Specifying Real-Time Requirements for SDL Specifications – A Temporal Logic-Based Approach*

Stefan Leue

Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada,
Email: `sleue@swen.uwaterloo.ca`, WWW: `http://swen.uwaterloo.ca/~sleue`

Abstract

The expressiveness of many state-transition based formal description techniques, e.g. the ITU-TS standardised *Specification and Description Language* (SDL), does not capture hard real-time requirements. In telecommunications systems engineering, hard real-time requirements, however, are an important class of properties. They occur in the description of progress properties in telecommunications protocols as well as in the specification of real-time related of Quality of Service (QoS) requirements. We suggest integrating functional system properties, given as SDL specifications, with real-time requirements expressed in terms of real-time temporal logic formulas. We call the resulting specifications 'complementary specifications'. First, we show the inexpressiveness of SDL with respect to hard real-time requirements. Next, we define a common model theoretic foundation which allows SDL specifications to be used jointly with temporal logic specifications. Then we give examples of commonly used real-time related QoS requirements, namely delay bound, delay jitter, and isochronicity. We also discuss the specification of various QoS mechanisms, like QoS negotiation, QoS monitoring and jitter compensation. Finally, we point at related formal verification problems.

Keywords

Specification, Specification and Description Language (SDL), Real-Time Requirements, Metric Temporal Logic, Model Theoretic Semantics, Quality of Service.

1. INTRODUCTION

Standard state machine model based formal description techniques like SDL or Estelle [24] enjoy wide acceptance in the field of telecommunications systems engineering. These languages are targeted to the specification of *functional* system properties, in particular, safety and liveness properties of the sets of sequences of observable behaviour. These techniques are relatively good at expressing safety properties but express only trivial liveness and progress properties. However, in telecommunications systems engineering an important class of properties is related to system progress as well as timely behaviour.

In this paper we will address the question of how real-time related system properties can suitably be expressed for SDL specifications. In Section 2 we investigate the SDL timer

*The work was partly supported by the Swiss National Science Foundation and the Swiss Federal Office for Education and Scientific Research (while the author was with the University of Berne, Switzerland), and by the National Science and Engineering Research Council of Canada.

mechanism and observe the limitations in its expressiveness. As a consequence we recommend the use of complementary real-time extended temporal logic (Metric Temporal Logic, MTL) [5] formulas complementing the SDL specifications to remedy this shortcoming. This requires providing a model for SDL specifications based on which temporal logic formulas can also be interpreted. We define this model in Section 3. In Section 4, we show how temporal logics can be used in conjunction with SDL specifications when interpreted on this state transition model. The underlying idea is that both the SDL specification as well as the temporal logic specification constrain the allowable behaviour of the system. We require that both specifications are satisfied by a system. In Section 5, we specify a range of different real-time constraint based progress and QoS requirements complementing SDL specification examples. These include: message transmission delay bounds, delay jitter bounds, isochronicity related requirements, and requirements on transmission rates. In Section 6, we exemplify how some QoS related mechanisms can be specified using our approach, like QoS negotiation and reaction to QoS guarantee violation. Although capturing requirements is our main concern in this paper we will briefly point at formal verification problems in Section 7. We conclude in Section 8.

2. A CRITIQUE OF THE SDL REAL-TIME MECHANISM

SDL has a built-in real-time mechanism, relying on an asynchronous timer mechanism. We will argue here that this mechanism is inexpressive with respect to the most important class of real-time requirements, namely hard real-time or bounded response constraints (see for example [13] [5]). We will briefly explain why this class of constraints is important for requirements specifications of real-time systems, and we will then address the unsuitability of the SDL mechanism.

Real-Time Requirements. Liveness properties are properties of a system which state that "something good will *eventually* happen" [2]. This class of theoretically interesting properties has proved to be of limited practical use. By asserting that one can rely on the fact that when one has requested a service, the request is eventually going to be served does not exclude the possibility that one may need to wait a finite but *apparently* limitless period of time for the servicing of the request [21]. It is theoretically possible to specify situations which are perfectly "legal" from a liveness point of view but which could result in the user having to wait for an impractically long period of time before the request is serviced (e.g. exceeding human life expectancy). To overcome this problem, real-time models enforcing progress by relying on the *urgence* of certain timed events have been introduced. In the context of SDL this means that notion of *time* needs to be introduced into the purely untimed basic state and event sequence model. A suitable timed execution model for our purposes is the model of timed traces [5], where steps in system traces are labeled with monotonically increasing timestamps. For example, the requirement that a request be serviced within t time units of the current moment in time is expressed in the timed trace execution model as: *the request will be serviced in a state* S_i, $i \geq j$, *so that the timestamp* $ts(S_i)$ *differs from the current time stamp* $ts(S_j)$ *by not more than* t *time units.* We call such a requirement a *bounded response* requirement [14]. Bounded response requirements are crucial in many control system specifications, e.g. in communication protocols and safety-critical systems [13].

