An Application Oriented Real-Time Algebra

Steven Bradley William Henderson David Kendall
Adrian Robson *

Abstract

Many attempts have been made to define timed process algebras as a route to formal reasoning about real-time systems. In this paper we argue that existing timed process algebras unsuccessfully try to address all of the aspects which their untimed counterparts do — specification, design and modelling — where they would be more useful if they were restricted to one of these roles. Drawing on this, an Application Oriented Real-Time Algebra (AORTA) is introduced, which has special features making it suitable for the design of real-time systems which may need to be formally verified.
Keywords: real-time, formal methods, process algebra

1 Introduction

Process algebras such as CCS [1], CSP [2] and LOTOS [3] have proved themselves to be useful tools in the formal specification and verification of concurrent communicating systems. One of the reasons for their success is their broad spectrum of uses, from specification of system behaviour to high-level system modelling to parallel programming; the availability of automated proof techniques for relating different levels of abstraction or checking properties adds to their attraction [4].

Although very successful within their domain, such process algebras are limited in the aspects of a system which they can model or specify, and in particular they cannot represent the actual time between events, only the ordering in time of these events. Trying to build on their success, and to provide the basis for a formal method for real-time systems, many timed process algebras have been developed, including extensions to existing algebras [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] — for reviews see [17, 18]. In section 2 we argue that, while useful, timed process algebras cannot as easily be used as broad-spectrum languages (in the way that CCS and Z [19] can be), and that a different approach needs to be taken. The focus of the rest of the paper is then made clear in section 3, where an Application Oriented Real-Time Algebra (AORTA) is introduced. Examples of using AORTA are given in section 4 and the language and semantics are formalised in section 5. In section 6, some ideas are given as to how an AORTA design can be implemented, and finally, section 7 presents some conclusions, and outlines directions for further work.

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

2 The Problems of Including Time

There are many technical problems to overcome in the definition of a timed process algebra, but before considering any of these, it is worth looking at the motivation for a timed algebra, and the extent to which standard (untimed) process algebra techniques can be extended to timed situations. Each of the applications for untimed process algebras mentioned above — specification, modelling and design — are still of interest in a timed scenario, but we argue that they cannot all be handled as well by a timed algebra.

The main reason for the difficulty in applying many timed process algebras is the level of detail of the behaviour which they describe. In untimed algebras only the ordering of events is considered, and this seems to lie at the level of detail which is just right for many systems: to use standard examples, it is important that a coffee machine should not offer a drink before a coin has been inserted; a communications protocol should not wait for an acknowledgement until after it has sent a message; a level crossing should not allow cars to cross the track after it knows a train is approaching. It could be argued that these examples have become standard because they show off untimed formalisms to good advantage, but it does appear that this level of abstraction is a useful one in many cases. The level of detail given by many timed process algebras, however, is very much higher, as not only the order of events but the *exact* time at which they occur or become available is given. Although some notion of time is important in many reactive systems, nearly all behaviours are better specified or modelled by time *bounds*: a nuclear power plant controller must respond to a rise in core temperature within a certain amount of time, and a set of traffic lights must leave sufficient time for all cars to get past in one direction before allowing the other cars to cross. Bounds are not only more useful in specification, but also in modelling and design, as most systems cannot guarantee exact performance, due to unpredictability of program execution times, communication delays and scheduler performance, but most can guarantee maximum and/or minimum times.

There are two common methods for verifying correctness of systems using process algebras — bisimulation and model-checking — and these have both been extended for use with timed process algebras. Although model-checking, in which properties stated in a timed logic are tested for a timed algebra term, does seem to extend well, the idea of bisimulation, which is a cornerstone of untimed process theory, suffers from the level of detail involved in timed process algebras. In a bisimulation a relation is made between terms which have the same behaviour, and in a timed bisimulation related terms must have the same behaviour in time. It seems that the level of detail given in existing timed process algebras is such that bisimulation equivalence makes too fine a distinction between systems, as is borne out by the profusion of definitions of timed bisimulations, but the lack of examples of equivalent systems (this view is supported in [20]). If we accept that bisimulations are not very applicable to timed systems, we have two alternative approaches to finding verification methods for timed process algebras:

1. Use process algebra terms purely for representing designs, and adopt other languages, such as temporal logics, for high-level specification. Verification methods such as model-checking can then be used.
2. Develop new methods which still use process algebras at different levels of

abstraction, using a notion of refinement instead of bisimulation.

In this paper we adopt the first approach, using a timed process algebra for representing designs, and relying on model-checking (or at least hand verification of timed logic specifications) as our proof method. Having said that, there is a large literature on timed logics and proof techniques (including model-checking) [21, 22, 23, 24, 25, 18, 15], so we are going to leave aside this issue for the moment, and concentrate on the implications of using timed process algebras solely for representing designs.

If an algebra is to be used as a design language, careful consideration must be given to how terms in the algebra (i.e. designs) are to be implemented. In our algebra, AORTA, more restrictions are placed on terms than in other algebras, precisely because the restrictions make implementation easier. Some of the most important differences are because of the difficulty of guaranteeing the time performance of a real-time system: parallel composition may only take place at the top level in order to fix the number of processes, as time guarantees then become easier to give (see section 6); time bounds on performance and communication times can be given rather than precise figures. Implementing multiway synchronisation and broadcast events is difficult, particularly where performance figures are needed, so communication may only take place between pairs of explicitly named gates. The question may be raised as to whether such a restricted process algebra is still useful. We would argue that it is useful as an implementable design language (almost a programming language) which has a formal semantics, and so allows formal verification of the timing aspects of safety-critical systems from specification to implementation. Although more detailed justification needs to be given of the reasons for our choices (and more will be given in this paper), for the moment we move on to the development of a timed process algebra which is useful as a design language.

3 Introducing AORTA

In keeping with the conclusions of the previous section we now introduce an application oriented real-time algebra (AORTA), which has certain features making it more suitable for representing designs of real-time systems (including timing information) than for giving specifications. AORTA can almost be thought of as a programming language with a formal semantics, and although there are no automatic compilation techniques, there are ways of implementing AORTA designs. More of this in section 6; for the moment, we concentrate on the language as a process algebra.

There are several ways of describing a process algebra: an informal description of the constructs of the language is very helpful, and a formal semantics is at least as important. There are also different ways of giving a formal semantics, the three main types of semantics being operational semantics, denotational semantics and algebraic semantics, these three being represented in the process algebra world by CCS, CSP and ACP respectively. These three presentation techniques are not mutually exclusive — a lot of work done with CCS is concerned with equational (algebraic) reasoning, CSP has been given an operational semantics, and operational transition rules are used in ACP — so a good conceptual understanding can be as important as a detailed knowledge of the formalism concerned. In this paper we use operational semantics given by transition rules, but before that is an informal introduction to AORTA and

some examples.

