Validation, Verification and Implementation of Timed Protocols using AORTA

Steven Bradley William Henderson David Kendall Adrian Robson *

February 15, 1995

Abstract

AORTA is an implementable timed process algebra which has been proposed as a design language for hard real-time systems. In this paper we show how AORTA can be used to design and model timed protocols, illustrated by the alternating bit protocol. We also describe tools which have been developed for simulation, verification and automatic implementation of AORTA systems, and outline a relationship between the formal models which are verified and the code which is generated.

1 Introduction

Communication protocols and embedded hard real-time systems share many important features — timing, concurrency and communication are central issues to the designers of both. Because of this close relationship, many embedded system design techniques can be applied to communication protocols and vice-versa, and in this paper we show how AORTA [4] can be applied to timed communication protocols. The main distinction between AORTA and other timed process algebras such as ET-LOTOS [14], TCSP [22] and TCCS [17] is that AORTA is aimed specifically at design, whereas other algebras are meant as wide-spectrum languages, useful for specification, design, and modelling. Because of its focus, AORTA is in some ways more restrictive and in some ways more expressive than these languages, to guarantee than designs can be implemented directly. Rather than using timed bisimulations or preorders, which relate sytems at different levels of abstraction in the same language, we prefer to use timed model-checking [1], based on temporal logic, as this makes it much easier to construct abstract specifications. To demonstrate the practicality of AORTA, a set of tools has been developed, which allow systems to be simulated, verified and automatically implemented.

The structure of the paper is as follows. After introducing the syntax of AORTA in section 2, a description of the alternating bit protocol in AORTA is given in section 3; this example is used throughout the paper to illustrate the techniques described. Validation and testing of AORTA systems by simulation is described in section 4, and verification by model-checking is explained in section 5, including a translation from AORTA to timed graphs. These timed graphs are used not only in verification, but also in implementation, providing assurance that verification theorems apply directly to the implementation. We also show in sections 4 and 5 how AORTA can be used to model different assumptions about the behaviour of a communication line, ranging from bounds on the time taken to transmit, to the loss of messages. The use of code generation techniques, and their relationship to timed graphs, are then outlined in section 6, before the concluding section 7.

*The authors are with the Department of Computing, University of Northumbria at Newcastle, Ellison Place, Newcastle upon Tyne, NE1 8ST, UK

2 AORTA syntax

AORTA is a timed process algebra, and draws on untimed process algebras, in particular CCS [16], for its syntax. Each AORTA system is statically defined as a parallel composition of sequential processes, which may intercommunicate. Each individual process may wait for communication (all communication is synchronised and so blocks progress), perform computation, branch between different behaviours, recurse, or do nothing. The simplest process which can be defined is

```
A = a.A
```

which waits for communication on gate `a`, before behaving like process `A`. In other words the process is always ready to offer `a` actions. A simple buffer process can be defined by adding computation and recursion, so that

```
A=a.[5.0,15.0]b.A
```

describes a process which communicates on gate `a` (possibly accepting some data), performs some computation, which takes between 5.0 and 15.0 time units to complete, before communicating on gate `b` (possibly offering some data) and returning to the start again. There are two important points to note here, firstly that data is not handled explicitly in AORTA (although there are extensions which do [6]), so computation is represented only by the amount of time it takes, and secondly that computation delays can be represented as *bounds* on execution times rather than exact figures. The use of bounds for computation delays (as well as for communication delays and time-outs) make it much easier to provide verifiable implementation techniques. Implementation, however, will be considered in more detail in section 6.

Behaviour branching which is dependent on communication is represented by the `+` operator. Two or more communications are offered at once, and the subsequent behaviour depends on which is taken up first. This operator can be used to implement a channel which accepts and delivers two kinds of messages:

```
Channel = in1.out1.Channel
          +
          in2.out2.Channel
```

Here once another process sends a message via `in1`, the channel is not available until the message is received at the far end (gate `out1`). As well as using choice, communication can be extended with time-out, which allows another behaviour branch to be followed if no communication occurs within a certain time. This could be used to force a limit on how long the channel would wait for a message to be accepted:

```
Channel = in1.(out1.Channel)[5.0,5.1>Channel
          +
          in2.(out2.Channel)[5.0,5.1>Channel
```

Notice the use of brackets to indicate which communication the time-out is to affect, and the use of time bounds for specifying the time-out value.

There are other forms of branching behaviour which do not depend on communication, such as data dependent branching and faulty behaviour. These are both represented by the non-deterministic choice operator `++` in AORTA, which we shall describe in a little more detail in section 4. Table 1 summarises the syntax of sequential process syntax.

Having defined the processes that make up a system, they must be placed in parallel and have their gates connected for either internal or external communication. Communication delays (again, expressed with bounds) are also given at the system level, to represent the amount of time taken for the system to

communication	`a.S`
communication choice	`a1.S1 + ... + an.Sn`
bounded delay	`[t1,t2]S`
bounded time-out	`(a1.S1 + ... + an.Sn)[t1,t2>S`
non-deterministic choice	`S1 ++ ... ++ Sn`
recursion	equational definition

Table 1: Summary of AORTA sequential process syntax

notice and effect any communication. All of this information is given in the *connection set*, which lists pairs of gates for internal connection (each gate may only be connected once) and externally connected gates, and the corresponding delay bounds. These connection sets can be realised graphically, and together with the processes correspond to Milner's flow graphs [16]. An example of a connection set is given in the next section, where we describe the alternating bit protocol in AORTA.

3 Alternating Bit Protocol in AORTA

The alternating bit protocol is a widely discussed example, probably because it is one of the simplest examples of a protocol which can do something 'useful' — it guarantees integrity of communication in a situation where messages may be duplicated or lost. Our description is based on Milner's CCS description in [16]. We are concerned with constructing `Send` and `Reply` processes which can be connected by possibly noisy channels `Trans` and `Ack`. The way the alternating bit protocol works is that messages and acknowledgements are tagged with a bit (0 or 1), with successive messages and acknowledgements being tagged with alternating bits. Having sent a 0-tagged message, the sender waits for a 0-tagged acknowledgement; if one does not arrive within a certain amount of time, the message is sent again. Once an acknowledgement does arrive, the next message can be sent, this time tagged with a 1. The replier, meanwhile, waits for a 0-tagged message, delivers the data, and sends a 0-tagged acknowledgement. If it receives another 0-tagged message it simply sends another 0-tagged acknowledgement, but a 1-tagged message causes it to deliver the message and return a 1-tagged acknowledgement and so on. The layout of the system is shown in figure 1.