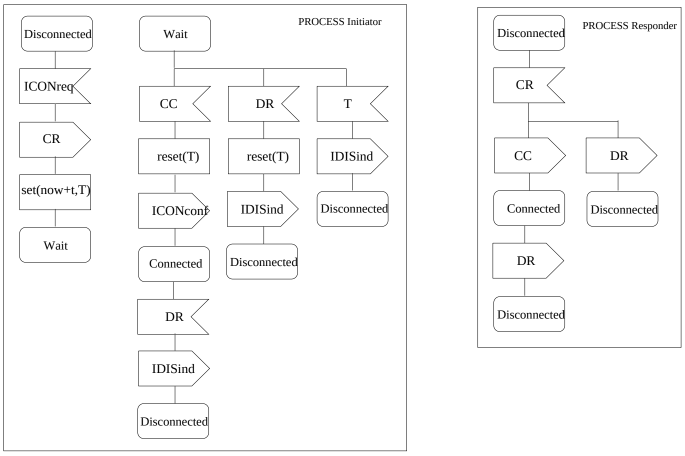

Figure 1. SDL specification of the INRES connection establishment

The SDL Real-Time Mechanism. Real-time is introduced into SDL by an *asynchronous timer* mechanism [8]. An SDL specification can access the value of a global clock by reference to a variable called `NOW` which always refers to the current moment in time. The SDL command `set(now+t, T)` sets the value of a timer called `T` to a point of time which is t time units greater than the current moment of time. We shall call a process which sets a timer, the *timed* process. The set timer is managed by an independent *timer process*. Each time a timed process sets a timer an instance of the timer process is generated. The timer process continuously compares the value to which the timer is set with the current global time. When the value to which the timer is set is reached or exceeded, the timer process communicates the expiry to the timed process by placing a *timer signal* at the end of the input queue of the timed process. Similar to any other signal, the timed process may consume the timer signal from its input queue whenever it has reached the head of the queue, and react accordingly. Timers may also be *reset* by the timed process in which case the timer process *deactivates* the respective timer and removes the timer signal from the timed processes input queue in case the timer expired before the `reset`.

Example and Critique. In Figure 1 we present an SDL specification of the INRES connection establishment protocol, using the SDL timer mechanism [8]. The `Initiator` process sets he timer `T` to time `NOW` plus the time distance value `t` when it sends a `CR` PDU. When the initiator is in state `wait`, it will either receive a response from the `Responder` process, which is considered to be the normal case of operation, or a timer signal `T`. This mechanism is generally assumed to ensure progress of the system by forcing the timed process `Initiator` to react within a bounded time frame after sending the `DR` signal. We will now show that this assumption is false:

- Processes receive timer signals asynchronously through their input queue. The expiry therefore occurs asynchronously from the timed process, we may therefore only infer that the system reacts *some time after* the timer expires. Assume that at time T_{now}, a process sets a timer τ to a time value $T_{now} + T_v$. When the timer process expires, a *timer signal* is generated and placed in the timed processes' input queue some $T_1 \geq 0$ time units after the expiry at $T_{now} + T_v$.

- Furthermore, no estimation can be made of the time it takes to consume all events in the queue which may (potentially) have arrived earlier than the timer signal and which have not yet been consumed by the timed process. The timer signal will be consumed some $T_2 \geq 0$ time units after arriving at the input queue by the timed process[2].

- The interaction between the input queue and the process is asynchronous. Even if a timer signal has arrived when the input queue of the timed process was empty it cannot be guaranteed how long it will take for the timed process to actually consume the timer signal and react accordingly. Formally, the earliest reaction to the timer expiry will happen $T_3 \geq 0$ time units after the consumption of the time expiry signal.

This means that the delay δ, between the point of time when the timer expires, and the moment at which the SDL specification reacts to the expiry, can be estimated as $0 \leq \delta \leq T_1 + T_2 + T_3$. None of the values T_1, T_2 and T_3 is bounded, and hence δ is unbounded. We conclude that as there is no upper bound for the value of δ it is not possible to specify a bounded response requirement using the described timer mechanism.

Remedies. For SDL as a state transition model based language, approaches based on real-time extended automata and temporal logics seem to be most suitable in order to capture complementary real-time requirements. The description of real-time constraints based on state transition systems by so-called *timed automata* has been suggested in [22] and [4]. Transitions of timed automata are attributed by time constraints and real-time is introduced by clock variables. We consider the attribution of automaton transitions (potentially corresponding to *SDL process transitions*) as too inflexible a means for specifying real-time constraints and target at a more flexible solution. A similar criticism applies to [7] which in addition lacks a formal foundation. [6] suggests a method to upgrade a programming language by introduction of clock variables and a so-called *guarded wait* statement. Applying this suggestion to SDL is certainly a very appealing idea, but we refrain from changes to SDL itself for the time being. Finally, the use of real-time extended temporal logics has been put forward by many authors to specify and reason about real-time constraints for reactive, state-transition based systems [14, 5] [1] [16]. This approach enjoys a high degree of flexibility in the specification of real-time constraints, and we will therefore pursue the idea in the following Sections.

3. A STATE-TRANSITION MODEL FOR SDL SPECIFICATIONS

In this Section we define a rudimentary computational model for SDL specifications, a so-called *Global State Transition System* (GSTS), which will serve as a common formal model for the interpretation of SDL specifications and temporal logic formulas.

[2] Note that a finite but unbounded number of messages can be in the input queue of the timed processes ahead of the timer signal. Note also, that depending on the structure of the specification, the timed process may run into some state from which it will never neither reach a reset instruction nor a timer signal input statement, which implies that the system may never react to the timer expiry at all.

The main components of the GSTS model are as follows: I. *Process control and data manipulation* of an SDL process when executing a transition. II. *Communication:* SDL processes communicate via potentially unbounded queues, and each SDL process has exactly one input queue handling all incoming messages from any other process. Communication statements `INPUT` and `OUTPUT` will change the state of these queues[3]. The local state of an SDL process hence consists of the combination of current values for the data variables, the point of local process control, and the state of the input queue. III. *Global System States and State Transitions:* The global system state (GSS) is the product of all local states of all processes of an SDL specification. SDL processes run concurrently and we choose an interleaving approach to represent this concurrency. We assume a nondeterministic choice when more than one process has an enabled transition in a given GSS. Note that the resulting GSTS model for SDL specifications is not finite.

For a given SDL specification, the unwinding of the corresponding GSTS model will describe all admissible sequences of states of an SDL specification, called its computations. In describing sequences of states, the model also describes sequences of state transitions, which are in turn triggered by events (e.g. input and output) in the system. The computations will later serve as models for what we call *complementary* temporal logic specifications, only those specifications which satisfy both the properties expressed by the SDL specification *as well as* the properties expressed by the temporal logic specifications are considered to satisfy the composed specification. It should be emphasized that the goal here is not to define *yet another* formal semantics for SDL in addition to the ones defined in different documents (e.g. [10]), but to provide for an adequate capture of real-time requirements in the context of SDL specifications for which none of the existing formalisations is suitable.