3.1 Concrete Syntax and Informal Semantics

Of the common untimed algebras, AORTA is most similar to CCS, both in notation (`.` for action prefixing and `+` for choice) and in semantics (only two-way synchronisation allowed), but even apart from the time considerations there are some important differences. One of the restrictions placed on the language to aid implementation is that the number of processes in a system may not vary, and this restriction is enforced by insisting that all parallel composition should happen at the top level. This gives rise to two levels of description: one for the sequential processes within a system, and another for the parallelism and connectivity of the system. Restricting systems to a fixed number of processes is not uncommon in real safety-critical systems, and the limitations imposed are partly justified by the verifiable implementation techniques described in section 6. Some familiarity with CCS is assumed in the following.

3.1.1 Sequential Processes

The description of sequential processes is where the relation of AORTA to CCS is shown most strongly. Actions can be offered, which must be matched by a communicating partner before the process can proceed, and a choice may be offered between a number of actions. As in CCS, action prefix and choice (sometimes called summation) are represented by `.` and `+` respectively, with `0` for the null process which offers no actions. Recursion can be written using the same equational format as used in CCS (e.g. `A = a.A`), but all recursion must be guarded (i.e. all process names must appear inside an action prefix). The other constructs do not have analogues in CCS, and are concerned with including time information into the process description.

There are two constructs which are used to introduce time, and each of these has a deterministic and nondeterministic form. The first construct is a delay which causes the process to pause for the amount of time specified, during which time no actions are offered — time consuming operations like computation are represented in this way. As precise times are not always known, the delay may be specified with an upper and lower bound, rather than a precise figure. A process which delays for precisely t time units before behaving like S is written `[t]S`, and if the delay is bounded by times $t1$ and $t2$ the process is written `[t1,t2]S`. The second construct is a timeout extension to summation, so that if none of the branches of the choice are taken up within the given time, control is transferred to another branch. Again, depending on how the timeout is implemented a precise figure for the time at which control is transferred may not be available, so an interval of possibilities can be given instead. A choice process S which times out to process T if no communication happens within time t is written `S [t> T`, and if the time is bounded by $t1$ and $t2$ it is written `S [t1,t2> T`.

Having given the time behaviour of our new constructs it is necessary to go back to describe the time behaviour of prefix and choice. A simple prefix forces the process to wait until communication can take place on the named channel, so the process `a.S` can wait for any length of time without changing, provided communication is not possible. Consideration of how a choice should behave in time leads us to restrict choice to processes which start with an action

prefix	`a.S`
choice	`S1 + S2`
delay	`[t]S`
bounded delay	`[t1,t2]S`
timeout	`(S1 + ... + Sn)[t>S`
bounded timeout	`(S1 + ... + Sn)[t1,t2>S`
data dependent choice	`S1 ++ S2`
recursion	equational definition

Table 1: Summary of concrete syntax for sequential processes

prefix or another choice. If a choice were allowed between processes that began with a delay, e.g. `[3]a.0 + [2]b.0`, then either the choice would have to be resolved at the first instant of time, leading to time nondeterminism (and a very counter-intuitive system), or both branches of the choice would have to run concurrently, which goes against the idea of a sequential process. As both of these are unacceptable, we restrict the language so that choices can only be made between processes which start with an action prefix or another choice.

One of the reasons for uncertainty in the execution times of programs is that there is no information available about the data on which the program is running — we either don't know what the data is or we choose to ignore it to avoid complexity. At the moment no attempt is made to model data in AORTA, so any branch in a sequential process which depends purely on data (in particular on the outcome of a computation) rather than on communication (which is handled by the existing choice) appears to be nondeterministic. To allow for such branches, a data dependent (or nondeterministic) choice can be offered between two (or more) processes: such a choice is written `P++Q`, and is similar to the nondeterministic choice $P \sqcap Q$ of CSP.

In summary, a sequential process may be constructed from action prefixes, summations (choices over prefixed processes), time delays, timeouts over choices, nondeterministic choices and guarded recursion. The syntax is summarised in table 1. Each process has a behaviour in time which says which actions it is prepared to engage in, or in other words, at which of its gates it is prepared to engage in communication. Obviously, for communication to take place there has to be more than one process in the system — the way that a system is constructed from its component processes is kept separate from process definition in AORTA.

3.1.2 Parallel Composition and Communication

Apart from fixing the number of processes in a system in order to provide reliable timing predictions (see section 6), there are other steps which can be taken to aid implementability. One area which is crucial to process algebras and real-time systems is inter-process communication, and this is perhaps where AORTA is most different from existing process algebras.

In all of the common process algebras the communication actions of any process are visible to any other process unless explicitly hidden or restricted, which leads to problems on two fronts. From an implementation point of view this requires some way of broadcasting all available actions to all processes. Even more problems are encountered in implementing the multiway synchronisation

of CSP and LOTOS, as witnessed by the restriction to two-way communication in occam [26] and the need for a special protocol in LOTOS [27]. For a simple system, which is all we can hope to formally verify at the moment, the mechanism for providing such communication facilities may be an excessively costly overhead, both in terms of implementation and verification.

The availability of all actions to all processes can also cause problems in verification, as checking for all possible communications requires testing of each pair of processes for communication on each action, leading to an explosion in the number of checks to be made. This explosion can be contained by restricting communication to a named set of channels between processes. There is an analogy here with sequential programming, where the techniques of object-oriented and functional programming have tried to limit the means of access to each part of the program data, making reliable and verifiable design easier. As AORTA is to be used as a kind of parallel programming language which admits verification, similar restrictions on the availability of program data and communication will ease verification.

In the light of these problems, AORTA requires explicit connections to be made for a communication to become possible, and these connections are made statically in the system definition. Each process has a set of named gates (like the syntactic sort of CCS), and communication links between processes are made by explicitly naming pairs of gates to be linked. By using explicit linking the restriction or hiding operators of other process algebras are not needed, and by allowing gates with different names to be linked, renaming operators become unnecessary. The use of explicit connections may appear to restrict the use of compositional verification techniques, such as are described in [25], which could cause problems given the complexity of the model-checking problem. However, as we shall see later, the semantics of AORTA is layered, separating the sequential process behaviour from the derived system behaviour. At the sequential process level, all communications, internal and external, are represented in the same way, allowing abstract reasoning about individual processes and compositional reasoning at the system level.

Two or more processes may be put in parallel using `|`, so that `P|Q|R` represents three processes in parallel, where each of `P`, `Q`, and `R` is a sequential process. In order to enable communication, a collection of processes may have some pairs of gates linked, using a connection set written in angle brackets after the processes. An element of the connection set is a pair of gates to be connected, but this can be abbreviated to a single name if both ends of the link have the same name. In section 4, there are some examples to show the notation in practice, and then we give a more abstract syntax and a formal semantics for AORTA in section 5.