The `Reply` process is the simpler of the two, and is defined in AORTA as follows

```
Reply = trans0.Deliver0
Deliver0 = deliver.Reply0
Reply0 = reply0.(trans1.Deliver1+trans0.Reply0)
Deliver1 = deliver.Reply1
Reply1 = reply1.(trans0.Deliver0+trans1.Reply1)
```

A time-out is added to the `Send` process to resend messages if acknowledgements are not sent within a certain amount of time.

```
Send = accept.Send0
Send0 = send0.Sending0
Sending0 = (ack0.Accept1 + ack1.Sending0)[100.0,101.0>Send0
Accept1 = accept.Send1
Send1 = send1.Sending1
Sending1 = (ack1.Accept0 + ack0.Sending1)[100.0,101.0>Send1
Accept0 = accept.Send0
```

The channel processes `Trans` and `Ack` can also be modelled in AORTA. A simple model of these

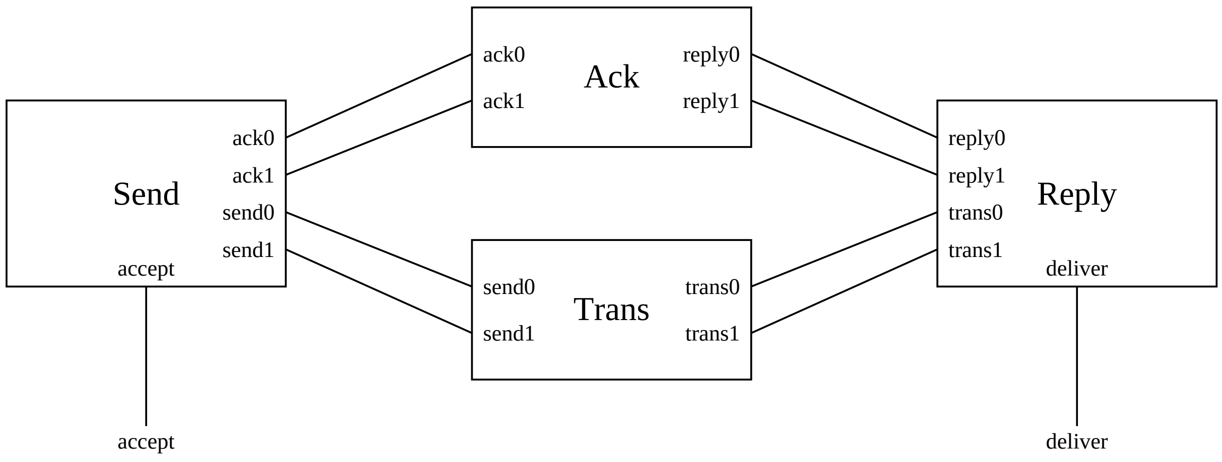


Figure 1: Alternating Bit Protocol System

processes has them accepting transmissions or replies, and passing them on after a certain delay, but without loss or replication. In AORTA, this is written

```
Ack = reply1.([25.0,75.0]ack1.Ack)
      +
      reply0.([25.0,75.0]ack0.Ack)

Trans = send0.([25.0,75.0]trans0.Trans)
        +
        send1.([25.0,75.0]trans1.Trans)
```

Notice here that if the minimum delay of 25 time units is taken by both buffers then the acknowledgement will reach the `Send` process before the 100 unit time-out expires, but if the maximum delay of 75 is incurred, then the sending process will retransmit its data. These processes are connected up with the `Send` and `Reply` processes, by defining the parallel composition and connection set

```
( Send | Reply | Ack | Trans )
<(Send.send0,Trans.send0 : 0.5,1.0),
 (Send.send1,Trans.send1 : 0.5,1.0),
 (Send.ack0,Ack.ack0 : 0.5,1.0),
 (Send.ack1,Ack.ack1 : 0.5,1.0),
 (Reply.reply0,Ack.reply0 : 0.5,1.0),
 (Reply.reply1,Ack.reply1 : 0.5,1.0),
 (Reply.trans0,Trans.trans0 : 0.5,1.0),
 (Reply.trans1,Trans.trans1 : 0.5,1.0),
 (Send.accept,EXTERNAL : 0.5,1.0),
 (Reply.deliver,EXTERNAL : 0.5,1.0)>
```

Connections are listed inside angle brackets, and each connection has an associated communication delay. The graphical representation of this system is presented in figure 1, and this same diagram is used within the AORTA simulator to indicate which internal and external communications are available. In the next section we discuss the use of the simulator, and its relation to the formal semantics of AORTA.

4 Simulation and Validation

AORTA has a formal semantics, given in terms of a timed transition system defined by operational rules [4]. If an AORTA expression S_1 becomes S_2 after t units of time, this is written