Related work. Our definitions here are close to the *Basic Transition Systems* of [20]. Our pSTS models can be seen as a logic based formulation of *Extended Finite State Machines* (EFSM) [19]. The modeling of SDL processes as EFSMs has been suggested in [8] and [23]. However, as we will see later, the mapping of SDL process transitions as informally described in these approaches is too coarse in order to adequately represent the structure of an SDL transition. Alternative formalizations of EFSMs can be found in [15] (where the state space is finite by limitation of the range of data variables and variables representing the state of communication channels to finite domains), and in [11] and [17] (from where we take part of our formalization). [9] describes and formalizes the use of queues to model the collective behaviour of concurrent FSM which communicate asynchronously via queues (there called *protocols*) and we use part of their formalization for our work.

3.1. Process State Transition Systems

The process state transition systems (pSTS) we define here represent an SDL process by a set of symbolic states, a set of program variables (consisting of control and data variables), and by its interactions with the environment (input and output of signals). The 'logic' of an SDL process is encoded in its state transition relation.

---

[3] For reasons of conciseness we do not address inter-process communication mechanisms like *viewing* or *remote procedure call*, but a treatment of these communication mechanisms within our framework is straightforward. Furthermore, we only consider so-called non-delay channels in the SDL specifications.

Formal Definition Process State Transition System (pSTS). A Process State Transition System P is defined as a tuple (S, D, V, O, I, Q, T, C) where S is a finite set of *symbolic states*, D is an n-dimensional linear space where each D_n is an *interpretation domain*, V is a finite set of *program variables*, $V = \{\pi, v_1, \ldots, v_n\}$ where π is a *control* variable ranging over elements of S and $v_1, \ldots, v_n$ are *data* variables so that $v = (v_1, \ldots, v_n) \in D$, O is a finite set of *output signal types*, I is a finite set of *input signal types*, Q is a linear sequence $q_1, \ldots, q_m$ (in the standard mathematical sense) of elements from $I \times D$ which we call *input queue*, T is a *transition relation*, with $T : S \times 2^D \times Q \rightarrow S \times 2^D \times Q$, and C is an *initial condition* on $S \times 2^D \times Q$. A *state* s, is a function $s : V \times Q \rightarrow 2^S \times 2^D$ assigning a value to every variable in V and to Q. By $s[x]$ we denote the value of variable x in state s. We denote the set of all variables by V. Apparently, V can be infinite.

Transition Relation, Admissible Sequences, and Reachable States. We associate a set $\mathcal{T}_T = \{\tau_1, \ldots, \tau_m\}$ of *transitions* with the transition relation T of an pSTS. With each transition τ_j we associate a pair of state propositions P_j and Q_j and we call P_j a *precondition* and Q_j a *postcondition* of transition τ_j. We assume the existence of a satisfaction relation $\models_P$ which relates assertions about the system state to system states for a given pSTS P[4]. In particular, we write $s \models p$ iff state s satisfies state-proposition p[5]. Now, in order to relate states s and s' we say that $(s, s') \in T$ iff $(\exists \tau_j \in \mathcal{T}_T)(s \models P_j \wedge s' \models Q_j)$. Let $\sigma = s_0, \ldots, s_k$ denote a finite sequence of states. We call this sequence *admissible* iff $(\forall 0 \leq j < k)((s_j, s_{j+1}) \in T)$. This definition extends to infinite sequences in the obvious way. A state s_k is a *reachable* state iff the sequence $\sigma = s_0, \ldots, s_k$ is admissible and $s_0 \models C$, i.e. s_0 is the initial state. In state formulas, when referring to states s and s' with $(s, s') \in T$ we sometimes denote $s[v]$ by v and $s'[v]$ by v'. In order to express that a transitions τ_k is *enabled* in a state s we write $s \models en(\tau_k)$ iff $s \models P_k$. For a pair of states (s, s') we say the transition τ_l has been *taken* iff $s \models en(\tau_l)$ and $s' \models Q_l$. We denote this by $ta(s, s', \tau_l)$ Let the variables X and Y range over the queues of a pSTS, i.e. over sequences of signal types, and A over signal types. The concatenation of a sequence and a singleton element is expressed by juxtaposition. For a signal queue X and a signal type A the term XA describes a sequence where A is the *last* element. Conversely, AY describes a sequence where A is the *first* element.

3.2. Interpreting SDL-Processes as pSTS

We now explain the mapping of an SDL process to the components of a pSTS. So-called *transitions* in an SDL specification describe the change of processes control from one symbolic state to a symbolic successor state. In the example in Table 1 the two symbolic states are `S1` and `S2`, hence for the corresponding pSTS $S = \{S1, S2\}$. The body of a transition consists of different sorts of statements, like assignments, decisions, communication statements, etc. In order to describe the state of the system before and after the execution of a transition we assign pre- and postconditions to every transition. In a few cases, when the transition body has a trivial structure, the determination of pre- and post-conditions is straightforward. However, as we shall see later, we also need to treat more complex transition structures differently.

[4] We omit the reference to P when this is clear from the context.