4 Examples in AORTA

In this section there are two examples to show how AORTA can be used. One feature of the concrete syntax just given is that it uses only standard ASCII characters, and this is emphasised by using `typewriter script` when using the concrete syntax. The order of binding (tightest first) for sequential processes is `. [..] + [..> ++` and then recursion, and parallel composition constructs `| <..>` bind most loosely of all. As usual, brackets can be used to override this ordering.

4.1 A Mouse Button

One system which cannot be expressed in an untimed algebra is that of a mouse which can either be clicked or double-clicked. If the mouse button is clicked twice within 250ish milliseconds then a double-click event will be offered; otherwise a press of the button will yield a single click event. This system quite naturally uses a timeout in its implementation, which is reflected in its expression in AORTA:

```
Mouse = click?.(click?.double!.Mouse)[0.245,0.255>single!.Mouse
```

`Mouse` is a recursive process, with recursion defined using an equational format. The timeout `[0.245,0.255>` ensures that if a click is not followed by another within about 250 milliseconds then a single click is offered. If the second click occurs within 245 milliseconds of the first then it will definitely be accepted as a double, and it may be accepted as such up to 255 milliseconds after the first. An implementation would follow quite easily from this, with a process which waited for a click, and then read a clock, before waiting for either another click or the current clock value to exceed the old one by 250 milliseconds. Depending on which happens first a double or single click event will be offered before returning to wait for another click. As long as the clock was accurate to within 5 milliseconds it would have behaviour modelled by the AORTA expression, so any reasoning done on that expression would apply to the implementation. Note that we use gate names with an exclamation or question marked attached, corresponding to 'output' and 'input', but that this is purely for the ease of the reader and is not required.

4.2 A Car Cruise Controller

A standard example of a real-time safety-critical system is a cruise controller for a car, which was used as an example for comparing different methodologies, and is outlined in [28]. An extract from the specification (taken from [28]) is

> The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. The driver must be able to enter several commands, including: Activate, Deactivate, Start Accelerating, Stop Accelerating, and Resume. The cruise control function can be operated any time the engine is running and the transmission is in top gear. When the driver presses Activate, the system selects the current speed, but only if it is at least 30 miles per hour, and holds the car at that speed. Deactivate returns control to the driver regardless of any other commands. Start Accelerating causes the system to accelerate the car at a comfortable rate until Stop Accelerating occurs, when the system holds the car at this new speed. Resume causes the system to return the car to the speed selected prior to braking or gear shifting.
>
> The driver must be able to increase the speed at any time by depressing the accelerator pedal, or reduce the speed by depressing the brake pedal. Thus, the driver may go faster than the cruise control setting simply by pressing the accelerator pedal far enough. When the pedal is released, the system will regain control. Any time the brake pedal is depressed, or the transmission shifts out of top gear,

> the system must go inactive. Following this, when the brake is released, the transmission is back in top gear, and Resume is pressed, the system returns the car to the previously selected speed. However, if a Deactivate has occurred in the intervening time, Resume does nothing.

It also adds that in the implementation

> The system controls the car through an actuator attached to the throttle. This actuator is mechanically in parallel with the accelerator pedal mechanism, such that whichever one is demanding greater speed controls the throttle . . . For smooth and stable servo operation the system must update its outputs at least once per second.

An AORTA implementation of such a system can be given by breaking the system into four processes: the speedometer system, the controller system, the throttle system and the brake and gear system.

Looking firstly at the speedometer system, we assume that there is a measurement that can be made which will give the speed, but the acceleration is also needed (for the Accelerate function), and this is calculated and offered by this process. About every half a second the speed is reread and a new value for the acceleration calculated, and for the rest of the time there are two channels on which the speed is available (one for the controller and one for the throttle mechanism) and one channel with the acceleration. This is written in AORTA as

```
Speedo1 = (speedout1!.Speedo2 + speedout2!.Speedo2 +
             accelout!.Speedo2)[0.4,0.5>Speedo2
Speedo2 = speedin?.[0.2,0.3]Speedo1
```

The delay `[0.2,0.3]` corresponds to the time taken to calculate the new value for the acceleration.

The system which monitors the brakes and gears is also quite simple, and simply checks the state of the gears and brakes, and depending on what it finds offers a `fast!` action, indicating that everything is fine or a `slow!` action, indicating that either the brakes are on or the transmission is not in top gear. The choice as to which to offer depends on the data which comes from `gearstate?` and `brakestate?` actions, and so is represented by a data dependent choice `++`. As in the speedometer, new readings are taken about every half a second, giving the definition

```
Brakengear1 = (fast!.Brakengear1)[0.4,0.5>
                gearstate?.brakestate?.(Brakengear1 ++ Brakengear2)
Brakengear2 = (slow!.Brakengear2)[0.4,0.5>
                gearstate?.brakestate?.(Brakengear2 ++ Brakengear1)
```

A slightly more complex system is the throttle system, which has to monitor the speed and ensure that it is kept at the speed specified by the controller, and which sometimes has to enter an accelerating phase, when it must keep control of the acceleration. Because of the parallel accelerator pedal mechanism, manual control will be resumed if the accelerator pedal is down further than the cruise

control actuator, and in particular, if the actuator is set to zero the driver has complete manual control over the throttle. The output of the throttle controller system is the `setthrottle!` action, which passes a value for the actuator to be set at. In usual operation, the system monitors the speed (via `getspeed?`) and adjusts the throttle about every half a second. It may also allow a new speed to be set (by `setspeed?`), the speed to be set to zero (`resetspeed?`), or put the system into an accelerating phase (`accelon?`). During the accelerating phase the acceleration is monitored and adjusted until told to stop accelerating (`acceloff?`), when the current speed is read for use as the new control speed and control returned to the usual state. If a `resetspeed?` is encountered, the system waits for a new value to be passed via `setspeed?` before returning to usual operation. All of this is described in AORTA by

```
Throttle1 = (setspeed?.Throttle1 + accelon?.Throttle2 +
              resetspeed?.Throttle3)[0.4,0.5>
                getspeed?.setthrottle!.Throttle1
Throttle2 = (acceloff?.getspeed?.Throttle1)[0.4,0.5>
              getaccel?.setthrottle!.Throttle2
Throttle3 = setthrottle!.setspeed?.Throttle1
```

The system which has overall control, and which accepts commands from the driver (or at least from an interface to the driver) has a more complicated logical structure, but does not have any real computation to do nor any time dependent behaviour in the form of timeouts. There are four major states of the controller, corresponding to the controller being inactive, the controller being active, the controller being in an accelerating state, and the controller waiting for a Resume after the brake being pressed or the transmission leaving top gear. Before entering an activated state, the controller has to check that the brakes are not on and that the transmission is in top, which it does by looking for a `fast?` from the brake and gear system — if a `slow?` is encountered the system must be reactivated. The speed is checked, and if it is greater than 30 mph the controller becomes active, otherwise the system must be reactivated. As this decision is based on the data about the speed, it is modelled by a data dependent choice `++`, giving the structure

```
Cont1 = activate?.(fast?.checkspeed?.(Cont1 ++ setspeed!.Cont2) +
          slow?.Cont1)
```

