Modelling Timeouts without Timelocks

Howard Bowman*

Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, United Kingdom. (Email: H.Bowman@ukc.ac.uk)

Abstract. We address the issue of modelling a simple timeout in timed automata. We argue that expression of the timeout in the UPPAAL timed automata model is unsatisfactory. Specifically, the solutions we explore either allow timelocks or are prohibitively complex. In response we consider timed automata with deadlines which have the property that timelocks cannot be created when composing automata in parallel. We explore a number of different options for reformulating the timeout in this framework and then we relate them.

1 Introduction

A timeout is perhaps the most basic and widely arising specification structure in real-time systems. For example, they arise frequently when modelling communication protocols and work on enhancing "first generation" specification techniques with real-time has frequently been directly motivated by the desire to model timeouts in communication protocols [9].

From within the canon of timed specification notations, timed automata [1] are certainly one of the most important. One reason for this is that powerful real-time model checking techniques have been developed for timed automata, as exemplified by the tools, UPPAAL [2], Kronos [7] and HyTech [8].

However satisfactorily modelling timeout structures proves surprisingly difficult in timed automata. Broadly, problems arise because it is difficult to define timeout behaviour in a manner that avoids the possibility of *timelocks*. By way of clarification:

> *we say that a system is timelocked if it has reached a state from which no path can be found to a time passing transition.*

Timelocks are highly degenerate situations because they yield a *global* blockage of the systems evolution. For example, if a completely independent component is composed in parallel with a system that is timelocked, then the entire composition will inherit the timelock. This is quite different from a classic (local) deadlock, which cannot affect the evolution of an independent process. These characteristics of timelocks will be illustrated in section 2.

* Howard Bowman is currently on leave at VERIMAG, Centre Equation, 2 rue Vignate, 38610 GIERES, France and CNR-Istituto CNUCE, Via S. Maria 36, 56126 - Pisa - Italy with the support of the European Commission's TMR programme.

This paper addresses the issue of how to model timeouts without generating timelocks. We illustrate how the difficulty arises in current timed automata models and then we consider a new timed automata model - Timed Automata with Deadlines (TADs) [3] - which guarantee timelock free parallel composition of automata components.

We begin (in section 2) by presenting background material - we introduce timed automata, discuss the nature of timelocks and outline the timeout behaviour that we require. These requirements have been identified during practical study of the specification and verification of a lip-synchronisation algorithm [6]. Then (in section 3) we discuss possible ways to model the timeout in timed automata. As a typical timed automata model we choose the UPPAAL [2] notation. This is one of the most important timed automata approaches. We argue that none of the legal UPPAAL approaches are completely satisfactory. In particular, avoiding the possibility of timelocks is difficult and leads to prohibitively complex solutions.

In response, (in section 4) we consider how the same timeout behaviour can be modelled in Timed Automata with Deadlines [3]. However, it turns out that the standard TADs approach, as presented in [3], resolves the timelock problem, but introduces a new difficulty which is related to the generation of escape transitions. Consequently, we consider a number of different TAD formulations in section 5 which resolve these difficulties in contrasting ways. Finally, in section 6 we relate these solutions and present a concluding discussion.

2 Background

2.1 Timed Automata

We briefly review some basic timed automata notation.

- A is the set of *completed* (or *basic*) actions.
- $\overline{A} = \{ a?, a! \mid a \in A \}$ is the set of *uncompleted* actions. These give a simple CCS style [10] point-to-point communication which has also been adopted in UPPAAL.
- $A^+ = \overline{A} \cup A$ is the set of *all* actions.
- We use a complementation notation over elements of A^+,

$$\overline{a} = a \quad \text{if } a \in A \tag{1}$$

$$\overline{b?} = b! \tag{2}$$

$$\overline{b!} = b? \tag{3}$$

In addition, we let v, v' etc, range over vectors of processes, which are written, $< l_1, ..., l_n >$, $|v|$ denotes the length of the vector, we use a substitution notation as follows: $< l_1, ..., l_j, ..., l_n > [l/l_j] =< l_1, ..., l_{j-1}, l, l_{j+1}, ..., l_n >$ and we write $v[l'_1/l_1][l'_2/l_2]...[l'_m/l_m]$ as $v[l'_1/l_1, l'_2/l_2, ..., l'_m/l_m]$. We assume the a finite set: $\mathcal{C}$

of clocks which range over $\mathbb{R}^+$ and $\mathcal{CC}$ is a set of clock constraints[1]. An arbitrary element of $\mathcal{A}$, the set of all timed automata, has the form:

$$(L, l_0, T, P)$$

where,

- L is a finite set of locations (these appear as small circles in our timed automata diagrams, e.g. see figure 1);
- l_0 is a designated *start location*;
- $T \subseteq L \times CC \times A^+ \times \mathbb{P}(\mathcal{C}) \times L$ is a transition relation (where $\mathbb{P}(S)$ denotes the powerset of S). A typical element of T would be, $(l_1, \mathtt{g}, \mathtt{a}, \mathtt{r}, l_2)$, where $l_1, l_2 \in L$ are automata locations; $\mathtt{g} \in \mathcal{CC}$ is a guard; $\mathtt{a} \in A^+$ labels the transition; and $\mathtt{r} \in \mathbb{P}(\mathcal{C})$ is a reset set. $(l_1, \mathtt{g}, \mathtt{a}, \mathtt{r}, l_2) \in T$ is typically written, $l_1 \xrightarrow{\mathtt{g},\mathtt{a},\mathtt{r}} l_2$, stating that the automata can evolve from location l_1 to l_2 if the (clock) guard $\mathtt{g}$ holds and in the process action $\mathtt{a}$ will be performed and all the clocks in $\mathtt{r}$ will be set to zero. When we depict timed automata, we write the action label first, then the guard and then the reset set, see e.g. figure 4. Guards that are `true` or resets that are empty are often left blank.
- $P : L \rightarrow CC$ is a function which associates a progress condition (often called an invariant) with every location. Intuitively, an automata can only stay in a state while its progress condition is satisfied. Progress conditions are shown adjacent to states in our depictions, see e.g. figure 2.

