1161, eds O. Spaniol, C. Linnhoff-Popien, B. Meyer, Springer-Verlag 1996.

[12] Reference Model for Open Distributed Processing: Part 3 – Architecture, ISO/IEC 10746-3, ITU-T X.903, Computational Viewpoint.

[13] For more information on TAU see http://www.telelogic.com.

[14] For more information on ObjectGeode see http://www.cs-verilog.com.

[15] M. Bozga et al, Timed Extensions for SDL, Proceedings of SDL Forum, pages 223-235, editors Rick and Jeanne Reed, Springer 2001.

[16] I. Ober, A. Kerbrat, Verification of Quantative Temporal Properties of SDL Specifications, Proceedings of SDL Forum, pages 182-202, see [15].

[17] S. Bornot, J. Sifakis, Relating Time Progress and Deadlines in Hybrid Systems, Proceedings of HART'97, LNCS volume 1201, Springer-Verlag, 1997.

[18] B. Algayres et al, GOAL: Observing SDL Behaviours with GEODE, Proceedings of SDL'95 with MSC in Case, eds R. Braek, A. Sarma, Elsevier Science, 1995.