Once activated, the system must allow itself to be deactivated, or suspended due to a brake/transmission event, or put into the accelerating step. This is achieved by

```
Cont2 = (deactivate?.setspeed0!.Cont1 +
          startaccel?.Cont3 + slow?.Cont4)
```

During acceleration the system can be deactivated, or stopped from accelerating, or suspended by a brake/transmission event

```
Cont3 = accelon!.(deactivate?.acceloff!.setspeed0!.Cont1 +
          stopaccel?.acceloff!.Cont2 + slow?.acceloff!.Cont4)
```

Finally, if a brake/transmission event occurs, the speed must be reset and resumption or deactivation allowed, checking that the brakes and gears are in order if necessary.

```
Cont4 = setspeed0!.(resume?.fast?.setspeed!.Cont2 +
           deactivate?.Cont1)
```

This design is certainly not the only one that could be employed, and probably contains some logical errors (for instance, if a **slow** event occurs followed by a **resume** before the brakes and gears are OK the system may deadlock), but it does show how AORTA can be used for designing a such a system. This initial design may be simulated to iron out any obvious problems, and it may be checked against a formal specification for inconsistencies or errors. The whole of the design (including communication links) is given by

```
Cont1 = activate?.(fast?.checkspeed?.(Cont1 ++ setspeed!.Cont2) +
           slow?.Cont1)
Cont2 = (deactivate?.setspeed0!.Cont1 +
           startaccel?.Cont3 + slow?.Cont4)
Cont3 = accelon!.(deactivate?.acceloff!.setspeed0!.Cont1 +
           stopaccel?.acceloff!.Cont2 + slow?.acceloff!.Cont4)
Cont4 = setspeed0!.(resume?.fast?.setspeed!.Cont2 +
           deactivate?.Cont1)

Speedo1 = (speedout1!.Speedo2 + speedout2!.Speedo2 +
             accelout!.Speedo2)[0.4,0.5>Speedo2
Speedo2 = speedin?.[0.2,0.3]Speedo1

Brakengear1 = (fast!.Brakengear1)[0.4,0.5>
                gearstate?.brakestate?.(Brakengear1 ++ Brakengear2)
Brakengear2 = (slow!.Brakengear2)[0.4,0.5>
                gearstate?.brakestate?.(Brakengear2 ++ Brakengear1)

Throttle1 = (setspeed?.Throttle1 + accelon?.Throttle2 +
              resetspeed?.Throttle3)[0.4,0.5>
                getspeed?.setthrottle!.Throttle1
Throttle2 = (acceloff?.getspeed?.Throttle1)[0.4,0.5>
              getaccel?.setthrottle!.Throttle2
Throttle3 = setthrottle!.setspeed?.Throttle1

Cruisesys = (Cont1|Speedo1|Brakengear1|Throttle1)
             <(Cont1.checkspeed?,Speedo1.speedout2!),
              (Cont1.acceloff!,Throttle1.acceloff?),
              (Cont1.accelon!,Throttle1.accelon?),
              (Cont1.setspeed0!,Throttle.restspeed?),
              (Cont1.setspeed!,Throttle1.setspeed?),
              (Cont1.slow?,Brakengear1.slow!),
              (Cont1.fast?,Brakengear1.fast!),
              (Speedo1.accelout!,Throttle1.getaccel?),
              (Speedo1.speedout1!,Throttle1.getspeed?)>
```

The parallelism and communication channels of the system are represented graphically in figure 1. Another small example, of a temperature conversion process, is described in [29].

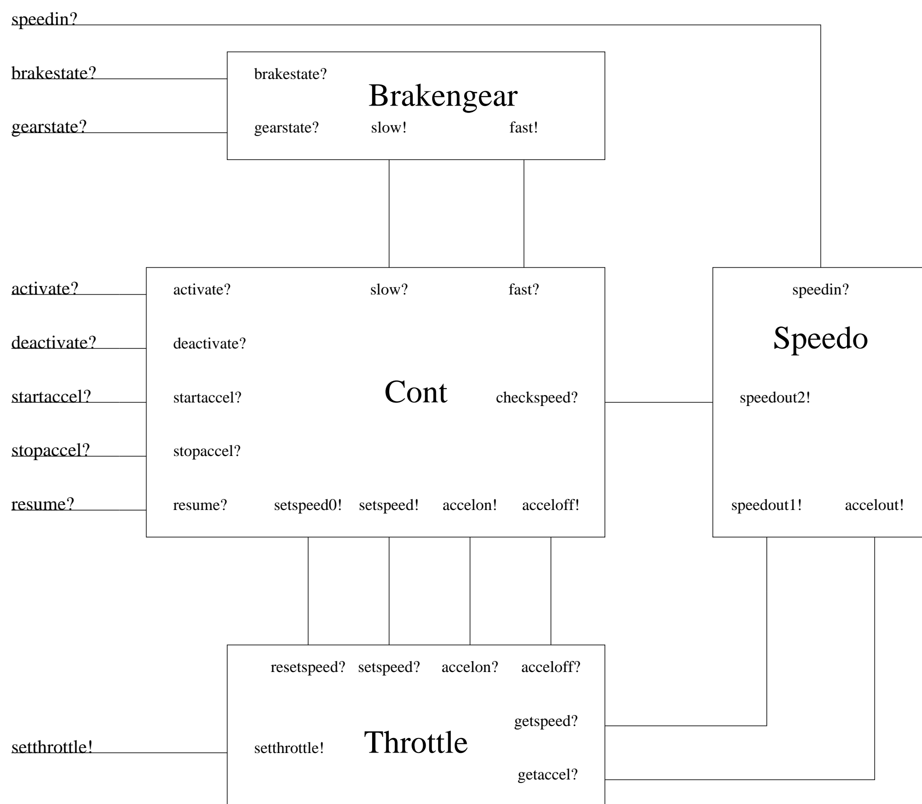

Figure 1: Parallelism and Connectivity of Cruisesys

5 Formalising AORTA

5.1 The Abstract Syntax

The syntax presented here is an abstract syntax, terms of which may be expressed in the concrete syntax given in section 3.1: the translation between the two is mostly straightforward, but some of the finer points are dealt with later.