Timed automata are interpreted over time/action transition systems which are triples, $(S, s_0, \rightarrow)$, where,

- S is set of states;
- s_0 is a start state;
- $\rightarrow \subseteq S \times \mathtt{Lab} \times S$ is a transition relation, where $\mathtt{Lab} = A^+ \cup \mathbb{R}^+$. Thus, transitions can be of one of two types: *action transitions*, e.g. $(s_1, \mathtt{a}, s_2)$, where $\mathtt{a} \in A^+$ and *time passing transitions*, e.g. $(s_1, \mathtt{x}, s_2)$, where $\mathtt{x} \in \mathbb{R}^+$ and denotes the passage of $\mathtt{x}$ time units. Transitions are written:

$$s_1 \xrightarrow{\mathtt{a}} s_2 \quad \text{respectively} \quad s_1 \xrightarrow{\mathtt{x}} s_2$$

A *clock valuation* is a mapping from clock variables $\mathcal{C}$ to $\mathbb{R}^+$. For a clock valuation u and a delay d, $u \oplus d$ is the clock valuation such that $(u \oplus d)(x) = u(x) + d$ for all $x \in \mathcal{C}$. For a reset set r, we use $r(u)$ to denote the clock valuation u' such that $u'(x) = 0$ whenever $x \in r$ and $u'(x) = u(x)$ otherwise. u_0 is the clock valuation that assigns all clocks to the value zero.

The semantics of a timed automaton $A = (L, l_0, T, P)$ is a time/action transition system, $(S, s_0, \rightarrow)$, where S is the set of all pairs $< l, u >$ such that $l \in L$

[1] The form that such constraints can take is typically limited, however since we are not considering verification this is not an issue for us.

and u is a clock valuation, $s_0 = < l_0, u_0 >$ and $\rightarrow$ is given by the following inference rules:-

$$\frac{l \xrightarrow{g,a,r} l' \quad g(u)}{< l, u > \xrightarrow{a} < l', r(u) >} \qquad \frac{\forall d' \leq d \,.\, P(l)(u \oplus d')}{< l, u > \xrightarrow{d} < l', u \oplus d >}$$

We assume our system is described as a network of timed automata. These are modelled by a process vector[2] denoted, $|| < A_1, ..., A_n >$. If $\forall i (1 \leq i \leq n) \,.\, A_i = (L_i, l_{i,0}, T_i, P_i)$ then the product automaton, which characterises the behaviour of $|| < A_1, ..., A_n >$ is given by,

$$(L^p, l_0^p, T^p, P^p)$$

where $L^p = \{\, ||v \mid v \in L_1 \times ... \times L_n \,\}$, $l_0^p = || < l_{1,0}, ..., l_{n,0} >$, T^p is as defined by the following two inference rules and $P^p(|| < l_1, ..., l_n >) = P_1(l_1) \wedge ... \wedge P_n(l_n)$.

$$\frac{l_i \xrightarrow{g_i, a?, r_i} l_i' \quad l_j \xrightarrow{g_j, a!, r_j} l_j'}{||v \xrightarrow{g_i \wedge g_j, a, r_i \cup r_j} ||v[l_i'/l_i, l_j'/l_j]} \qquad \frac{l_i \xrightarrow{g,a,r} l_i' \quad a \in A}{||v \xrightarrow{g,a,r} ||v[l_i'/l_i]}$$

where $1 \leq i, j \leq |v|$, $i \neq j$.

2.2 Timelocks

We can formulate the notion of a timelock in terms of a testing process. Consider, if we take our system which we denote `System` and compose it completely independently in parallel with the timed automaton, `Tester`, shown in figure 1, where the `zzz` action is independent of all actions in the system. Then for any `x`$\in \mathbb{R}^+$ if the composition `||<Tester(x),System>` cannot perform `zzz` then the system contains a timelock at time `x`.

This last illustration indicates why timelocks represent such degenerate situations - even though the `Tester` is in all respects independent of the system, e.g. it could be that `Tester` is executed on the Moon and `System` is executed on Earth without any co-operation, the fact that the system cannot pass time prevents the tester from passing time as well. Thus, *time really does stop* and it stops everywhere because of a degenerate piece of *local* behaviour.

This is a much more serious fault than a classical (local) deadlock. For example, the automaton `Stop`, also shown in figure 1, generates a local deadlock, however, it cannot prevent an independent process from evolving. In the sequel we will use the term *local deadlock* to denote such a non-timeblocking deadlock.

We consider two varieties of timelock which we illustrate by example, see figure 2,

[2] Although our notation is slightly different, our networks are a straightforward simplification of those used in UPPAAL. The simplifications arise because we do not consider the full functionality of UPPAAL. For example, we do not consider committed locations or data variables.

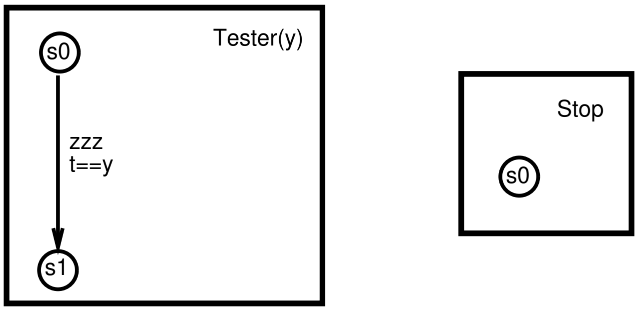

Fig. 1. A tester and a (locally) deadlocked timed automata

1. The first is `System1`; this is a zeno process which performs an infinite number of `aaa` actions at time zero. This system is timelocked at time zero and if we compose it independetly in parallel with any other system, the composite system will not be able to pass time. We call such timelocks *zeno timelocks*.
2. The second is the network `||<System2,System3>`; this composition contains a timelock at time 2, which arises because `System2` must have performed (and thus, synchronised on) action `aaa` by the time `t` reaches 2 while `System3` does not start offering `aaa` until `t` has past 2. Technically the timelock is due to the fact that at time 2 `System2` only offers the action transition `aaa` and importantly, it does not offer a time passing transition. Since the synchronisation cannot be fulfilled the system cannot evolve to a point at which it can pass time. We call such timelocks *composition timelocks*.

The interesting difference between these two varieties of timelock is that the first one locks time, but in the classical sense of untimed systems, is not deadlocked, since actions can always be performed. However, the second reaches a state in which neither time passing or action transitions are possible. Such composition timelocks are the variety we will address in this paper.

2.3 A Bounded Timeout

We describe a rather standard timeout behaviour, which we call a *Bounded Timeout*. The general scenario is that a `Timeout` process is monitoring a `Component` and the timeout should expire and enter an error state if the `Component` does not offer a particular action, which we call `good`, within a certain period of time.