$$S_1 \xrightarrow{(t)} S_2$$

and if a communication a takes place, this is written

$$S_1 \xrightarrow{a} S_2$$

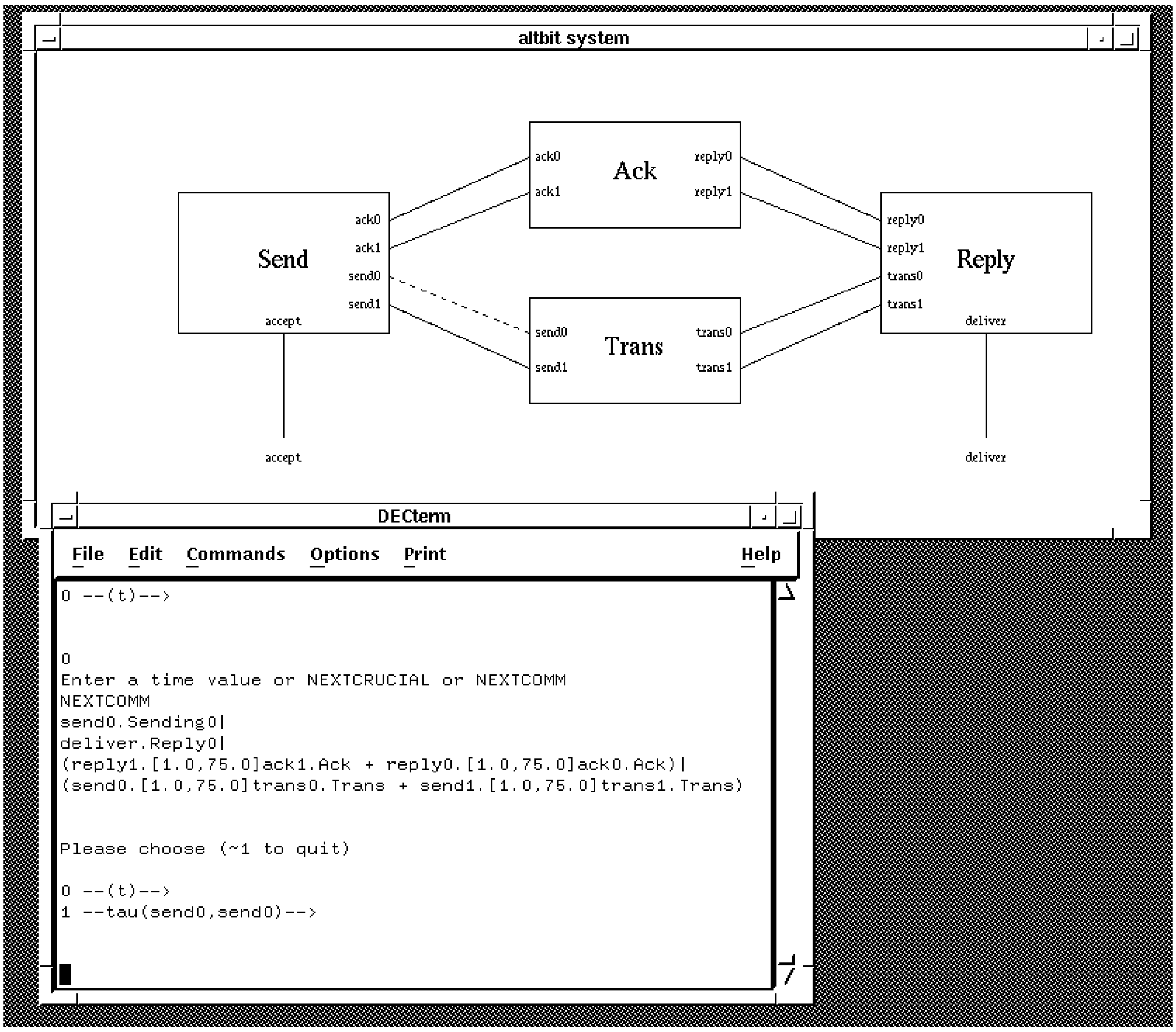
where a can be a gate name (for external communication) or the distinguished action τ (for internal communication). This behaviour can be observed by using the AORTA simulator, which takes an AORTA description of a system (such as the one described in section 3) and allows the behaviour to be stepped through using a simple menu-driven system. This much is similar to other simulators, such as can be found in the concurrency workbench [10] or other tools, but there is also a facility for showing graphically which communications are available. Figure 2 shows a screen shot of the simulator with an internal action available — the corresponding connection is shown as a dashed line.

After each transition (time or action), a menu of possible further transitions is offered, with any possible action transitions displayed on the system layout diagram. If an action transition is chosen then the corresponding connection is flashed on the diagram. If a time transition is chosen then a further prompt asks for a time value, or one of the commands `NEXTCRUCIAL` and `NEXTCOMM`. The two commands progress time up to the next crucial point (the end of a delay or time-out) and the next possible internal communication respectively. If a time value is given the system will be aged by that amount, provided it does not go through a possible internal communication. All internal communications must take place as soon as they become available (the maximum progress principle, enforced in the semantics), and the communication delay follows the occurrence of the τ action in the semantics.

The non-determinism expressed in time bounds and non-deterministic choice, has to be resolved by the simulator somehow. There is a variety of tactics available, which are prompted for when the simulator is started up. Resolution of time bounds can be done by always choosing the minimum value, always choosing the maximum value, choosing a random value, or always prompting the user for a value. Resolution of non-deterministic choice is always achieved by prompting the user.

Using the simulator, a system can be tested (although still using the formal semantics) before an attempt is made to formally verify it via model-checking (see section 5), or to implement it via coded generation (see section 6). This approach can save a lot of time and frustration spent trying to verify properties that are not true.

The alternating bit protocol system can helpfully be exercised with this simulator. By choosing time bounds to be resolved to the minimum value, the protocol is never required to retransmit data, so the normal behaviour can be examined. If maximum values for times are chosen the sending process always has to time-out, so that part of the behaviour is exercised. Different assumptions about buffer behaviour can be built into the system by altering the `Trans` and `Ack` processes and simulating the behaviour of the new system. For example, to include the possibility of the transmit buffer losing one of the messages we can use non-deterministic choice to represent failure. Non-deterministic choice is written with the `++` operator, which chooses between the branches non-deterministically, so we can have one branch as normal behaviour and another as faulty behaviour. A version of the `Trans` process which allows for the possibility of messages being lost, but which has no delays involved except internal communication delays, is given by



```
0 --(t)-->


0
Enter a time value or NEXTCRUCIAL or NEXTCOMM
NEXTCOMM
send0.Sending0|
deliver.Reply0|
(reply1.[1.0,75.0]ack1.Ack + reply0.[1.0,75.0]ack0.Ack)|
(send0.[1.0,75.0]trans0.Trans + send1.[1.0,75.0]trans1.Trans)


Please choose (~1 to quit)

0 --(t)-->
1 --tau(send0,send0)-->
```

Figure 2: Screen Dump of the Simulator

```
Trans = send0.((trans0.Trans) ++ Trans)
        +
        send1.((trans1.Trans) ++ Trans)
```

but this process allows arbitrarily many messages to be lost. This is the general case for such a channel, but no bounded response theorems can be proved of a system which may have to repeat a message arbitrarily many times. To describe a channel which may lose at most one copy of each message, this must be changed to

```
Trans = send0.((trans0.Trans) ++ (send0.trans0.Trans))
        +
        send1.((trans1.Trans) ++ (send1.trans1.Trans))
```

Such use of the ++ operator to represent failures is a powerful tool in modelling assumptions that can be made about a system, and can be used to evaluate to what extent a system may be perturbed before losing functionality. When this second process is used in the system, the simulator asks the user to choose between the two cases, corresponding to whether a message is to be transmitted at the first attempt, or whether a retransimission will be necessary. In either case, the simulator validates that the protocol successfully delivers the message and acknowledges the transmission. The formal verification of this fact can be automatically achieved by model-checking, as described in the next section. The duplication of messages can be handled in a very similar way.