A system is expressed as a product of sequential processes, each of which has a set of gates; gates of processes can be connected pairwise to allow communication. A system expression is then written

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

where each S_i is a sequential process, and K is a set of unordered pairs of gates to be linked. Each gate is specified by its process (i.e. an element of I) and its name, and may be connected to at most one other gate. At this level the communication delay bounds for each gate must also be specified; for linked pairs the bounds will be the same and will depend on internal communication delays, but for unlinked gates (i.e. gates that communicate with the environment) the delay will depend on what is being accessed and how. The delay bounds are specified by giving a function *delays* which takes a gate identifier, and returns an interval of possible communication delay times. This function is defined at the system level, when the details of how gates of processes are to be connected is known.

The structure of sequential expressions is given by the syntax

$$S ::= \sum_{i \in I} a_i.S_i \;\mid\; [t]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd^t S \;\mid\; [t_1, t_2]S \;\mid\; \sum_{i \in I} a_i.S_i \rhd^{t_2}_{t_1} S \;\mid\; \bigoplus_{i \in I} S_i \;\mid\; X$$

where t, t_1 and t_2 ($t_1 < t_2$) are time values taken from the time domain (either the positive reals or the naturals), and X is taken from a set of process names used for recursion. These constructs correspond to action summation, deterministic delay, deterministic timeout, nondeterministic delay, nondeterministic timeout, nondeterministic choice and recursive definition respectively.

There is a subset of these sequential expressions, the regular expressions, which is defined to be the set of expressions which evaluate to true under the function *regular* given in figure 2. Informally, regular expressions are those expressions which do not involve any nondeterminism (via delays, timeouts or nondeterministic choices) before the first action summation, and which contain only guarded recursion — the condition on guardedness is that all recursion should be guarded once reformulated as a fixed point.

The translation of a concrete syntax term into the abstract syntax is fairly direct, but does raise some interesting points. Although we do not wish to deal with bisimulations, for reasons already stated, there is an equivalence which yields some of the less interesting equalities on terms, namely syntactic equality on abstract syntax terms, modulo arithmetic and set equality. Because choice and parallel composition are indexed by sets in the abstract syntax, it does not matter in which order the subterms appear in the concrete syntax. This renders laws such as the commutativity and associativity of the concrete syntax `+` and `|` immediate, as well as the law `P + 0 = P`, where `0` has the usual translation of summation over the empty set.

$$
\begin{aligned}
regular(\sum_{i \in I} a_i.S_i) &= true \\
regular([t]S) &= regular(S) \\
regular(\sum_{i \in I} a_i.S_i \rhd^t S) &= regular(S) \\
regular([t_1, t_2]S) &= false \\
regular(\sum_{i \in I} a_i.S_i \rhd^{t_2}_{t_1} S) &= false \\
regular(\bigoplus_{i \in I} S_i) &= false \\
regular(X) &= false
\end{aligned}
$$

Figure 2: Definition of *regular*

5.2 The Formal Semantics

In order to define the semantics of system expressions, the semantics of regular expressions is given first. As usual for an operational semantics, a set of transition rules is given, from which is constructed the least relation to satisfy all of the rules. This semantics depends heavily on the *Poss* function, which describes all of the possible ways in which a non-regular (i.e. nondeterministic or recursive) expression could be resolved into a regular expression. Rules are only defined for action summation, timeout and deterministic delays because all other terms are non-regular; note that all terms on the right hand side of transition arrows are regular (in particular, all elements of $Poss(S)$ are regular), so the transition relation is well-defined on regular expressions. There are two types of transitions, namely action transitions, written $\xrightarrow{a}$ where a is a gate name, and time transitions, written $\xrightarrow{(t)}$ where t is a value in the time domain. The rules are given in figure 3, and the auxiliary function *Poss* is defined in figure 4.

When an action transition takes place all nondeterminism up to the next action is resolved, by the use of the *Poss* function. This is not only convenient from a theoretical standpoint, but does correspond to the situation in a real system. From a theoretical point of view, it avoids the problem of time nondeterminism. Practically speaking, the nondeterminism comes from a lack of knowledge about data in the system, and a lack of predictability as regards scheduling and communication delays; once a process has communicated, all of its data is fixed until the next possible communication, and if a scheduling mechanism such as that outlined in section 6 is used then once the starting time of a computation or communication is known, its completion time can be calculated exactly.

Each system expression is described as a product of (regular) sequential expressions, and the transitions of a system are derived from the transitions of each of its component processes as would be expected. The transition rules for system expressions are given in Fig. 5, but the transition system cannot be formed as the usual least relation, because of the negative premise of the rule for delay. Problems with negative premises in transition system specifi-

$$\frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{a_j} [t']S'_j} \quad \begin{array}{l} j \in I, S'_j \in Poss(S_j) \\ t' \in delays(a_j) \end{array} \qquad \frac{}{\sum_{i\in I} a_i.S_i \xrightarrow{(t)} \sum_{i\in I} a_i.S_i}$$

$$\frac{}{[t]S \xrightarrow{(t')} [t-t']S} \; t' < t \qquad \frac{}{[t]S \xrightarrow{(t)} S}$$

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{a_j} [t']S'_j} \quad \begin{array}{l} j \in I, S'_j \in Poss(S_j) \\ t' \in delays(a_j) \end{array}$$

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t')} \sum_{i\in I} a_i.S_i \rhd^{t-t'} S} \; t' < t \qquad \frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{(t)} S}$$

$$\frac{S_1 \xrightarrow{(t_1)} S_2 \quad S_2 \xrightarrow{(t_2)} S_3}{S_1 \xrightarrow{(t_1+t_2)} S_3}$$

Figure 3: Transition rules for regular expressions

$$\begin{array}{rcl}
Poss(\sum_{i\in I} a_i.S_i) & = & \{\sum_{i\in I} a_i.S_i\} \\
Poss([t]S) & = & \{[t]S' | S' \in Poss(S)\} \\
Poss(\sum_{i\in I} a_i.S_i \rhd^t S) & = & \{\sum_{i\in I} a_i.S_i \rhd^t S' | S' \in Poss(S)\} \\
Poss([t_1,t_2]S) & = & \{[t]S' | t \in [t_1,t_2], S' \in Poss(S)\} \\
Poss(\sum_{i\in I} a_i.S_i \rhd^{t_2}_{t_1} S) & = & \{\sum_{i\in I} a_i.S_i \rhd^t S' | t \in [t_1,t_2], S' \in Poss(S)\} \\
Poss(\bigoplus_{i\in I} S_i) & = & \{S'_i | i \in I, S'_i \in Poss(S_i)\} \\
Poss(X) & = & Poss(S) \quad \text{if } X \stackrel{\text{def}}{=} S
\end{array}$$

Figure 4: Definition of $Poss$

Internal Communication

$$\frac{S_j \xrightarrow{a} S'_j \quad S_k \xrightarrow{b} S'_k}{\prod_{i \in I} S_i < K > \xrightarrow{\tau} \prod_{i \in I} S'_i < K >} \quad \begin{array}{l} (j.a, k.b) \in K \\ S'_i = S_i \text{ if } i \neq j, k \end{array}$$