The precise functionality that we want the timeout to exhibit is[3]:

1. *Basic behaviour.* Assuming `Timeout` is started at time t, it should generate a `timeout` action at a time $t+D$ if and only if the action `good` has not already occured. Thus, if action `timeout` occurs, it must occur exactly at time $t+D$

[3] Our presentation here is similar to that in [6]

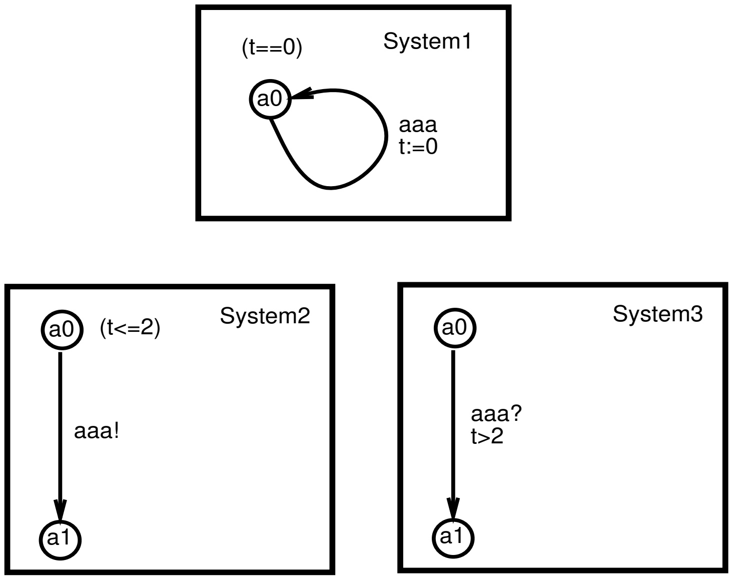

Fig. 2. Timelock Illustrations

and if action `good` occurs, then it must occur at some time from t up to, but not including, $t + D$. Using the terminology of [11] this yields a *strong* timeout. A *weak* timeout would, in contrast, allow a non-deterministic choice between the `good` action and the `timeout` at time $t + D$.

2. *Urgency of* `good` *action.* We also require that if the `good` action is enabled before time $t + D$ then it is taken *urgently*, i.e. as soon as `good` is enabled it happens. This urgency requirement is akin to the "as soon as possible" principle which has been applied in timed process algebra [12].
3. *Timelock Free.* Finally we want our composed system to be free of timelocks, for obvious reasons.
4. *Simple.* We also require that the solution is not "prohibitively" complex.

Notice that in the first two of these requirements, urgency arises in two ways. Firstly, we require that `timeout` is urgent at time $t + D$ and secondly, we require that `good` is urgent as soon as it is enabled. Without the former requirement the timeout might fail to fire even though it has expired and without the latter, even though the `good` action might be able to happen it might nonetheless not occur and thus, for example, the `timeout` may expire even though `good` was possible.

We also emphasize that although our work here was inspired by that in [6], it is somewhat different. In particular, [6] presents a bounded timeout in a discrete time setting, thus, the final time at which the `good` action can be performed and the time of expiry of the `Timeout` are at different discrete time points.

3 Modelling the Bounded Timeout in UPPAAL

In this section we describe the bounded timeout in UPPAAL. However, our discussion is not solely relevant to this notation, and could be extrapolated to timed automata notations in general.

Basic Formulation. We begin by considering the `Timeout` shown in figure 3. This process realises the first requirement that we identified for modelling the bounded timeout - `good` is offered at all times in which `t<D`. Then `timeout` is performed when `t==D`, in which case the system passes into state `a2` which plays the role of an error state. Importantly, the guard `(t<=D)` forces the required urgency on the `timeout` action. Thus, if `good` has not happened earlier, `timeout` *must* happen when `t==D`. Furthermore, it is easy to see that this is indeed a strong timeout - its behaviour is deterministic when `t==D`.

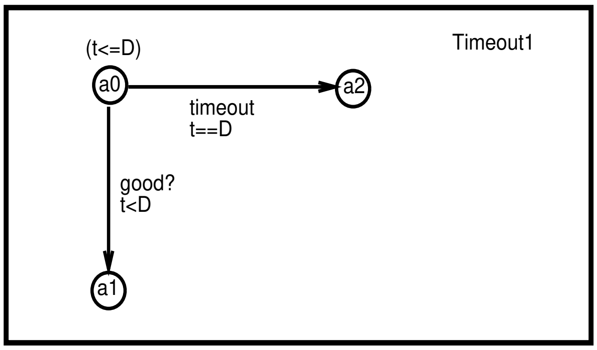

Fig. 3. An UPPAAL Automaton for Timeout1

However on its own, this automaton is not sufficient since nothing forces the `good` action to be taken if it can be. This was our second requirement. For example, consider `Component1` shown in figure 4 which will perform an internal action `tau`[4] at some time `r<=C` and then offer the `good` action. The internal action can be viewed as modelling some internal computation by `Component1`. The completion of which is signalled by offering `good!`. Now if we put `Timeout1` and `Component1` in parallel then even if `good` could occur while `t<D`, it might not be taken. Thus, a possible trace of the system:

```
||<Timeout1,Component1>
```

is, $(\mathtt{tau}, x_1)(\mathtt{timeout}, x_2)$ where, $x_1 < \mathtt{C}$, $x_1 < \mathtt{D}$ and $x_2 = \mathtt{D}$.

[4] In fact, internal transitions are left unlabelled in UPPAAL, however, we abuse notation in order that we can refer to the transition.

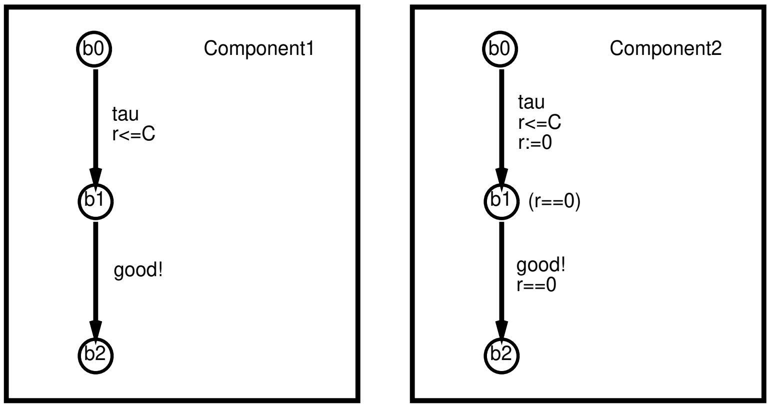

Fig. 4. UPPAAL Automata for Component1 and Component2

Thus, we need some way to make `good` urgent. The standard approach is to enforce urgency in the component. For example, we could use `Component2` shown in figure 4. This automaton will perform the internal action as before and then it *must* immediately perform the `good` action.