5 Timed Graphs and Verification

Approaches to the automatic verification of finite-state concurrent systems have been known for more than a decade [12, 9]. Such techniques are based upon checking that the state graph of a concurrent system is a model for the temporal logic formulae which are used to specify desired system properties. Such an approach is of great practical interest because it allows the developer to verify a system without constructing a proof and because, when the verification fails, it is possible to provide automatically a trace of the unsatisfactory behaviour; this can be very useful in debugging. Recent work has shown how systems with a large number of states can be checked by using a symbolic representation of the state graph [8, 15] and how this approach can be adapted to the verification of real-time systems [2, 13].

5.1 Timed Graphs

Timed graphs [1] have been shown to be appropriate models for real-time systems and have been adopted in the construction of model-checking tools [18, 23] Our present approach to verification depends upon translating AORTA expressions to timed graphs in order to make use of such tools. In this section, we describe in detail the basis of this translation which follows closely that of [18] but differs in a number of interesting respects. The syntactic restrictions on AORTA allow a simpler translation and lead to graphs which inevitably possess a number of desirable properties including bounded variability (only a bounded number of transitions are possible in a finite time) and non-zenoness (time is always able to progress eventually). A translator has been implemented in Standard ML and incorporated into the AORTA tool set. We adapt the variant of timed graphs described in [18] and present the relevant definitions here for completeness.

A timed graph is an automaton which is extended with a finite set of *clocks* where a clock is a real-valued variable which records elapsed time. Clocks advance uniformly with time or can be reset to zero. We assume throughout that the time domain is the non-negative reals although our results hold for other domains such as the natural or rational numbers.

For a finite set of clocks C and rationals $\mathbf{Q}$, the set of *clock formulae* $\mathcal{F}(C)$ is

$$\mathcal{F}(C) = \{c \geq r | c \in C, r \in \mathbf{Q}\}$$

A clock *valuation* $v \in \mathbf{R}^C$ is a function which assigns to each clock $c \in C$ a value $v(c) \in \mathbf{R}$. We write $v + t$ for the valuation v' such that $v'(c) = v(c) + t$ for all $c \in C$, and for $C' \subseteq C$ we write $v[C' := 0]$ for the valuation v' such that $v'(c) = 0$ for $c \in C'$ and $v'(c) = v(c)$ otherwise. The evaluation of clock formula f given clock valuation v is written $f(v)$ and we say v satisfies a clock formula $c \geq r$ if $v(c) \geq r$.

Definition 5.1 *A timed graph is a tuple,* $(N, n^0, C, E, \mathbf{tcp})$*, where*

- N *is a finite set of nodes*
- n^0 *is the initial node*
- C *is a finite set of clocks*
- $E \subseteq N \times \mathit{Label} \times \mathcal{F}(C) \times 2^C \times N$ *is a finite set of edges representing transitions. Each transition* $(n, l, f, C', n') \in E$ *consists of a source location* n *and a target location* $n' \in N$*, a label* l*, a clock formula* f *and a set of clocks* $C' \subseteq C$.

- $\mathbf{tcp} : N \rightarrow \mathbf{R}^C \rightarrow \mathbf{R} \rightarrow \mathbf{Bool}$ *is a predicate which determines for each location n, clock valuation v and time value t whether the system can remain at location n while time is allowed to progress by an amount t.*

A timed graph gives rise to a labelled timed transition system, $(S, s^0, \longrightarrow)$ where

- $S = N \times \mathbf{R}^C$ is the set of states
- $s^0 = (n^0, v[C := 0])$ is the initial state, and
- the transition relation $\longrightarrow$ is given by the rules

$$\boxed{\textbf{Action}}\ \frac{(n, a, f, C', n') \in E \wedge f(v)}{(n, v) \xrightarrow{a} (n', v[C' := 0])} \qquad \boxed{\textbf{Time}}\ \frac{\mathbf{tcp}(n)(v)(t)}{(n, v) \xrightarrow{(t)} (n, v + t)}$$

5.2 Translation Method

We first give an abstract syntax for AORTA expressions. For a finite indexing set I, $i, j \in I$, a finite set of gate names *Act*, $a_i \in$ *Act*, and a set of process names *Proc*, $X \in$ *Proc*, the set of sequential expressions *Seq* with $S, S_i \in$ *Seq*, is given by

$$S ::= \sum_{i \in I} a_i.S_i | S_1 \rhd_{t_1}^{t_2} S_2 | \bigoplus_{i \in I} S_i | X$$

which correspond to summation, time-out, non-deterministic choice and recursion, respectively. As usual, we write a sum over an empty indexing set as $\mathbf{0}$, the process which can not perform any action. Computation delay and deterministic versions of the timed operators then have natural abbreviations as follows:

$[t_1, t_2]S$	$\stackrel{\text{def}}{=}$	$\mathbf{0} \rhd_{t_1}^{t_2} S$	Non-deterministic computation delay
$[t]S$	$\stackrel{\text{def}}{=}$	$\mathbf{0} \rhd^{t} S$	Deterministic computation delay
$S_1 \rhd^t S_2$	$\stackrel{\text{def}}{=}$	$S_1 \rhd_t^t S_2$	Deterministic time-out

Although computation delays have equivalent formulations as time-outs at this level of abstraction, the notation for them is introduced not simply for convenience but because they require a different treatment in implementation as will become apparent in section 6.

The set of system expressions *Sys*, $\psi \in$ *Sys*, being the parallel composition of a finite number of sequential expressions, is given by

$$\psi ::= \prod_{i \in I} S_i < K >$$

where K is a finite set of internal connections, each connection being represented by an unordered pair of gate names; we require that each gate is connected either to exactly one other gate or to its environment via a single external connection and assume the latter in the case of any gate name occurring in a system expression but not in its associated connection set.