[5] We will not define all details of the relation $\models$ formally and refer the reader to [20].

```
STATE S1;
INPUT(A);
TASK x := y + 1;
NEXTSTATE S2;
```

Table 1
SDL Transition I

τ_j	P_j	Q_j
τ_1	$\pi = S1 \wedge Q = AX$	$\pi' = S2 \wedge Q' = X \wedge x' = y + 1$
τ_2	$\pi = S1 \wedge Q = CX \wedge C \neq A$	$\pi' = S1 \wedge Q' = X$

Table 2
pSTS predicates for Transition I

Formal Treatment of `INPUT` Statements, Control Flow, and Variable Assignments. For the time being we only consider local systems, we do not yet interpret effects of communication and only define a meaning of `INPUT` statements. Surprisingly, `INPUT` statements have a purely local semantics, namely to remove the signal at the head of the input queue and assign its value to a local variable. Table 2 shows the mapping of an SDL transition to transitions τ_j of a corresponding pSTS. More precisely, when executing a transition associated with an `INPUT(X)` statement, the process first checks whether the signal at the head of its input queue is of type `X`. If this is true the process consumes the signal by removing it from the head of the queue and assigning its value to a local variable with the name `X`. However, if the signal at the head of the queue does *not* have the expected type, then the message is removed from the head of the queue, discarded, and the same `INPUT` statement is re-enabled. We therefore need to split the treatment of `INPUT` statements into two logical cases, the first being the one where the expected signal type is not at the head of the queue, and the second where the expected signal is at the head. We treat transitions with `INPUT` statements as two transitions which are mutually exclusive (see transitions τ_1 and τ_2 in Table 2). The logical exclusion is encoded by the test $Q = AX$ which is *true* in case the head of the input queue contains the message of expected type `A`, and the test $Q = CX \wedge C \neq A$ which evaluates to *true* iff this is not the case. Attention has also to be paid to the *control flow* in a transition. If we consider a transition which brings a process from symbolic state `S1` into symbolic state `S2`, then this can be interpreted as though control lies in code location `S1` before execution of the transition, and in location `S2` afterwards. We defined a particular variable π to range over code locations, called symbolic states, and we use this variable to formulate pre- and postconditions characterising the control flow inside an SDL process (see the use of variable π in Table 2). *Variable assignments* are treated in a very standard way, as for example, described in [20]. Let x and y denote variables in a state s, let x' and y' denote these variables in the successor state s', and let the system transit from s to s' through the execution of a statement `y:= x + 1`. We describe this transition by the postcondition $y' = x + 1$ which is required to hold in state s' (see Tables 1 and 2 for the postcondition describing the the update of variable x).

Formal Treatment of `DECISION` Statements. We decompose a `DECISION P(x)` statement into two, again mutually exclusive transition alternatives. The first is that the decision predicate holds, namely $P(x)$ is *true*, the second is that $P(x)$ is *not* true. As an example see the treatment of the decision in Table 3 in Table 4.

Handling Iterative Transitions. So far we assumed that the symbolic states in the set S are identical to the symbolic states used in the SDL specification. However, SDL transitions may have iterative structure, achieved by a goto and labeling mechanism (the

```
STATE S1;
INPUT(A);
DECISION D(A);
 (true):
   NEXTSTATE S2;
 (false):
   NEXTSTATE S3;
ENDDECISION;
```

Table 3
SDL Transition II

τ_j	P_j	Q_j
τ_1	$\pi = S1 \wedge Q = AX \wedge D(A)$	$\pi' = S2 \wedge Q' = X$
τ_2	$\pi = S1 \wedge Q = AX \wedge \neg D(A)$	$\pi' = S3 \wedge Q' = X$
τ_3	$\pi = S1 \wedge Q = CX \wedge C \neq A$	$\pi' = S1 \wedge Q' = X$

Table 4
pSTS predicates for Transition II

goto statement is called `JOIN` in SDL, see Table 6). Therefore we need to abandon the idea that a transition in an SDL process leads from one symbolic state to a symbolic successor state, as for example suggested in [8]. We need to allow cyclic control flow structures and suggest introducing *auxiliary* symbolic states which correspond to the target locations in the control flow to which a process jumps back or forth when executing `JOIN statements`. In the example in Table 6 we introduced an additional symbolic state `S1-1`, corresponding to the point of control which is reached when jumping to label `l1` (we introduced a comment `/* S1-1 */` in the SDL code at the location corresponding to auxiliary state *S1-1*). The transitions τ_4 and τ_5 represent cases in which control lies in the auxiliary symbolic state `S1-1`.

```
STATE S1;
INPUT(A);
/* S1-1 */
l1:
DECISION D(A);
 (true):
   NEXTSTATE S2;
 (false):
   OUTPUT(B);
   TASK A:=A-1;
   JOIN l1;
ENDDECISION;
```

Table 5
SDL Transition III.

τ_j	P_j	Q_j
τ_1	$\pi = S1 \wedge Q = CX \wedge C \neq A$	$\pi' = S1 \wedge Q' = X$
τ_2	$\pi = S1 \wedge Q = AX \wedge D(A)$	$\pi' = S2 \wedge Q' = X$
τ_3	$\pi = S1 \wedge Q = AX \wedge \neg D(A)$	$\pi' = S1 - 1 \wedge Q' = X \wedge A' = A - 1$
τ_4	$\pi = S1 - 1 \wedge D(A)$	$\pi' = S2$
τ_5	$\pi = S1 - 1 \wedge \neg D(A)$	$\pi' = S1 - 1 \wedge A' = A - 1$

Table 6
pSTS for Transition III

pSTS and Extended Finite State Machines. The derivation of an EFSM from a pSTS is straightforward. For the example in Figures 5 and 6 the resulting EFSM would have 3 states ($S1$, $S1 - 1$ and $S2$), and 5 transitions, corresponding to τ_1 to τ_5.

3.3. State Propositions INPUT and OUTPUT

The state predicates we defined so far allow us to specify formulas referring to the current point of control (e.g. $\pi = S1$) or on the state of data variables (e.g. $Q = AX \wedge A =$ `DR`). However, sometimes one would much rather specify properties of events to happen, in particular referring to communication events and environment interactions, i.e. input or output of signals that are about to take place or that have just been executed. We therefore introduce state predicates which indicate which transition has been taken as a last step in a computation, and whether this transition entailed any communication events. Technically, we introduce two relations, *inlabel* and *outlabel*, which label the transitions of the pSTS with the `INPUT` or `OUTPUT` statements which are executed during the course of a transition. We omit the straightforward technical construction of this labeling here. In the example in Tables 5 and 6, we see that for example $inlabel(\tau_3) = \{\texttt{INPUT(A)}\}$ and $outlabel(\tau_3) = \{\texttt{OUTPUT(B)}\}$. Let $s = s_1, s_2, \ldots$ be an admissible state sequence for a given pSTS, and let $\mathcal{T}_T$ denote the set of transitions for this pSTS. We say that $s_i \models INPUT(A)$ iff $(\exists \tau \in \mathcal{T}_T)(ta(s_{i-1}, s, \tau) \wedge (\texttt{INPUT(A)} \in inlabel(\tau)))$, and $s_i \models OUTPUT(A)$ iff $(\exists \tau \in \mathcal{T}_T)(ta(s_{i-1}, s, \tau) \wedge (\texttt{OUTPUT(A)} \in outlabel(\tau)))$ which augments these labels to state propositions.