Now the problem with the composition:

```
||<Timeout1,Component2>
```

is the relative values of `D` and `C`. In particular, if `C` is larger than `D` then this system can time-block in the following way:-

1. the timeout could fire when `t==D`;
2. then if `tau` happens when `r==C` say, `good!` will become urgent, however it cannot be performed since `Timeout1` is no longer offering it, causing a timelock. `Component2` will not let time pass until `good` is performed, but `good` cannot be performed because of a mis-matched synchronisation.

We would argue that this is a big problem. In particular, it is not generally possible to ensure that `C` is less than `D` since our component behaviour would typically be embedded in the complex functioning of a complete system. In fact, writing `C` as we have done, abstracts from a likely multitude of complexity and deriving such a value from a system would typically require analysis of many components of the complete system, some of which might be time non-deterministic at the level of abstraction being considered.

Furthermore, in some situations we might actually be interested in analysing what happens if the good action arrives after the timeout has fired. Consider, for example, that our timeout behaviour is being used to wait for an acknowledgement in a sender process. The component performing `good` after `timeout` has fired corresponds to the acknowledgement arriving after the timeout has expired, which is of course a possible scenario in practical analysis of communication protocols.

The problem with our `||<Timeout1,Component2>` solution is that it does not enable us to analysis this situation, rather the system timelocks when `Component2` forces the `good` action to happen. Unfortunately, as mere mortals, we are unable to analyse systems after the end of time!

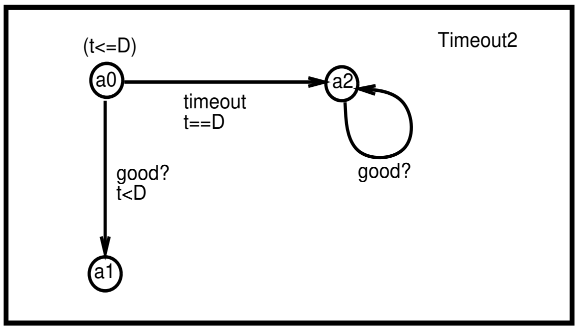

Fig. 5. An UPPAAL Automaton for Timeout2

One way to avoid this timelock is to add "escape" transitions in the timeout. For example, consider the timeout behaviour encapsulated by `Timeout2`. Now the composition,

`||<Timeout2,Component2>`

cannot block time. However, this is not a satisfactory solution since rather than `Timeout2` just evolving to a single deadlock state, `a2`, after performing `timeout`, it could evolve to a complex behaviour; of course in practice it is almost certain to do this. However then, escape transitions would have to be scattered throughout the complex behaviour. This would generate significant specification clutter, which would be compounded if the system contained more than one timeout.

The consequences become particularly severe if the timeout is enclosed in some repetitive behaviour, e.g. see figure 6. This is because, since no assumptions can be made about the time at which the component will want to perform the `good` action, escape transitions on `good` will have to be added at `a0`, `a2`, `b0`, `b1` (and actually `a1` as well). Thus, firstly, the behaviour prior to reaching the timeout has been altered, i.e. escape transitions must be added at `b0` and secondly, it is unclear how many escape transitions need to be added to each node in the loop, since state `a2` may be reached many times before the first `good` escape transition is performed.

Urgent Channels. UPPAAL also contains the concept of an urgent channel. The specifier is allowed to denote a particular channel as urgent, which means

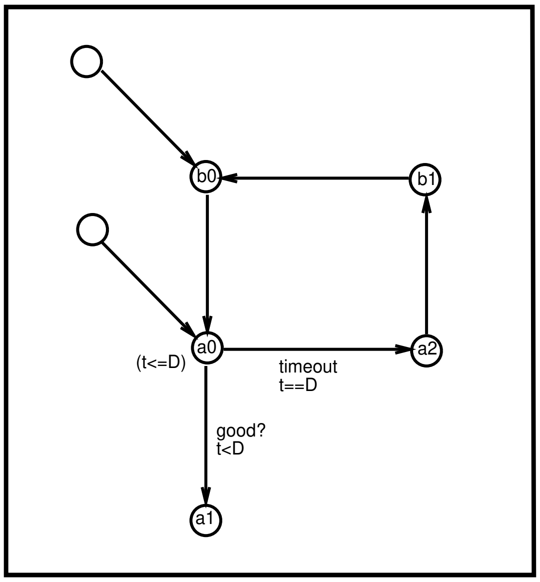

Fig. 6. Timeout2 in a repetitive context

that as soon as synchronisation on that channel can take place, it does. However, UPPAAL restricts the use of such urgent channels. In particular, an urgent transition can only have the guard `true`.

Intuitively, urgent channels seem to be what we require in order to avoid enforcing urgency in the component process. In particular, they enforce urgency in a "global" manner, rather than requiring it to be enforced in the component process. However, it turns out that the restriction on guarding of urgent channels that UPPAAL imposes prevents derivation of a suitable solution, see [5] which investigates possible solutions with urgent channels which were inspired by the solutions presented in [6].

In summary then, although we do not have a formal proof that a completely satisfactory UPPAAL description of the timeout cannot be found, we postulate that if it is possible, the complexity inherent in the solution would be prohibitive.

4 Timed Automata with Deadlines

A more radical approach to realising a satisfactory bounded timeout is to consider the Timed Automata with Deadlines (TAD) framework developed by Bornot et al [3, 4]. The reason for selecting this model is that it is argued that it has very nice properties with regard to time progress and timelocks. In particular, the following property holds, *a state cannot be reached in which neither action or time passing transitions can be performed.*

This property is referred to as *time reactivity* and since such situations (as opposed to zeno timelocks) arise through mismatched parallel compositions, it ensures freedom from composition timelocks.