We first give the translation for sequential process expressions. The translation depends on the fact that every sequential process can be implemented using a single clock (we write c_s for the clock associated with process S and abbreviate the singleton $\{c_s\}$ to c_s when the context is clear). The clock associated with a sequential process is reset on every transition and so simply records the time since the process last made a transition. For a sequential process S and its associated clock c_s, we define a compositional translation to a timed graph based on the structure of S.

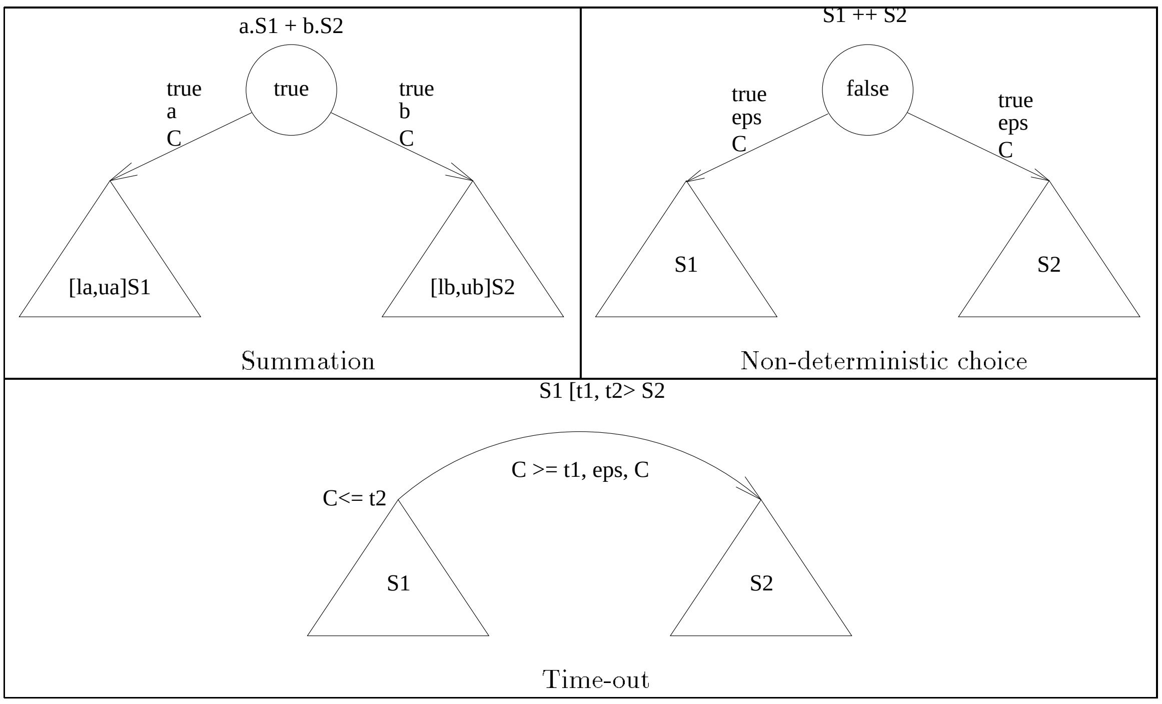


Figure 3: Simple timed graph constructions

In order to do this we introduce a new distinguished action $\epsilon \notin Act$, which is used to label time-out transitions and transitions arising from the resolution of non-deterministic choice. The set of labels is then $Label = Act \cup \{\tau, \epsilon\}$.

The translation is given by a function $\mathcal{G}[\![\_]\!]$ which takes an AORTA expression to its equivalent timed graph. Figure 3 shows some simple examples and figure 4 shows the graph constructed for the `Send` process in the alternating bit protocol. The latter exhibits a pleasing symmetry which reflects that found in the process description.

Summation In general, a sum has the form $\sum_{i\in I} a_i.S_i$. The translation for this expression covers the cases of the **0** process and also action prefixing, in addition to deterministic choice.

For a gate a, let the lower and upper bounds of the possible communication delay of a be written l_a and u_a, respectively. Then, if for $i \in I$ the graphs of $[l_{a_i}, u_{a_i}]S_i$ are $(N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$, then

$$\mathcal{G}[\![\sum_{i\in I} a_i.S_i]\!] = (N \cup \{n^0\}, n^0, c_s, E, \mathbf{tcp})$$

where $N = \bigcup_{i\in I} N_i, n^0 \notin N$,

$$E = \bigcup_{i\in I} E_i \cup \{(n^0, a_i, \mathtt{true}, c_s, n_i^0) | i \in I\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ and $\mathbf{tcp}(n^0)(v)(t) = \mathtt{true}$ for any clock valuation v and time value t.

Time-out Let $\mathcal{G}[\![S_i]\!] = (N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$ for $i \in \{1, 2\}$. Then

$$\mathcal{G}[\![S_1 \rhd_{t_1}^{t_2} S_2]\!] = (N_1 \cup N_2, n_1^0, c_s, E, \mathbf{tcp})$$

where

$$E = E_1 \cup E_2 \cup \{(n_1^0, \epsilon, c_s \geq t_1, c_s, n_2^0)\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ except that $\mathbf{tcp}(n_1^0)(v)(t)$ is $\mathbf{tcp}_1(n_1^0)(v)(t) \wedge v(c_s) + t \leq t_2$.

The case of non-deterministic time-out presented here subsumes deterministic time-out and computation delay in an obvious way.

Non-deterministic choice For $i \in I$ let $\mathcal{G}[\![S_i]\!] = (N_i, n_i^0, c_s, E_i, \mathbf{tcp}_i)$. Then

$$\mathcal{G}[\![\bigoplus_{i \in I} S_i]\!] = (N \cup \{n^0\}, n^0, c_s, E, \mathbf{tcp})$$

where $N = \bigcup_{i \in I} N_i, n^0 \notin N$,

$$E = \bigcup_{i \in I} E_i \cup \{(n^0, \epsilon, \mathtt{true}, c_s, n_i^0) | i \in I\}$$

and $\mathbf{tcp}(n_i) = \mathbf{tcp}_i(n_i)$ for all locations $n_i \in N_i$ and $\mathbf{tcp}(n^0)(v)(t) = \mathtt{false}$ for any clock valuation v and time value t. In other words the choice must be resolved before time can progress.