3.4. Global State Transition Systems

SDL specifications consist of collections of concurrent SDL processes. We say that the Global State Transition System (GSTS) G_P corresponding to an SDL specification P is a tuple $G_P = (P^0, \ldots, P^n)$ where each P^i for $i = 1, \ldots, n$ is a pSTS. P^0 (which represents the environment behaviour) is not a full pSTS, it only consists of an input and an output alphabet and an input queue. P^0 has no state and we rely on the facilitating assumption that P^0 will provide any of the other processes with input signals whenever they wish to consume any such signal, and that P^0 instantly consumes any signal it receives from any process of the SDL system. To model the SDL communication mechanism there is one input queue per SDL process. We interpret the sending of a signal A from a process P^1 to a process P^2, indicated by an `OUTPUT(A)` statement, such that a signal of type A is appended to P^2's input queue, Q^2. We slightly simplify the SDL mechanism of mapping of an output signal to a receiving process by assuming that a signal `A` is sent from a process P^i to a process P^j iff $A \in I^j$. Furthermore, we require $(\forall i = 1, \ldots, n)(\forall a \in O^i)(\exists j \neq i)(a \in I^j)$ and $(\forall i = 1, \ldots, n)(O^i \cap I^i = \emptyset)$. As we saw in Section 3.2, the execution of an `INPUT(A)` statement (the *signal-consumption*) represents an action purely local to an SDL process.

Transition Predicates for OUTPUT statements. The execution of an `OUTPUT` statement involves a non-local action. The execution of the statement involves a local event, the sending itself, and a remote event, the receiving of the message by adding it to the receiving process' input queue. Therefore, one can not formalize the respective transitions by state propositions that solely refer to state variables of only one process. Table 8 presents a simple example of a two-process SDL specification $P = (P^0, P^1, P^2)$ where transition τ_1^1 describes both the state change in P^1 and the appending of the signal B to the input queue of P^2. Although strictly speaking this transition also changes the state of process P^2, we consider transition τ_1^1 to be a transition belonging to process P^1.

Global System States, Transitions, Global State Sequences, and the Satisfaction Relation. Let $G_P = (P^0, \ldots, P^n)$ denote the GSTS for an SDL specification P. We say that the vector $s = (s^1, \ldots, s^n)$ is a *global system state* (GSS) of the SDL specification P iff s^i is a state of pSTS P^i for all $i = 1, \ldots, n$. In the course of each change of the GSS exactly one pSTS changes its local system state, hence we assume an interleaving of local system state changes to model the concurrency in an SDL specification. This means that in a given GSS s, a demon decides nondeterministically which

```
PROCESS P1;        PROCESS P2;
  STATE S1;          STATE S3;
  INPUT(A);          INPUT(B);
  OUTPUT(B)          NEXTSTATE S3;
  NEXTSTATE S2;
```

Table 7
SDL specification

τ_j^1	P_j^1	Q_j^1
τ_1^1	$\pi^1 = S1 \wedge Q^1 = AX$ $\wedge Q^2 = Y$	$\pi'^1 = S2 \wedge Q'^1 = X$ $\wedge Q'^2 = YB$
τ_2^1	$\pi^1 = S1 \wedge Q^1 = CX \wedge C \neq A$	$\pi'^1 = S1 \wedge Q'^1 = X$

Table 8
Predicates describing SDL specification

out of all enabled transitions in all pSTS of an SDL specification is going to be executed next, which defines the successor GSS s'. Let $\sigma = s_0, \ldots, s_k$ denote a finite sequence of GSS. We call this sequence *admissible* iff $(\forall 0 \leq j < k)(\exists \tau_l^i)((s_j^i, s_{j+1}^i) \in T^i))$. This definition extends to infinite sequences in the obvious way. Also, the interpretation of the state propositions *en*, *ta*, *INPUT* and *OUTPUT* extend in the obvious way from pSTS states to GSS. Based on the above definitions we may now define a satisfaction relation $\models_{\mathrm{SDL}}$ for SDL specifications. Let P an SDL specification and let Σ_P^ω the set of all infinite sequences of GSS of P. For a $\sigma \in \Sigma_P^\omega$ we write $\sigma \models_{\mathrm{SDL}} P$ iff σ is an admissible sequence with respect to P.

4. USING TEMPORAL LOGIC FOR SDL SPECIFICATIONS

The characterisation of properties by the use of temporal logic is accomplished by interpreting the temporal logic specification such that the models satisfying all formulas determine the set of admissible state sequences of the system. Now, as we have seen in Section 3, SDL specifications also specify admissible sequences of states. Temporal logic formulas can be thought of as filters on the admissible sequences specified by the SDL specification and therefore can be used to specify those real-time and liveness properties inexpressible in SDL. A crucial point is the selection of a suitable temporal logic language. We will use a temporal logic similar to the logic described in [20], called Propositional Temporal Logic (PTL) and a real-time extensions based on PTL, called Metric Temporal Logic (MTL), see [14] and [16]. However, other temporal logics can be linked to SDL specifications in very much the same way.