Basic Framework. For a full introduction to TADs, we refer the interested reader to [3, 4]; here we highlight the main principles:

- *Deadlines on Transitions.* Rather than placing invariants on states, deadlines are associated with transitions. Transitions are annotated with 4-tuples:

$$(\mathtt{a}, \mathtt{g}, \mathtt{d}, \mathtt{r})$$

where `a` is the transition label, e.g. `good`; `g` is the guard, e.g. `t<=D`; `d` is the deadline, e.g. `t==D`; and `r` is the reset set, e.g. `t:=0`. `a`, `g` and `r` are familiar from standard timed automata and the deadline is new. Conceptually, the guard states when a transition is enabled, i.e. *may* be taken; while the deadline states when it *must* be taken and taken immediately.
It is also assumed that the constraint,

$$\mathtt{d} \Longrightarrow \mathtt{g}$$

holds, which ensures that if a transition is forced to happen it is also able to. Clearly, if this constraint did not hold then we could obtain timelocks because a transition is forced to happen, but it is not enabled.
Since we have deadlines on transitions there is no need for invariants on states. Thus, they are not included in the framework.
- *(Timewise) Priorities.* By restricting guards and deadlines in choice contexts, prioritised choice can be expressed. For example, if we have two transitions:

$$\mathtt{b1} = (\mathtt{a1}, \mathtt{g1}, \mathtt{d1}, \mathtt{r1}) \quad \text{and} \quad \mathtt{b2} = (\mathtt{a2}, \mathtt{g2}, \mathtt{d2}, \mathtt{r2})$$

then when placing them in a choice context we can give `b2` priority over

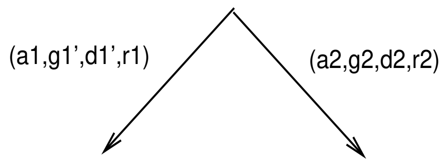

Fig. 7. A Prioritised Choice

`b1` by restricting the guards and deadlines of `b1`, see figure 7. [3] considers a variety of priority operators, which ensure that if the higher priority action will *eventually* be enabled within a particular period of time then it takes precedence over competing actions. These different priority mechanisms are obtained by including timed temporal operators in the restricted guards and deadlines. The extreme example of which is to enforce the following restricted guard and deadline:

$$\mathtt{g1}' = \mathtt{g1} \ \wedge \ \Box\neg\mathtt{g2} \ \text{ and } \ \mathtt{d1}' = \mathtt{d1} \ \wedge \ \mathtt{g1}'$$

which ensures that `b1` is only enabled if `g1` holds and there is *no point in the future* at which `g2` will hold.
- *Parallel Composition with Escape Transitions.* The TADs framework employs a different parallel composition operator to that arising in standard timed automata. The key idea is that of an *escape transition*. These are the local transitions of automata components that are combined when generating a synchronisation transition. Thus, not only are synchronisations included, but component transitions of the synchronisation are as well. The timewise priority mechanism is then used to give the synchronisation transition highest priority. Intuitively, the escape transitions can only happen if the synchronisation transition will *never* be enabled. We will illustrate this aspect of TADs shortly.
- *Synchronisation Strategies.* [3] also consider a number of different synchronisation strategies, but these are not relevant to our discussion. In terms of [3] we only consider *AND* synchronisation.

In fact, in addition to ensuring time reactivity, the TADs framework limits the occurrence of local deadlocks. Specifically, the escape transitions allow the components of a parallel composition to escape a potential local deadlock by evolving locally. Associated with such avoidance of local deadlocks is the enforcement of *maximal progress*[5], which exactly requires that if a synchronisation is possible, it is always taken in preference to a corresponding escape transition.

Basic Definitions. We now briefly review the definition of timed automata with deadlines. Also, in order to preserve some continuity through the paper we continue to use the UPPAAL synchronisation notation even though it is different to that used in [3].

An arbitrary element of $\mathcal{A}$, the set of TADs, has the form:

$$(L, l_0, T)$$

where, L is a finite set of locations; l_0 is the *start location*; and

- $T \subseteq L \times CC \times CC \times A^{+} \times \mathbb{P}(\mathcal{C}) \times L$ is a transition relation. A typical element of which is, $(l_1, \mathtt{g}, \mathtt{d}, \mathtt{a}, \mathtt{r}, l_2)$, where $l_1, l_2 \in L$ are automata locations; $\mathtt{g} \in \mathcal{CC}$

[5] Note, the term is used in a related but somewhat different way in the timed process algebra setting [12].

is a guard; $\mathtt{d} \in \mathcal{CC}$ is a deadline; $\mathtt{a} \in A^+$ labels the transition; and $\mathtt{r} \in \mathbb{P}(\mathcal{C})$ is a reset set. $(l_1, \mathtt{g}, \mathtt{d}, \mathtt{a}, \mathtt{r}, l_2) \in T$ is typically written,

$$l_1 \xrightarrow{\mathtt{g},\mathtt{d},\mathtt{a},\mathtt{r}} l_2$$

In addition, we will use the function:

$$\theta_B(l) = \{\, (b,g) \mid \exists l' \,.\, l \xrightarrow{g,d,b,r} l' \ \wedge \ b \in B \,\}$$

Standard TADs. We will introduce a number of different TADs approaches in this paper. These are distinguished by their rules of parallel composition. Here we consider the basic approach, as introduced in [3, 4], which we call *standard TADs*. A TADs expansion theorem for deriving the product behaviour from a parallel composition is given in [3]. Here we give an equivalent inference rule definition for our state vector notation:-

$$(R1) \quad \frac{l_i \xrightarrow{g_i,d_i,a?,r_i} l_i' \quad l_j \xrightarrow{g_j,d_j,a!,r_j} l_j'}{\begin{array}{l} ||v \xrightarrow{g',d',a,r_i \cup r_j} ||v[l_i'/l_i, l_j'/l_j] \\ ||v \xrightarrow{g_i',d_i',a?,r_i} ||v[l_i'/l_i] \\ ||v \xrightarrow{g_j',d_j',a!,r_j} ||v[l_j'/l_j] \end{array}}$$

where $1 \leq i, j \leq |v| \ \wedge \ i \neq j$ and,

$$\begin{aligned} g' &= g_i \ \wedge \ g_j \\ d' &= g' \ \wedge \ (d_i \ \vee \ d_j) \\ g_i' &= g_i \ \wedge \ \Box\neg(g_i \ \wedge \ g_j) \\ d_i' &= g_i' \ \wedge \ d_i \\ g_j' &= g_j \ \wedge \ \Box\neg(g_i \ \wedge \ g_j) \\ d_j' &= g_j' \ \wedge \ d_j \end{aligned}$$