External Communication

$$\frac{S_j \xrightarrow{a} S'_j}{\prod_{i \in I} S_i < K > \xrightarrow{a} \prod_{i \in I} S'_i < K >} \quad \begin{array}{l} j \in I \\ (j.a, _) \notin K \\ S'_i = S_i \text{ if } i \neq j \\ \prod_{i \in I} S_i < K > \stackrel{\tau}{\not\longrightarrow} \end{array}$$

Delay

$$\frac{\forall i \in I. S_i \xrightarrow{(t)} S'_i}{\prod_{i \in I} S_i < K > \xrightarrow{(t)} \prod_{i \in I} S'_i < K >} \quad \forall t' < t. \prod_{i \in I} Age(S_i, t') < K > \stackrel{\tau}{\not\longrightarrow}$$

Figure 5: Transition rules for system expressions

cations are discussed in [30], and a technique called stratification is provided to give a meaning to such transition systems. Applying this to AORTA, all of the transitions of sequential expressions should be worked out first, then all internal communications of system expressions, and finally the time transitions and external communications of system expressions. By applying the transition rules in three stages we ensure that no transition's validity depends on its own negation, as may be the case in a transition system with negative premises; this layering is equivalent to a three layer stratification. For further details, see [30].

The rule for external communication also has a negative premise attached, in order to enforce a simple priority on actions: here we insist that internal communications be preferred to external ones, as the permanent availability of some environment actions may make choices unfair. A similar technique can be used to attach a full set of priorities to the actions, both internal and external, allowing one internal communication to be preferred to another and so on. In order to give a well defined semantics to this, negative premises can be attached to all actions other than the highest, stating that the communication may not take place if any higher priority action is possible. A larger stratification is then used, with a different stratum attached to each priority level, as well as strata for sequential processes and time transitions.

As well as a stratification, the rule for delay uses an auxiliary function, Age, which is defined on regular expressions as in figure 6. This function is really meant to make the side-condition easier to express, as the Age function takes a process and a time, and returns the state of the process after having delayed for the specified time. This is captured in the theorem

Theorem 1 *For any regular sequential expressions S and S' and any time t*

$$S \xrightarrow{(t)} S' \iff Age(S, t) = S'$$

where = is syntactic identity modulo equality on time expressions

$$Age(\sum_{i\in I} a_i.S_i, t') = \sum_{i\in I} a_i.S_i$$

$$Age([t]S, t') = \begin{cases} [t-t']S & (t' < t) \\ S & (t' = t) \\ Age(S, t'-t) & (t' > t) \end{cases}$$

$$Age(\sum_{i\in I} a_i.S_i \rhd^t S, t') = \begin{cases} \sum_{i\in I} a_i.S_i \rhd^{t-t'} S & (t' < t) \\ S & (t' = t) \\ Age(S, t'-t) & (t' > t) \end{cases}$$

Figure 6: Definition of Age

The intuitive interpretation of the transition system formed by these rules is worth mentioning, as there is often some ambiguity, particularly in untimed algebras, as to what it all means. The two types of transition, $\xrightarrow{a}$ and $\xrightarrow{(t)}$ correspond to ability to communicate and ability to age. If $S \xrightarrow{a} S'$ then S is ready to communicate externally on gate a, and if this communication takes place the process will then become S'. If a system can communicate internally then it does (maximum progress principle, as enforced by the side condition on the delay rule), and this is represented by the distinguished action $\xrightarrow{\tau}$. If more than one τ action is possible then a nondeterministic choice is made between the available actions. The $\xrightarrow{(t)}$ transition describes how a system or process may age in time, and it is a property of the system that any process has only one way in which to age: in other words, it is time deterministic. The behaviour of a system is then represented by a a series of transitions, with the behaviour of the environment affecting which external communication events ($\xrightarrow{a}$) take place. As each communication has a minimum (non-zero) delay attached, only finitely many external events can occur within a finite time, so the system has finite variability; as the number of processes is fixed, it also has bounded variability [17]. This formal semantics is also presented in [31], along with a discussion of the problem of verifying the correctness of AORTA designs.

5.3 The Mouse Button Revisited

The mouse button process, which was described in section 4.1, is used here to show how the formal semantics of a process can be derived. Its definition was

```
Mouse = click?.(click?.double!.Mouse)[0.245,0.255>single!.Mouse
```

To define a system which uses this mouse, let us use a very simple computer, which reacts to one or two clicks on the mouse by performing a piece of computation.

```
Computer = one?.[0.4,0.5]Computer + two?.[1.2,1.4]Computer
```

Putting these in a system, we get

```
( Mouse | Computer )
<(Mouse.single!,Computer.one?),
 (Mouse.double!,Computer.two?)>
```
and define the *delays* function to give the interval $[0.001, 0.003]$ for all gates (internal and external).

Firstly, the sequential transitions within the system can be worked out, using the rules of figure 3. As both `Mouse` and `Computer` begin with choice, the time transitions are very straightforward

$$\texttt{Mouse} \xrightarrow{(t)} \texttt{Mouse}$$

$$\texttt{Computer} \xrightarrow{(t)} \texttt{Computer}$$

and the action transitions include

$$\texttt{Mouse} \xrightarrow{click?} [\texttt{0.0025}](\texttt{click?.double!.Mouse})[\texttt{0.249} > \texttt{single!.Mouse}$$

$$\texttt{Computer} \xrightarrow{one?} [\texttt{0.0012}][\texttt{0.41}](\texttt{one?}.[\texttt{0.4}, \texttt{0.5}]\texttt{Computer} \;+\; \texttt{two?}.[\texttt{1.2}, \texttt{1.4}]\texttt{Computer})$$

$$\texttt{Computer} \xrightarrow{two?} [\texttt{0.003}][\texttt{1.38}](\texttt{one?}.[\texttt{0.4}, \texttt{0.5}]\texttt{Computer} \;+\; \texttt{two?}.[\texttt{1.2}, \texttt{1.4}]\texttt{Computer})$$

There are many more possible action transitions of `Mouse` and `Computer`, as the $Poss$ function allows the resolution of nondeterminism to any time value within the bounds, but they can only be via a `click?`, `one?` or `two?` action.