A State Proposition Language. We assume that the state propositions we use in complementary temporal logic formulas all refer to observable components of the system state, and we use, in particular, the following state propositions for an SDL specification P: 1. *Actual State:* let $S = S_1^i, \ldots, S_n^i$ denote the symbolic states for a given process P^i of P, then $at\_S_k^i$ denotes the state proposition that the i-th component of the global system state is in symbolic state S_k^i, i.e. $\pi^i = S_k^i$. 2. *Input and output:* we use the state propositions *INPUT* and *OUTPUT* as defined above to denote that we are in a state where an input or an output of a signal has just occurred in the last GSS transition. 3. *Data:* we allow the reference to visible data variables and allow standard comparison operators on the variables. We allow state formulas to be constructed by using boolean operators between state propositions and we call composed state formulas *state predicates.*

Temporal Logic. The Propositional Temporal Logic (PTL) we use here is a linear time temporal logic taken from [20] to which we refer the reader for a complete syntax and semantics definition. In addition to the standard operators of PTL as defined in

[20] we define a *strong eventuality* operator $\diamondplus$ so that $\diamondplus p$ holds in some *future* state s, formally $s_i \models \diamondplus p$ iff $(\exists j > i)(s_j \models p)$. The formal semantics of PTL define a satisfaction relation $\models_{\mathrm{PTL}}$. An execution sequence $\sigma = s_0, \ldots$ of states s_i satisfies a formula ϕ iff ϕ holds in s_0, and we write $\sigma \models_{\mathrm{PTL}} \phi$. We say that a system satisfies a formula ϕ iff all its execution sequences satisfy ϕ. We use an extension of PTL for the specification of real-time requirements, called *metrical temporal logic* (MTL). For a complete formal definition of the syntax and semantics of MTL we refer the reader to [5] and [14, Section 3.4]. The models over which we interpret MTL formulas are timed observation sequences $o = o_1, \ldots$ (see [5]) where each o_i corresponds to a pair s_i, l_i in which s_i denotes a state and l_i denotes a numeric value, called a time stamp. We only consider instantaneous state changes. We assume l_i sequences to be monotonic, as well as a finite precision of our clocks, i.e. we assume that every state change coincides with a click of the clock from which we derive the timed observation. Therefore the set of natural numbers, N suffices as a domain for the interval expressions [5]. When selecting a time model we have to find one which is suited to comply with the SDL interleaving semantics approach. In other words, for GSS s_1, s_2 and s_3 of a given SDL specification, assume that both $s_1, s_2, s_3, \ldots$ and $s_1, s_3, s_2, \ldots$ are admissible sequences in the untimed model. If we now want to express that both s_2 and s_3 may occur at the same time (which means that they have the same time stamp) in any order, we have to consider both the timed observation sequences $(s_1, l_1) \rightarrow (s_2, l_2) \rightarrow (s_3, l_3) \rightarrow \ldots$ as well as the sequence $(s_1, l_1) \rightarrow (s_3, l_2) \rightarrow (s_2, l_3) \rightarrow \ldots$ to be admissible and to allow that $l_2 = l_3$. Hence, for our time model we assume the sequence l_i to be *weakly*-monotonic [5].

Informally, MTL contains formulas of the form $\diamond_I \phi$ which assert that *one* of the following states within the time-interval described by expression I is a state which satisfies ϕ. Formulas of the form $\Box_I \phi$ assert that *all* states in the time-interval described by I satisfy ϕ. The expression I describes an either open or closed interval over the time domain and we sometimes use semi-algebraic expressions to refer to these intervals. We write $o \models_{\mathrm{MTL}} p$ iff the sequence o satisfies the MTL formula p.

Complementary Specifications. Assume we have an SDL specification P and a set of formulas M in MTL. Now, P and M are *complementary* specifications if we require from the specified system that for all its timed observation sequences $o = (s_0, t_0), \ldots$ the following condition holds: $s \models_{\mathrm{SDL}} P \wedge o \models_{\mathrm{MTL}} M$.

5. SPECIFYING DELAYS

In this Section we will exemplify the application of complementary specifications to delay related real-time requirements.

Liveness and Progress in the INRES example. Let us consider the INRES connection establishment example in Figure 1 again and use a complementary specification in order to guarantee progress of the system. First, we will look at a liveness requirement that when a request for a connection establishment has been issued by sending a `CR` message, then the process `Initiator` will eventually receive either a `CC` or a `DR` signal, or it will eventually issue a `IDISind` signal to the service user. As pointed out earlier, apart from trivial liveness properties SDL does not have the expressiveness to capture more complex liveness properties like the one stated above. However, complementing the

SDL specification of the INRES example with the following MTL formula will express the desired liveness property:

$$\Box(OUTPUT(CR) \supset \Diamond(INPUT(CC) \vee INPUT(DR) \vee OUTPUT(IDISind))).$$

Now, as we argued in Section 2, it is important to assert that any of these responses to the sending of the `CR` signal happens within a reasonable period of time, say within t time units. In the SDL specification, the timer `T` has been used to require this, but we have argued above why the usage of the timer in this context cannot guarantee this condition to hold. Therefore we specify a real-time bounded response requirement using MTL in the following way:

$$\Box(OUTPUT(CR) \supset \Diamond_{\leq t}(INPUT(CC) \vee INPUT(DR) \vee OUTPUT(IDISind))).$$

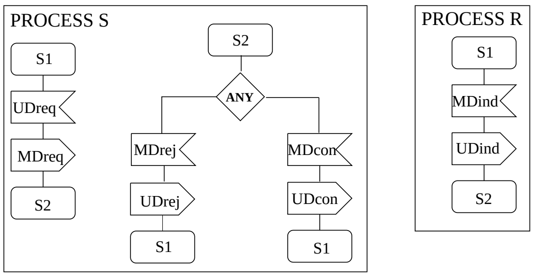

Figure 2. SDL Specification of SRS example.