$$(R2) \quad \frac{l_i \xrightarrow{g,d,a,r} l_i' \qquad a \in \overline{A} \Rightarrow \bigcup_{k \in (\{1..|v|\} - \{i\})} \theta_{\{\overline{a}\}}(l_k) = \emptyset}{||v \xrightarrow{g,d,a,r} ||v[l_i'/l_i]}$$

where $1 \leq i \leq |v|$. (R1) generates synchronisation and escape transitions with the constrained guards and deadlines ensuring that synchronisation has priority in the required manner. (R2) is the interleaving rule, which is straightforward apart from the second condition which ensures that transitions on incomplete actions are only generated by this rule if synchronisation, and hence rule (R1), is not possible.

As an illustration of these inference rules consider `||<A1,A2>` where `A1` and `A2` are shown in figure 8. The unreduced composition arising from directly applying the inference rules is shown in figure 9(a) ($\Box$ is denoted `[]` and $\neg$ is denoted $\sim$) and figure 9(b) depicts the resulting composed TAD when guards and deadlines have been reduced by expanding out temporal operators and applying propositional logic. In addition, transitions with unfulfillable guards, e.g. `false`, have been removed.
We can observe the following:-

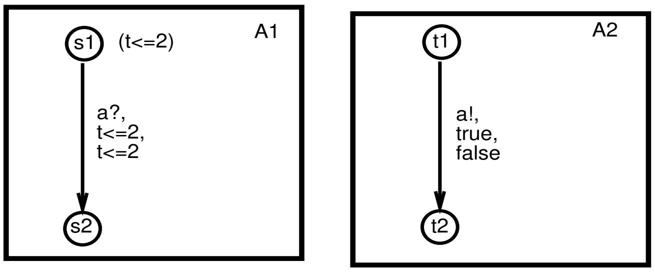

Fig. 8. TADs `A1` and `A2`

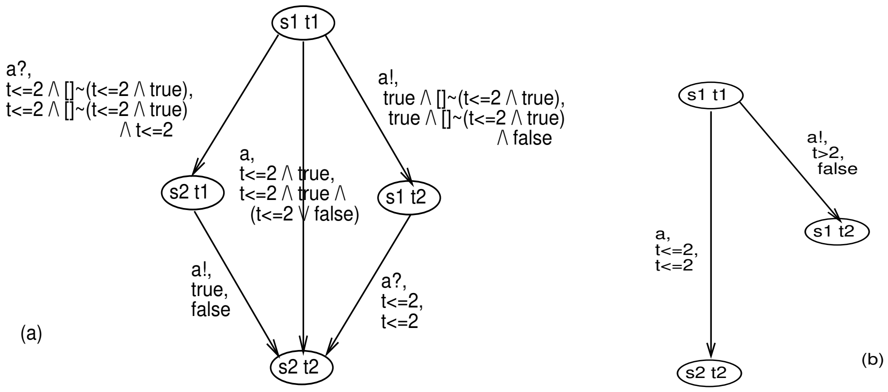

Fig. 9. Unreduced and reduced composition of `A1` and `A2`

1. In figure 9(a) the transition coming from `s1 t1` labelled `a` is the synchronisation transition.
2. In figure 9(a) the two transitions coming from `s1 t1` labelled `a?` and `a!` respectively, are the escape transitions. The first arises from automaton `A1` and the second from automaton `A2`. The guards of these escape transitions ensure that they can only fire if the synchronisation will never in the future be possible. Thus, synchronisation transitions have priority over escape transitions.
3. Figure 9(b) shows that since the synchronisation transition inherits the guards of `a?` from `A1`, no escape transition on `a?` is possible. If `s1 t1` is entered with `t>2` then the escape transition on `a!` can be taken, enabling `A2` to escape its local deadlock.

Bounded Timeout in Standard TADs. Now we reformulate our bounded timeout in standard TADs. The component that we consider is `Component3` and the timeout is `Timeout4` both shown in figure 10.

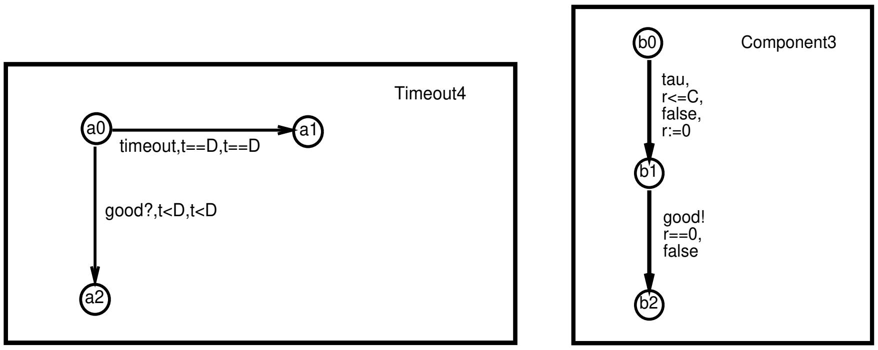

Fig. 10. A TAD for `Timeout4` and `Component3`

So, `Component3` behaves similarly to `Component2` except the `good!` transition is not urgent, i.e. the `good` transition is never forced to happen[6]. In the terminology of [3], such transitions are called *lazy*. In contrast, all the transitions in `Timeout4` are *eager* [3], since their guard and deadline are the same. This implies that as soon as the transition can happen it will happen.

Now by applying the above inference rules and removing impossible transitions, the composite automaton shown in figure 11 results. The full version of this paper [5] presents the intermediate steps required to derive this composition.

If we first focus on state `a0 b1` then we can see that this composite behaviour gives priority to the synchronisation between `good?` and `good!` which is indicated by the transition labelled `good`. Thus, while `t<D` this is the only transition that can fire (notice `r==0` automatically when entering state `a0 b1`) and furthermore it is eager.

Also, if state `a0 b1` is entered with `t==D` then `timeout` is urgent. Furthermore, from this state the action `good!` can happen (but lazily) either at time `D` or later. This is the escape transition, which allows `Component3` to move out of state `b1`. Remember the timelock that we obtained previously arose because the component could not exit the state where it wished to perform `good!`[7].