The initial system transitions are derived from the initial sequential transitions by the rules of figure 5. As `click?` does not appear in the connection set, and it is the only possible action transition of `Mouse`, no internal communication (τ transitions) can take place. Also

$$Age(\texttt{Mouse}, t) = \texttt{Mouse}$$

and

$$Age(\texttt{Computer}, t) = \texttt{Computer}$$

so we can derive the time transition

$$(\ \texttt{Mouse} \mid \texttt{Computer}\ ) \xrightarrow{(t)} (\ \texttt{Mouse} \mid \texttt{Computer}\ )$$

as well as the action transition

$$(\ \texttt{Mouse} \mid \texttt{Computer}\ ) \xrightarrow{click?}$$
$$(\ [\texttt{0.0025}](\texttt{click?.double!.Mouse})[\texttt{0.249} > \texttt{single!.Mouse} \mid \texttt{Computer}\ )$$

If the mouse button is not pressed again, the subsequent time transitions of the mouse process are

$$[\texttt{0.0025}](\texttt{click?.double!.Mouse})[\texttt{0.249} > \texttt{single!.Mouse} \xrightarrow{(0.0025)}$$
$$(\texttt{click?.double!.Mouse})[\texttt{0.249} > \texttt{single!.Mouse} \xrightarrow{(0.249)}$$
$$\texttt{single!.Mouse}$$

For these first 0.2515 seconds, only the `click?` action is available, which is not internally connected, so we can derive the system time transition

$$(\ [\texttt{0.0025}](\texttt{click?.double!.Mouse})[\texttt{0.249} > \texttt{single!.Mouse} \mid \texttt{Computer}\ ) \xrightarrow{(0.2515)}$$
$$(\ \texttt{single!.Mouse} \mid \texttt{Computer}\ )$$

At this point an internal communication between `single!` and `one?` becomes available, so no more time transitions can take place until the following transition has taken place

$$
\begin{array}{l}
(\ \texttt{single!.Mouse} \mid \texttt{Computer}\ ) \xrightarrow{\tau} \\
(\ [0.0018](\texttt{click?.(click?.double!.Mouse)}[0.245, 0.255 > \texttt{single!.Mouse}) \mid \\
\quad [0.0012][0.41](\texttt{one?.}[0.4, 0.5]\texttt{Computer}\ +\ \texttt{two?.}[1.2, 1.4]\texttt{Computer}\ )
\end{array}
$$

The computer can will now execute for 0.41 seconds after its communication delay, so that after a time transition of 0.4112 we are back to the starting point of `( Mouse | Computer )`

6 Implementing AORTA Designs

One of the motivations for this work is to provide a route for building verified real-time systems from specification to implementation. Having put aside timed process algebras as broad spectrum languages, we have chosen to use timed logic for specification, and AORTA as a design language; it remains to show how AORTA designs can be implemented, and how these implementations can be verified to match their designs. The implementation described in this section is not meant as the only way to implement AORTA designs, but rather to show that the kind of performance modelled by AORTA can be achieved by real systems.

The most challenging part of implementing a timed algebra lies in the timing aspects, and it is on these aspects that we concentrate. Perhaps the most important thing to achieve in a real-time system is predictability: if timing requirements are to be as important as the functional requirements we have to give them equal status, and no one would be happy with a database which 'usually' keeps its integrity, or a structural analyser that works 'provided you don't give it too much data at once'. If enough assumptions cannot safely be made about an environment to guarantee that a system will function safely, then that system is not safe to use in such an environment. There are those who prefer to allow some possibility of failure within a system and justify that failure is so unlikely as to render the system effectively safe, but for two reasons we prefer to stick to a more predictable approach:

- Software, at least, does not wear out or fail at random, so there is no reason why this part of a system should not be predictable.
- The full stochastic analysis of a system is very expensive, and relies on assumptions about independence of events which cannot be verified. (In particular, the behaviour of a system in extremal circumstances is of most interest, and this is when assumptions may be least valid.)

Admittedly, there are elements in a system which are unpredictable, such as hardware failure or noisy communication, but we prefer to account for these separately and do most of our analysis on a predictable basis.

Having said all this, we return to our earlier point that precise times can seldom be predicted for a system, and that bounds are much easier to achieve. Note that we claim predictability is important, not determinism: a system whose

performance is bounded is predictable, so we present a scheme for producing predictable systems in the absence of hardware failures.

There are three areas where an implementation of an AORTA design has time bounds to achieve: in sequential execution, in communication, and in timeouts. As far as sequential execution goes, we assume that bounds can be placed on the processing time required for a piece of sequential code (see [32, 33, 34]), so if the parallelism within a system is to be implemented solely by distributing the processing there is no more to do as processing time is the same as elapsed time; the situation is more interesting when multitasking on a single processor is to be used, and processing time for a process is different from elapsed time.

The scheduler described here is very predictable, if not the most efficient under light loads. It should be noted, though, that it is under heavy processing loads that prediction becomes most important, and this scheduler has the pleasing property that the more processing required the more efficient it becomes. The basis is a very simple round-robin scheduler which switches processes at a fixed rate, regardless of their state of execution. This makes the performance of each process independent of the others, (as opposed to the situation in a priority based scheduler,) except when dependence is explicitly introduced by waiting for communication, and in a way makes each process look like it is being executed on a separate processor. In a system with a fixed number of processes (such as AORTA) the time between schedules for any process is fixed, as is the amount of time it gets. Figure 7 shows the schedule for two processes: the l labels refer to the length per schedule, and d to the time (or distance) between calls, with the scheduler being process 0. A more sophisticated scheduling mechanism may be more efficient, but is likely to be much less predictable. The arguments as to which scheduling policy is 'best' are long and not necessarily enlightening here — for the moment we only want a scheduler that 'will do'. For discussion of some of the issues see [35, 36], although their conclusions are not necessarily in line with ours.

In general, if process i needs t units of processing time to complete a task, bounds can be put on the amount of real time needed to complete. The minimum elapsed time is required if measurement starts at exactly the point at which the process becomes scheduled (as this minimises the amount of time spent waiting). In this case the amount of time spent waiting is

$$\lfloor t/l(i) \rfloor \times (d(i) - l(i))$$

(where $\lfloor x \rfloor$ is the largest integer such that $\lfloor x \rfloor \leq x$), so the total elapsed (real) time is

$$t + \lfloor t/l(i) \rfloor \times (d(i) - l(i))$$

In the worst case time is measured from the point at which the process has just become descheduled, in which case there is a whole extra idling cycle, giving an elapsed time of

$$t + (\lfloor t/l(i) \rfloor + 1) \times (d(i) - l(i))$$

Putting these bounds together with the bounds on processing time for our sequential processes we can arrive at bounds on elapsed time for the execution for a section of code running in a process.