Maximal and Minimal Service Response Time. Consider the simple *Sender / Recei-ver Service* (SRS) specified in Figure 2. A user of the service requests the transmission of some data by sending a `UDreq` signal to the sender process `S` which in turn requests the transmission of the data from an (unspecified) medium service `M` by sending a `MDreq`. The medium service is unreliable. However, in case the transmission is successful the medium service will deliver the data to the receiver process `R` by means of an `MDind` message, and the receiver delivers the data to its user process. We assume that the medium service is capable of reliably indicating to the sender process by means of an `MDcon` signal whether the data has been delivered successfully to the receiver process, or by an `MDrej` that this is not the case. Successful delivery will be indicated to the service user of process `S` by an `UDcon` signal, and unsuccessful delivery by an `UDrej` signal. Now, we may like to require that if the service process `S` has received a `UDreq`, it will issue within at most t_1 time units either an `UDcon` or a `UDrej` signal to the service user in order to indicate successful or unsuccessful delivery of data. We describe this requirement by

$$\Box(OUTPUT(UDreq) \supset \Diamond_{\leq t_1}(INPUT(UDcon \vee UDrej))).$$

In some situations it may also be interesting to state that between two events there is a *minimal* time that will always pass. The following formula states that if after the request the data will eventually be successfully delivered by the medium service by issuing a `MDind` signal, then this will happen at least t_2 time units after the request has been issued.

$$\Box((INPUT(MDreq) \wedge \Diamond OUTPUT(MDind)) \supset \Box_{<t_2} \neg INPUT(MDind))$$

Delay Jitter. Successive data units routed through a complex packet switched network may be subject to varying delays over time. The ATM service is, as one example, prone to this sort of delay variation [18], caused by changing network load or by routing successive cells on different routes. However, in particular multimedia applications which need to reconstruct continuous signals require data to be delivered within a time interval of around the mean value of the transmission delay, depending on the coding scheme used. The delay variance is called *delay jitter* and is formally defined as follows: let d_{min} denote the minimal and let d_{max} denote the maximal delay between sending and receiving of a sequence of transmitted data units, then $J = d_{max} - d_{min}$ denotes the delay jitter. We use again the SRS example specified in SDL (see Figure 2), but this time we assume that the underlying medium service is reliable. We assume that d_{min} and d_{max} are known constant values. The requirement bounding the delay jitter for the user interface service can then be specified by the formula

$$\Box(INPUT(UDreq) \supset (\Box_{\leq d_{min}} \neg OUTPUT(UDind)) \wedge (\Diamond_{\leq d_{max}} OUTPUT(UDind))).$$

Isochronous sending and receiving. Isochronicity is a characteristic feature of many multimedia applications. The isochronicity we refer to means that events, for example sending and receiving of data units, occur periodically at equally distanced points of time. Again, we refer to the SRS example. Isochronous sending is a characteristic of a traffic source, in particular of the coding scheme and algorithm used there. In SRS, the characterization of isochronous sending of the application served by process `S` reads

$$\Box(INPUT(UDreq) \supset (\neg \Diamond_{<t} INPUT(UDreq) \wedge \Diamond_{=t} INPUT(UDreq))).$$

The receiver may have to rely on having successive data units available at isochronous moments in time. This may be expressed in a way very similar to the isochronous send characterization, namely as

$$\Box(OUTPUT(UDind) \supset (\neg \Diamond_{<t} OUTPUT(UDind) \wedge \Diamond_{=t} OUTPUT(UDind))).$$

6. SPECIFYING QOS-MECHANISMS

QoS Negotiation. Assume the SRS example to be embedded in a mechanism which allows the negotiation of certain QoS guarantees with the (unspecified) underlying medium service. We are not interested in the mechanism itself, but in specifying the effect that a successful renegotiation has. Assume that the process `S` is capable of requesting an increase in the *medium service delay* QoS parameter and that when the increase request is granted by the medium service (indicated by an $INPUT(MINCcon)$ inside `S`), the delivery bound for successfully delivered packets is limited to t_4. Hence, we require that whenever $INPUT(MINCcon)$ has been executed, the delivery delay is henceforth limited to t_4:

$$\Box(OUTPUT(MINCcon) \supset$$

$$\Box((INPUT(MDreq) \wedge \Diamond OUTPUT(MDind)) \supset \Diamond_{\leq t_4} OUTPUT(MDind))).$$

Reaction to QoS Violation. It may be useful to specify a desired reaction on the violation of QoS requirement without implying that the violation invalidates the respective system behaviour. Let us assume that we monitor the response time behaviour of the medium service in SRS and that we require that if the medium service does not respond by either `MDind` or `MDrej` within t_7 time units after the `MDreq` has been issued, an `ALARM` signal is to be issued after at most t_8 time units, with $t_8 > t_7$. We specify this as

$$\Box(\neg(OUTPUT(MDreq) \supset \Diamond_{\leq t_7}(INPUT(MDind) \vee INPUT(MDrej))) \supset$$

$$(\Box_{\leq t_7} \neg OUTPUT(ALARM) \wedge \Diamond_{\leq t_8} OUTPUT(ALARM))).$$

Delay Jitter Compensation. Guaranteeing a bound on the delay jitter does not yet guarantee isochronous delivery of messages to a user, even if the source is sending data isochronously. In order to compensate the residual delay jitter and to guarantee an isochronous delivery of data units to a user it is often suggested to use a jitter compensation buffer between the network service and the user (e.g. the ATM *playout* buffer [18]). Assume that the process `R` in SRS has the functionality of a playout buffer. Then, `R` accepts the possibly non-isochronous but jitter-bounded data stream from the Medium service by `MDind` signals. Every signal will be delayed for a minimum time span of d_1 time units. This means that the first data units in a stream will fill the buffer up to a certain threshold number. Then, at latest $t_2 > t_1$ time units after the arrival at the buffer the data units will be delivered to the user by means of a `UDind` signal. The delivery of successive `MDind` signals then occurs isochronously with an inter-signal delivery time of p, which ideally should correspond to the inter-send event time at the sender in order to ensure an isochronous traffic with identical inter-send times on the sender as on the receiver side. The jitter compensation requirement for the process `R` reads