This solution seems to fulfil our requirements - it is a strong timeout, urgency is enforced as required on both `timeout` and `good` and the solution is timelock free. However, there are some peculiarities with the resulting composite behaviour. Consider for example, the transition from `a0 b0` labelled `good?`. This represents the timeout performing its `good` escape transition. However, con-

[6] We prefer to enforce the urgency of `good` in the timeout because in some of our case studies, e.g. [6], there are situations in which enforcing the urgency of `good` on the system side can cause problems, since nothing ensures that the timeout is ready to synchronise on the `good` exactly when it is offered. In order to avoid this possibility we require the system to passively offer its action and thus, wait until the timeout is ready to receive it.

[7] Actually, the situation is not as severe here since `good!` is lazy.

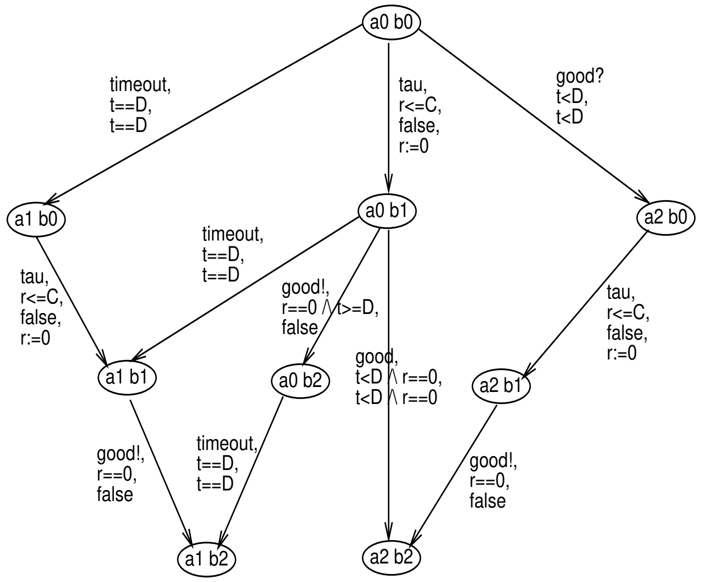

Fig. 11. `||<Timeout4,Component3>` in standard TADs

ceptually it is being performed too early - before the synchronisation on `good` is even offered. In fact, if `a0 b0` is entered with `t==0`, which will almost certainly be the case, then `good?` will almost always be selected since it is an eager transition. In response to this observation we consider alternative TADs formulations in the next section.

5 Alternative TAD Formulations

We consider two alternative TAD formulations[8]. [5] actually considers a third formulation, but this turns out to be unsatisfactory. Both satisfy the requirements that we identified in the introduction for our bounded timeout. Thus, in particular, they are both time reactive. However, the solutions vary in the extent to which they limit local deadlocks.

5.1 Sparse Timed Automata with Deadlines

This is a minimal TADs approach, in which we do not generate *any* escape transitions. Furthermore, since escape transitions are not generated, we do not have to enforce any priority between the synchronisation and escape transitions.

[8] We still call these timed automata with deadlines, because the basic principles, as concieved by Bornot et al [3, 4], still apply, i.e. placing deadlines on transitions and using prioritised choice.

With sparse TADs the following parallel composition rules are used:

$$\frac{l_i \xrightarrow{a?,g_i,d_i,r_i} l'_i \quad l_j \xrightarrow{a!,g_j,d_j,r_j} l'_j}{||v \xrightarrow{a,g',d',r_i \cup r_j} ||v[l'_i/l_i, l'_j/l_j]} \qquad \frac{l_i \xrightarrow{a,g,d,r} l'_i \quad a \in A}{||v \xrightarrow{a,g,d,r} ||v[l'_i/l_i]}$$

where $1 \leq i, j \leq |v|$, $i \neq j$, $g' = g_i \wedge g_j$ and $d' = g' \wedge (d_i \vee d_j)$.

These rules prevent uncompleted actions from arising in the composite behaviour; they only arise in the generation of completed actions, while completed actions offered by components of the parallel composition can be performed independently. This definition has the same spirit as the normal UPPAAL rules of parallel composition. The difference being that here we have deadlines which we constrain during composition to preserve the property $d \Rightarrow g$. It is straightforward to see that as long as this property holds, we will have time-reactivity.

Let us consider once again the behaviour,

```
||<Timeout4,Component3>
```

which is the network we were focussing on in the previous section. Now with our new parallel composition rules, we obtain the composite behaviour shown in figure 12. This is an interesting and very reasonable solution. Firstly, it meets all

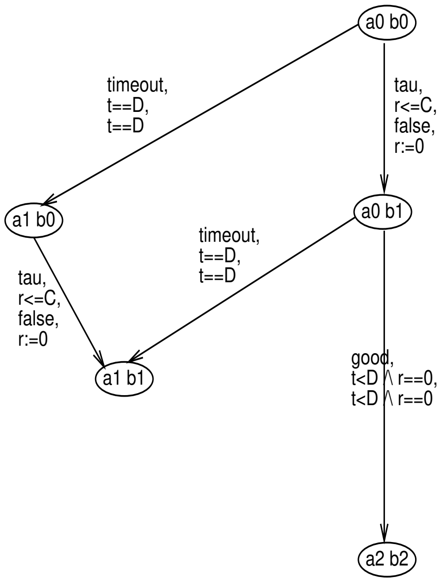

Fig. 12. `||<Timeout4,Component3>` in Sparse TADs

the requirements identified at the start of this paper for our bounded timeout. Thus, in particular, it is time-reactive. However, it makes no effort to limit local deadlocks, so communication mis-matches yield local deadlocks rather than timelocks.

5.2 TADs with Minimal Priority Escape Transitions

The idea here is to ensure maximal progress as standard TADs do, but rather than just giving escape transitions lower priority than their corresponding synchronisation, we also give them lower priority than other completed transitions. Thus, a component can only perform an escape transition if the component will never be able to perform a completed transition.