Recursion The syntactic restrictions on the use of recursion allow its translation to proceed in a very straightforward manner. When a process name X is encountered in the translation of a sequential expression, its translation is simply the graph associated with X; such an association will exist if X has been encountered before but not otherwise. In the latter case, we associate $\mathcal{G}[\![\mathbf{0}]\!]$ with X and add X to a list of names whose graphs are yet to be constructed. Following the first pass of our translation, we construct the graph for each name in this list, by translating the right-hand side of the defining equation for the name. The initial node of each graph constructed in this way is identified with the initial node of the graph previously associated with the name. We continue in this way until we have constructed the graphs for all names encountered.

Parallel composition In giving the translation for parallel composition we adopt the the following notational abbreviations:

$\vec{N}$	for $N_1 \times N_2 \times \ldots \times N_{\|I\|}$
$\vec{n}$	for $(n_1, n_2, \ldots, n_{\|I\|})$
$\vec{n}_{ij}$	for $(n_1, n_2, \ldots n_i, \ldots n_j, \ldots n_{\|I\|})$
$\vec{n}_{i'j'}$	for $(n_1, n_2, \ldots n_i', \ldots n_j', \ldots n_{\|I\|})$

where we assume some indexing set I, $\{i, j\} \subseteq I, i \neq j$.

The translation for an AORTA system expression is given by

$$\mathcal{G}[\![\prod_{i \in I} S_i < K >]\!] = (\vec{N}, \vec{n^0}, \{c_{S_i} | i \in I\}, E, \mathbf{tcp})$$

The set of transitions is $E = IC \cup EC \cup TO$, where

$$\begin{aligned} IC &= \{(\vec{n}_{ij}, \tau, \mathtt{true}, \{c_{S_i}, c_{S_j}\}, \vec{n}_{i'j'}) | \\ &\quad (n_i, a, \mathtt{true}, c_{S_i}, n_i') \in E_i, (n_j, b, \mathtt{true}, c_{S_j}, n_j') \in E_j, (a, b) \in K\} \quad (1) \\ EC &= \{(\vec{n}_i, a, \mathtt{true}, \{c_{S_i}\}, \vec{n}_{i'}) | \\ &\quad (n_i, a, \mathtt{true}, c_{S_i}, n_i') \in E_i, (a, \_) \notin K, (\vec{n}_i, \_, \_, \_, \_) \notin IC\} \quad (2) \\ TO &= \{(\vec{n}_i, \epsilon, \phi, \{c_{S_i}\}, \vec{n}_{i'}) | (n_i, a, \phi, c_{S_i}, n_i') \in E_i\} \quad (3) \end{aligned}$$

For any location $\vec{n} \in \vec{N}$, clock valuation v and time value t, $\mathbf{tcp}(\vec{n})(v)(t)$ is $\bigwedge_{i \in I} \mathbf{tcp}_i(n_i)(v)(t)$ except that for any location $\vec{n}$ such that $(\vec{n}, \_, \_, \_, \_) \in IC$ we require that $\mathbf{tcp}(\vec{n})(v)(t)$ is false; in other words,