$$\Box(INPUT(MDind) \supset ((\Box_{\leq t_1} \neg OUTPUT(UDind) \wedge \Diamond_{\leq t_2} OUTPUT(UDind)))$$

$$\wedge \Box(OUTPUT(UDind) \supset \dot{\Diamond}_{=p} OUTPUT(UDind)).$$

7. QOS VERIFICATION

So far requirements capture has been our main interest. However, we will now point at verification questions arising from the use of complementary specifications in the described manner. Let us consider the SRS example again and let us assume that SRS has been translated into a logic specification $\mathcal{S}$. Furthermore, assume the system performance to be described by the following minimal response time formula:

$$\mathcal{P} : \Box((INPUT(MDreq) \wedge \Diamond OUTPUT(MDind)) \supset \Box_{< t_5} \neg OUTPUT(MDind)).$$

Let a QoS requirement be described by the following formula:

$$\mathcal{Q} : \Box(OUTPUT(UDreq) \supset \Diamond_{\leq t_1}(INPUT(UDcon) \vee INPUT(UDrej))).$$

This gives rise to a verification problem, namely the question, whether based on $\mathcal{S}$ and $\mathcal{P}$ the QoS requirement $\mathcal{Q}$ can at all be satisfied, hence whether the assertion $\mathcal{P} \wedge \mathcal{S} \supset \mathcal{Q}$

holds. Intuitively, the answer depends amongst others on the choice of values for t_1 and t_5. To formally establish this conjecture it is necessary employ adequate formal verification methods or model checking algorithms (see for example [1] for a formal verification approach, and [3] for a real-time model checking algorithm).

8. CONCLUDING REMARKS

We described a method for the specification of real-time constraint based QoS requirements for SDL specifications. Starting point was an analysis of SDL specifications and the insight that the SDL timer mechanism is unsuitable to express the important class of bounded response real-time requirements. We mapped SDL specifications to global state transition systems and showed how SDL system states and state transitions can be described in terms of logic formulas over state propositions. Next we connected standard real-time temporal logic specifications to SDL specifications and defined so-called complementary specifications. We then gave some general example specification for QoS requirements for SDL specifications. Examples included delay bounds, delay jitter bounds, and isochronicity requirements. We then showed how QoS mechanisms can be specified in the framework of our method, in particular QoS negotiation and QoS monitoring, and hinted at arising formal verification problems.

Acknowledgements. The author wishes to thank all those who have commented on earlier versions of this paper, in particular Reinhard Gotzhein, Dieter Hogrefe, Peter Ladkin and Tony Savor.

REFERENCES

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *[12]*, pages 1–27, 1992.
2. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
3. R. Alur, C. Courcoubetis, and D. L. Dill. Model checking for real-time systems. In *Fifth Annual Symposium on Logic in Computer Science*, pages 414–425, 1990.
4. R. Alur and D. Dill. The theory of timed automata. In *[12]*, pages 45–73, 1992.
5. R. Alur and T. A. Henzinger. Logics and models of real-time: A survey. In *[12]*, pages 45–73, 1992.
6. R. Alur and T. A. Henzinger. Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*, pages 1–30, 1994. To appear.
7. F. Bause and P. Buchholz. Protocol analysis using a timed version of SDL. In J. Quemada, J. Mañas, and E. Vazquez, editors, *Formal Description Techniques, III*, pages 269–285. North-Holland, 1991.
8. F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.
9. D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.
10. CCITT. Recommendation Z.100: CCITT Specification and Description Language (SDL). CCITT, Geneva, 1992.
11. K.-T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th Design Automation Conference*

DAC-93, pages 86–91, 1993.
12. J. W. de Bakker, C. Huizing, W.P. de Roever, and G.Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
13. S. R. Faulk and D. L. Parnas. On synchronisation in hard-real-time systems. *Communications of the ACM*, 31(3):274–287, March 1988.
14. T. A. Henzinger. *The Temporal Specification and Verification of Real-Time Systems*. Phd thesis, Stanford University, Department of Computer Science, August 1991. Also published as Report No. STAN-CS-91-1380.
15. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.
16. R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Technical University of Eindhoven, 1989.
17. A. S. Krishnakumar. Reachability and recurrence in extended finite state machines: Modular vector addition systems. In C. Courcoubetis, editor, *Computer Aided Verification: Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 111–122. Springer Verlag, 1993.
18. J.-Y. Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Network and ISDN Systems*, 24:279–309, 1992.
19. M. T. Liu. Protocol engineering. In M. C. Yovitis, editor, *Advances in Computers*, volume 29, pages 79–195. Academic Press, Inc., 1989.
20. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.
21. P. M. Melliar-Smith. Extending interval logic to real-time systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the Conference on Temporal Logic in Specifications, 1987*, volume 398 of *Lecture Notes in Computer Science*, pages 224–242. Springer-Verlag, 1989.
22. M. Merritt, F. Modugno, and M. R. Tuttle. Time-constrained automata. In *CONCUR 91: 2nd International Conference on Concurrency Theory, Lecture Notes in Computer Science 527*, 1991.
23. H. Saito, T. Hasegawa, and Y. Kakuda. Protocol verification system for SDL specifications based on acyclic expansion algorithm and temporal logic. In K. R. Parker and G. A. Rose, editors, *Formal Description Techniques, IV*, pages 511–526. North-Holland, 1992.
24. K. J. Turner, editor. *Using Formal Description Techniques*. John Wiley & Sons, 1993.

BIOGRAPHY

The author received his Master's Degree in Computer Science (Diplom-Informatiker) from the University of Hamburg, Germany, in 1991, and his Ph.D. degree from the University of Berne, Switzerland, in 1995. Currently, he is an Assistant Professor at the Electrical and Computer Engineering Department of the University of Waterloo, Canada. His research interests are in the area of software engineering for telecommunications and distributed systems, in particular in specification, verification, implementation and reliability aspects.