The parallel composition rules are:

$$(R1) \quad \frac{l_i \xrightarrow{a?,g_i,d_i,r_i} l_i' \quad l_j \xrightarrow{a!,g_j,d_j,r_j} l_j'}{||v \xrightarrow{a,g',d',r_i \cup r_j} ||v[l_i'/l_i, l_j'/l_j]}$$

where, $1 \leq i,j \leq |v|$, $i \neq j$, $g' = g_i \wedge g_j$, $d' = g' \wedge (d_i \vee d_j)$. and,

$$(R2) \quad \frac{l_i \xrightarrow{a,g,d,r} l_i' \quad a \in A}{||v \xrightarrow{a,g,d,r} ||v[l_i'/l_i]} \qquad (R3) \quad \frac{l_i \xrightarrow{a,g,d,r} l_i' \quad a \in \overline{A}}{||v \xrightarrow{a,g'',d'',r} ||v[l_i'/l_i]}$$

where, $1 \leq i \leq |v|$ and,

$$\begin{aligned} g'' = g \wedge & \bigwedge_{(b,\, g') \in \theta_A(l_i)} \Box\neg g' \wedge \\ & \bigwedge_{(b,\, g_1) \in \theta_{\overline{A}}(l_i)} \ \bigwedge_{j \in (\{1..|v|\} - \{i\})} \ \bigwedge_{(\overline{b},\, g_2) \in \theta_{\{\overline{b}\}}(l_j)} \Box\neg(g_1 \wedge g_2) \\ d'' = d \wedge & \ g'' \end{aligned}$$

R1 is the normal synchronisation rule; R2 defines interleaving of completed transitions; and R3 defines interleaving of incomplete, i.e. escape, transitions. In this final rule, g'' holds when,

1. g holds;
2. it is not the case that an already completed transition from l_i could eventually become enabled; and
3. it is not the case that an incomplete transition (including `a` itself) offered at state l_i could eventually be completed.

Applying these rules to the composition:

```
||<Timeout4,Component3>
```

and removing impossible transitions, see [5] for a full presentation, yields the composition shown in figure 13. This solution removes the excessively early escape transition from `a0 b0`, but preserves all other transitions.

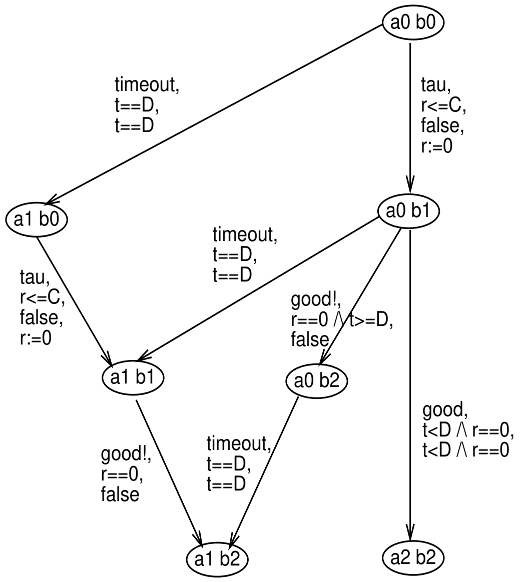

Fig. 13. `||<Timeout4,Component3>` in TADs with minimum priority escape transitions

6 Discussion and Conclusions

We failed to find a fully satisfactory UPPAAL specification of a bounded timeout. In response, we presented three different TADs solutions - standard TADs, sparse TADs, and TADs with minimal priority escape transitions. The latter two of which are new to this paper[9]. All three meet all the requirements we identified at the start of the paper for our timeout. However, our preference is for the 2nd and 3rd solutions. The 2nd is interesting because it gives a timelock free solution but does not seek to minimise local deadlocks, while the 3rd adds escape transitions to limit such local deadlocks.

It is interesting to consider a specific timeout example. In the same way as earlier in the paper, we consider the implications if we view the `good` action as the passing of an acknowledgment from the medium to a waiting sender process. The situation that the component wishes to perform `good` after the timeout has fired corresponds to the medium delivering the acknowledgment too late. The two solutions handle this situation differently.

With the sparse TADs solution, a local deadlock is generated. Conceptually, this indicates that the medium is prevented from delivering the acknowledgement. This is in fact the normal manner in which mis-matched communications are handled in untimed systems - local deadlocks result. In contrast, with TADs

[9] In particular, the *stiff* parallel composition of [13] which seem related to our sparse TADS are in fact rather different since they do not ensure that deadlines imply guards when generating the product. Thus, they do not ensure time reactivity.

with minimal priority escape transitions the late delivery of the acknowledgement yields a local transition in the medium, which could be viewed as the acknowledgement packet being consumed/dropped. This avoids the local deadlock and allows the system to proceed. Choosing between these two should be made according to the application domain under consideration.

Acknowledgements. The author has benefited greatly from discussions with Sebastian Bornot, Joseph Sifakis and Stavros Tripakis and would also like to recognise the contribution of Giorgio Faconti, Joost-Pieter Katoen, Diego Latella and Meike Massink who were involved in preliminary discussions from which this paper has grown. In addition, the reviewers made a number of valuable recommendations.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, pages 183–235, 1994.
2. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, and Paul Pettersson amd Wang Yi. Uppaal - a tool suite for automatic verification of real-time system. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.
3. S. Bornot and J. Sifakis. On the composition of hybrid systems. In *Hybrid Systems: Computation and Control*, LNCS 1386, pages 49–63, 1998.
4. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality, COMPOS'97*, LNCS (to appear), 1997.
5. H. Bowman. Discussion document - modelling timeout behaviour in timed automata. Technical report, Available from author, 1998.
6. H. Bowman, G. Faconti, J-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation algorithm using UPPAAL. In *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, 1998. To Appear in Special Issue of Formal Aspects of Computing.
7. C.Daws, A.Olivero, S.Tripakis, and S.Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, LNCS 1066. Springer-Verlag, 1996.
8. Th. A. Henzinger and Pei-Hsin. HyTech: The Cornell HYbrid TECHnology tool. In *Proceedings of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995.
9. L. Leonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sens itive systems. *Computer Networks and ISDN Systems*, 29:271–292, 1996.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
11. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebra. In *Real-time Theory in Practice*, LNCS 600, pages 549–572. Springer-Verlag, June 1991.
12. T. Regan. Multimedia in temporal LOTOS: A lip synchronisation algorithm. In *PSTV XIII, 13th Protocol Specification, Testing and Verification*. North-Holland, 1993.
13. J. Sifakis and S. Yiovine. Compositional specification of timed systems, (extended abstract). In *STACS'96, Proceedings of the 13th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 1046, pages 347–359. Springer-Verlag, 1996.