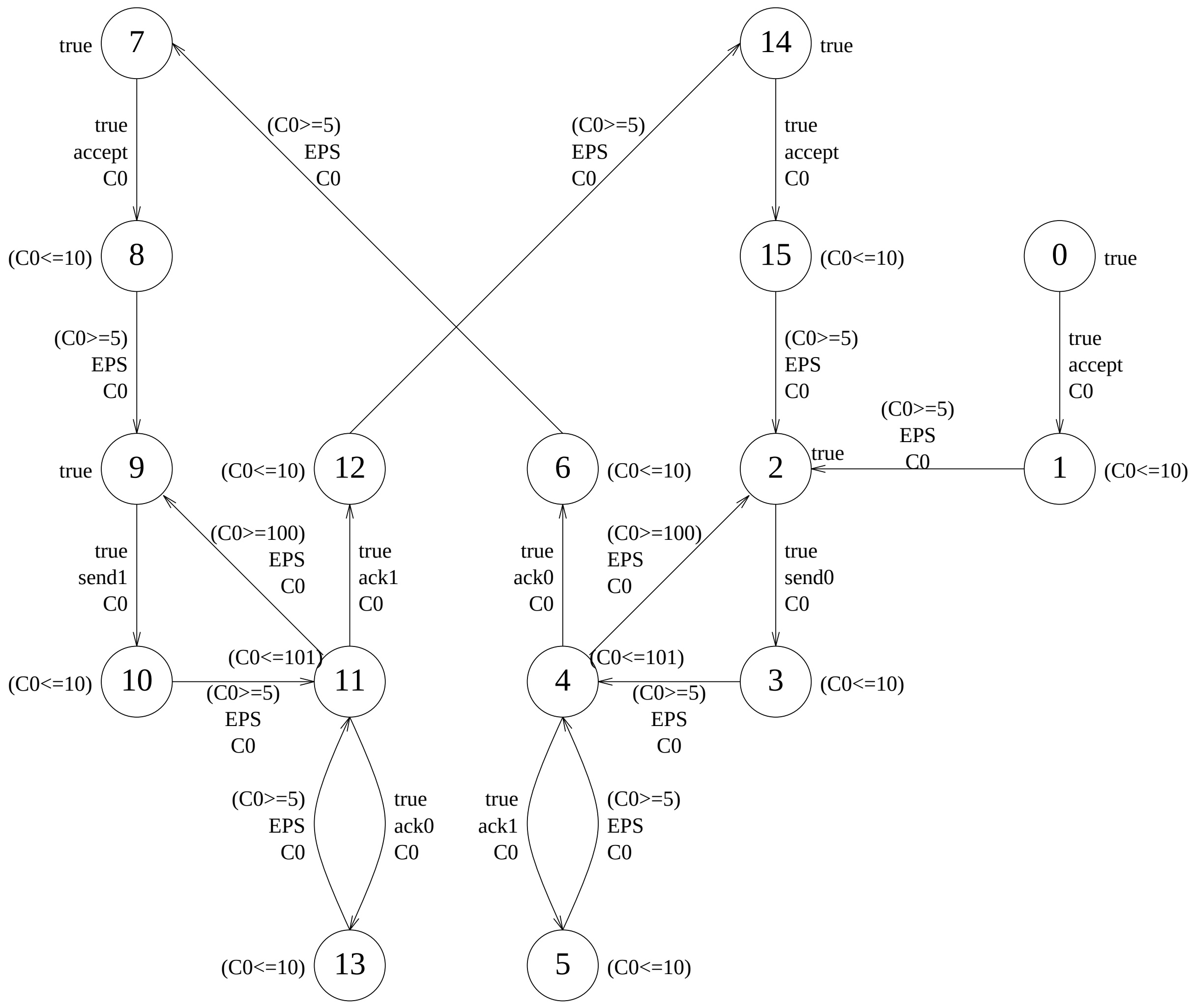


Figure 4: The timed graph of the `Send` process

time can progress only when all processes allow it but even then, not at any location which has the capacity for internal communication, so enforcing the maximal progress principle.

The sets (1) – (3) give the transitions for internal communication, external communication and time-outs / nondeterministic choice, respectively. Notice that an external communication is allowed only in a state where no internal communication is possible. This simple priority mechanism ensures the desirable implementation property that a component cannot become swamped by communication with its environment.

The labelled transition system induced by the timed graph constructed for an AORTA expression is equivalent to that given directly by the AORTA semantics [4]. We omit the proof and the semantics for reasons of space.

We observe that the graphs constructed by this approach have a rather simple structure in that the only edges which are constrained by a clock condition are those arising from time-outs and computation delays and then only by a simple constraint on a single clock. We intend to explore whether this simplicity of structure can be exploited in the construction of a more efficient model checker.

5.3 Using KRONOS to verify timing requirements

KRONOS [18, 23, 19] is a symbolic model checker which implements the approach described by Henzinger et al. [13]. It allows timed graphs to be checked for properties expressed in the real-time logic TCTL [1].

For a finite set of atomic propositions P, the formulae of TCTL are defined as follows:

$$\phi ::= p|\neg\phi|\phi_1 \vee \phi_2|\phi_1 \exists\mathcal{U}_{\#n}\phi_2|\phi_1\forall\mathcal{U}_{\#n}\phi_2$$

where $p \in P$, n is a natural number and $\#$ is one of the relational operators $<$, $\leq$, $=$, $\geq$, or $>$.

TCTL formulae are interpreted over the sequences of states generated by the transition system of a timed graph. The details can be found in [13]. Intuitively, $\phi_1\exists\mathcal{U}_{\#n}\phi_2$ means that there exists a sequence with a finite prefix such that ϕ_2 is satisfied by the last state at time t where $t\#n$ and ϕ_1 is satisfied continuously until then. $\phi_1\forall\mathcal{U}_{\#n}\phi_2$ means that for every sequence this property holds. A number of abbreviations are commonly used: $\forall\Diamond_{\#n}\phi$ for $\texttt{true}\forall\mathcal{U}_{\#n}\phi$, $\exists\Diamond_{\#n}\phi$ for $\texttt{true}\exists\mathcal{U}_{\#n}\phi$, $\exists\Box_{\#n}\phi$ for $\neg\forall\Diamond_{\#n}\neg\phi$, and $\forall\Box_{\#n}\phi$ for $\neg\exists\Diamond_{\#n}\neg\phi$.

TCTL is expressive enough to allow us to express most system properties of interest. For example, a bounded response property can be easily stated,

$$\forall\Box(\texttt{stimulus}\Longrightarrow\forall\Diamond_{\leq 5}\text{response})$$

which captures the requirement that after any occurrence of a `stimulus`, a `response` will always happen within 5 time units. Other useful properties such as bounded invariance, bounded inevitability, self-stabilization and so on can be expressed just as easily.

For a timed communication protocol the property of most interest is that, under certain assumptions, a message which is accepted for sending will eventually be delivered correctly within a certain time; in other words a bounded response property. In the context of our description of the alternating bit protocol, such a property can be stated as

$$init\Longrightarrow\forall\Box(\texttt{after(accept)}\Longrightarrow\forall\Diamond_{\leq 200}\texttt{enable(deliver)})$$

Of course it is possible also to state and check properties concerning the correct operation of the protocol, namely that the sending of a message strictly alternates with the receiving of a message and that whenever a 0-tagged (respectively, 1-tagged) message is sent a 0-tagged (respectively, 1-tagged) message is received [9].

It is of most interest in this case to explore the design of the protocol by checking these properties under varying assumptions about the transmission and acknowledgement channels, in much the same way as we discussed in section 4 on the use of the simulator. We have used KRONOS to check the bounded response property for the protocol assuming error-free communication and also assuming that the transmission channel loses at most one message between successful deliveries. It can be seen easily how this approach can be extended to check this property under more elaborate assumptions about possible communication faults including lost, garbled or duplicated transmissions and/or acknowledgements.

6 Implementation Via Code Generation

Having validated and verified the AORTA system, there are semi-automatic techniques for implementing the design. These techniques are discussed in some detail in [3, 5]. In this section we concentrate on the implementation of the individual processes, and its relationship to the construction of timed graphs for model-checking described in section 5. Once the individual processes have been constructed they can either be executed separately (as would be the case for a distributed implementation of the alternating bit protocol), or multitasked on the same processor. Multitasking complicates the issue of timing, but this is addressed using a dedicated AORTA kernel [3].

The construction of code to implement an AORTA process can be done for any imperative language which admits timing analysis. Here we use C for entirely pragmatic reasons, viz. the availability of cross compilers and timing tools [21, 20]; in effect we use C as a portable assembler. Although some work has been done on implementing process algebra systems using synchronous languages, our work uses more standard techniques, and does not rely on the standard synchronous language assumptions about the immediate responsiveness of the computer system.

Code can be generated for all parts of the program which are related with communication, choice, time-out and recursion, which accounts for all of the `Send` and `Reply` processes. The structure of the process is built up in exactly the same way as the timed graph for the process, with each state of the timed graph corresponding to a label within the generated C program. Thus a transition to a state corresponds to a C `goto` statement. Communication is handled by a kernel call [3], which takes an array of gates to offered in choice, and returns a value corresponding to which gate communicates first. For a simple communication with no choice, such as is found in the `Send` process in the equation

```
Send0 = send0.Sending0
```

the generated code looks like this

```
/* process section Send0. Code for
send0.Sending0 */

Send0_1:
gatenames[0] = GATEsend0;
gatenames[1] = 0;

switch (communicate(PROCSend,gatenames,gatevalues)){
case 1: goto Send0_2;
}
```

where the array `gatenames` contains the names of the gates, terminated with a 0, and the array `gatevalues` is used for passing data in and out during communication (not used here for simplicity). The label `Send0_2` is used to pass control to the process `Sending0`, via another `goto`. It would be possible to use a more sophisticated approach which eliminated the 'goto a goto', but this will be done by most compilers anyway, so the resulting object code will have exactly the same structure as the corresponding timed graph.

A more complicated communication, which offers a choice and has a time-out can be found in the definition

```
/* process section Sending0. Code for
(ack0.Accept1 + ack1.Sending0)[100.0,101.0>Send0 */

Sending0_1:
gatenames[0] = GATEack0;
gatenames[1] = GATEack1;
gatenames[2] = 0;

switch (communicatet(PROCSend,100000,gatenames,gatevalues)){
case 0: goto Sending0_4;
case 1: goto Sending0_2;
case 2: goto Sending0_3;
}
```

Here there are two gates in the `gatenames` array, and the kernel call has an extra argument which specifies the minimum real-time clock increment required to activate the time-out. This time the value

0 is returned if the time-out takes place, and the values 1 and 2 correspond to communications on the gates `ack0` and `ack1` respectively.

In the `Send` process there are no computation delays or non-deterministic choices, but these are implemented by annotating the design with the relevant piece of hand-written C code or branch condition, so that they are inserted into the code at the correct point (corresponding to the relevant node or edge of the timed graph).

Using these small pieces of code, connected by `goto`s, the whole process is built up, forming a graph which corresponds to that described in section 5 in topology at least. The labels on the graph nodes and edges define the timing behaviour of the system. This timing behaviour is guaranteed by the timing analysis of the kernel, combined with code timing of any pieces of computation [3]. Having formed a graph topologically equivalent to the timed graph, and with the timing constraints on nodes and edges guaranteed by the kernel, we can have confidence that any properties of the system proved by model-checking will hold of the implemented system. The implementation of communication is handled entirely by the kernel. Internal connections are managed by checking through a list of pairs of connected gates, and external communication is achieved by supplying an I/O function in a standard form, which is called if the corresponding gate is waiting for communication. Because these I/O functions are in a standard form they are very easily replaced, so help to make small-scale prototyping very easy.

If only some of the processes are to be implemented in this way, then model-checking can only prove properties based on assumptions about the way other processes will behave. In the case of the alternating bit protocol, this means that we have to make assumptions about the way the `Ack` and `Trans` buffers will behave. If they behave in the way they are modelled (losing at most one copy of any single message, for example) then verified propertied of the whole system will hold.

7 Conclusion

We have described AORTA, an implementable real-time algebra, and shown how it can be used to model the alternating bit protocol which, although simple, captures many important features of communication protocols. We have also shown how AORTA systems can be validated via simulation, and formally verified through model-checking. Finally, implementation techniques for AORTA systems have been partly described, and a relationship established with the timed graph model used in the formal verification. AORTA has been used to describe more complex systems, such as a car cruise controller [4] and a submersible control and logging system [7], which has been implemented using the techniques outlined here. Although translation of process algebras to timed graphs is not new [18], and timed graphs have been used for model-checking of timed protocols [11], the novelty of our work is in providing a tool-supported framework in which timed systems can be designed, tested, verified and verifiably implemented.

Current and future work on AORTA includes the development of more efficient model-checking algorithms, the use of more advanced scheduling algorithms for implementation, and further investigation into the distributed and parallel implementation of systems.

Acknowledgements

The authors would like to thank the University of Northumbria at Newcastle and Northern IT Research for their financial support.

References

[1] R Alur, C Courcoubetis, and D Dill. Model-checking for real-time systems. In *IEEE Fifth Annual*

Symposium On Logic In Computer Science, Philadelphia, pages 414–425, June 1990.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2 – 34, 1993.

[3] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. *Microprocessors and Microsystems*, 18(9):513–521, November 1994.

[4] S Bradley, W D Henderson, D Kendall, and A P Robson. Application-Oriented Real-Time Algebra. *Software Engineering Journal*, 9(5):201–212, September 1994.

[5] S Bradley, W D Henderson, D Kendall, and A P Robson. Designing and implementing correct real-time systems. In H Langmaack, W-P de Roever, and J Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer-Verlag, September 1994.

[6] S Bradley, W D Henderson, D Kendall, and A P Robson. Modelling data in a real-time algebra. Technical Report NPC-TRS-95-1, Department of Computing, University of Northumbria, UK, 1995. Submitted for publication.

[7] S Bradley, W D Henderson, D Kendall, A P Robson, and S Hawkes. A formal design and implementation method for systems with predictable performance. Technical Report NPC-TRS-95-2, Department of Computing, University of Northumbria, UK, 1995. Submitted for publication.

[8] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[10] R Cleaveland, J Parrow, and B Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[11] C. Daws, A. Olivero, and S. Yovine. Verifying et-lotos programs with kronos. 1994.

[12] E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[13] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[14] G. Leduc and L. Léonard. A timed lotos supporting dense time domain and including new timed operators. *Formal Description Techniques*, V:87–182, 1993.

[15] K.L. McMillan. *Symbolic Model Checking: An approach to the State Explosion Problem.* Kluwer, 1993.

[16] R Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[17] F Moller and C Tofts. A temporal calculus of communicating systems. Technical Report ECS-LFCS-89-104, Edinburgh University, December 1989.

[18] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions of Software Engineering*, 18(9):794 – 804, 1992.

[19] A. Olivero and S. Yovine. *Kronos: A tool for verifying real-time systems – Users' guide and reference manual – draft 0.0*, 1993.

[20] C Y Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[21] C Y Park and A C Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[22] S Schneider, J Davies, D M Jackson, G M Reed, J N Reed, and A W Roscoe. Timed CSP: Theory and practice. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 640–675. Springer-Verlag, June 1991.

[23] S. Yovine. *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, May 1993.