As well as allowing each of the processes enough processing time to complete their tasks within the time bounds required, there also has to be a way of implementing the communication or synchronisation between processes. Many timed process algebras enforce the maximum progress principle (τ urgency), where

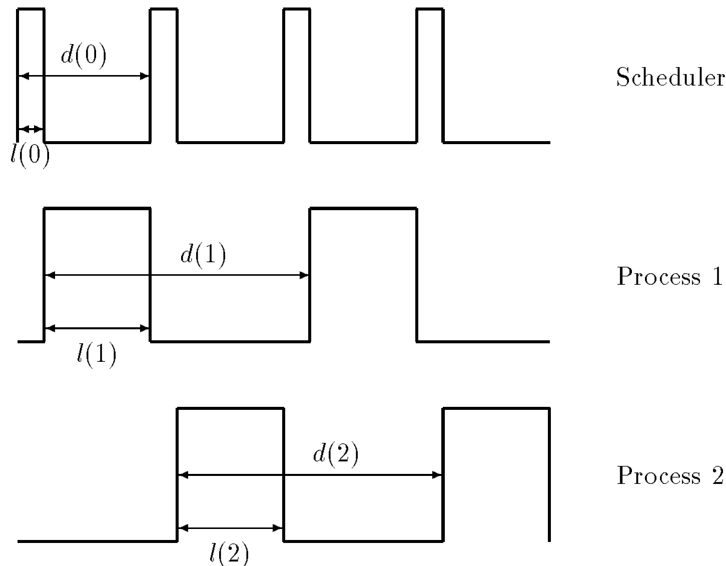

Figure 7: Predictable process scheduling

communication between processes happens as soon as both parties are ready, as this makes the semantics simpler, and ensures that things like timeouts can be modelled (if communication may stall indefinitely a timer becomes useless). In AORTA immediate communication is not necessary, as it is virtually impossible to implement, but a time bound is put on communication by the delay in the rule for action transitions of sequential processes, shown in figure 3.

This can be implemented relatively easily, especially with explicit connection of gates. An area of memory shared by all of the processes and the scheduler is used, with a region for each communication channel (i.e. a linked pair of gates). Inside each such region there should be a bit for each gate, corresponding to readiness to communicate. These are usually both set to zero, but when a process wishes to communicate it sets the relevant bit to one, and waits for it to return to zero before continuing. The scheduler is the only process with the power to reset a bit, and each time it is called it runs through all of the channels and resets to zero any pair of ones. In this way there is an upper bound on the amount of time before an enabled communication occurs, essentially the length of the time slice $d(0)$. Note that values can also be passed by simply having a value store associated with each communication channel, written to by one of the processes and read by the other on completion of the communication. Choice does complicate the issue, as some communication possibilities must be removed when a choice is resolved, and this leads to some nondeterminism in the amount of time the scheduler takes to complete its communication check; again this can be bounded, so the system remains predictable. Because the act of setting a bit can be atomic (in terms of CPU instructions) and all of the resetting is done by the scheduler which cannot be interrupted, no problems arise from preemption of processes.

Timeouts in AORTA are merely an extension of choice, and can be implemented as such. When a choice with a timeout is started the local clock can be read and the time at which the timeout will expire can be calculated. If

this time is stored somewhere that the scheduler can see it, then the scheduler can compare any timeouts with the current time and adjust the relevant process control accordingly, as well as removing any redundant choices from an activated timeout. As before, bounds can be placed on the time at which any particular timeout will expire, but exact figures cannot necessarily be given.

This general scheme then allows bounds to be placed on sequential execution times, communication delays and timeout events so that an implementation can be shown to have the same timing characteristics of an AORTA design. If the design is also shown to correct with respect to a timed logic specification then we can can be satisfied that the implementation is also correct with respect to that specification. For a more detailed timing analysis of this kernel, and a description of some of the implementation details, see [37].

7 Conclusion

In this paper we have argued for and presented a timed process algebra which is amenable to formal verification and yet can only represent systems which can be implemented; some indication as to how these systems might be realised in practice is also given. AORTA is certainly not the first timed process algebra (the introduction references many others), or the first attempt to design timing predictability into a system from the start [38], or the first attempt to provide a (formally based) middle ground between implementation and specification [39]: the novelty of this approach lies rather in doing all of these things at once.

Further possible work lies in several directions. From an implementation point of view, investigation into more advanced scheduling techniques, and distributed implementations would be useful. Work also needs to be done on methods for verifying that AORTA designs satisfy timed logic predicates (i.e. formal specifications), as well as extending (perhaps annotating) AORTA designs to include information about data within the system. This would allow for functional as well as temporal verification, and the resolution of some of the nondeterminism in the system. Finally, tool support is crucial, and there are many areas which would gain from computer assistance. Although some tools already exist, including a simulator and code generator [37], work needs to be done on verification of designs, and calculation of time bounds from the distribution and scheduling characteristics of the system.

Acknowledgements

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

References

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

[2] C A R Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[3] International Standards Organisation. *Informations processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, volume ISO 8807. ISO, 1989-02-15 edition, 1989.

[4] 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.

[5] J C M Baeten and J A Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[6] T Bolognesi and F Lucidi. LOTOS-like process algebras with urgent or timed interactions. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91, Sydney*, pages 249–264. Elsevier, November 1991.

[7] L Chen. An interleaving model for real-time systems. Technical Report ECS-LFCS-91-184, Edinburgh University, November 1991.

[8] R Gerber and I Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.

[9] M Daniels. Modelling real-time behavior with an interval time calculus. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 53–71. Springer-Verlag, 1992.

[10] H Hansson. A calculus for communicating systems with time and probabilities. In *11th real-time systems symposium, Lake Buena Vista*, pages 278–287. IEEE, 1990.

[11] P Krishnan. A model for real-time systems. In *16th International Symposium on Foundations of Computer Science, Kazimierz*, pages 298–307, 1991.

[12] G Leduc. An upward compatible timed extension to LOTOS. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91, Sydney*. Elsevier, November 1991.

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

[14] J Quemada and A Fernandez. Introduction of quantitative relative time into LOTOS. In H Rudin and C H West, editors, *Protocol Specification, Testing and Verification VII, Zurich*, pages 105–121. Elsevier, 1987.

[15] 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.

[16] W Yi. Real-time behaviour of asynchronous agents. In *CONCUR '90, Amsterdam, Lecture Notes in Computer Science 458*, pages 502–520. Springer-Verlag, 1990.

[17] X Nicollin and J Sifakis. An overview and synthesis on timed process algebras. 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 526–548. Springer-Verlag, 1991.

[18] J S Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18(1):33–60, April 1992.

[19] J M Spivey. *The Z notation: A reference manual.* Prentice Hall, New York, 1989.

[20] J S Ostroff. A verifier for real-time properties. *Real-Time Systems*, 4(1):5–36, March 1992.

[21] 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.

[22] L Chen and A Munro. Applications of modal logic for the specification of real-time systems. In J C P Woodcock and P G Larsen, editors, *Formal Methods Europe '93, Odense, Lecture Notes in Computer Science 670*, pages 235–249. Springer-Verlag, April 1993.

[23] E A Emerson, A K Mok, A P Sistla, and J Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, December 1992.

[24] E Harel, O Lichtenstein, and A Pnueli. Explicit clock temporal logic. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 402–413, June 1990.

[25] J Hooman. Compositional verification of real-time systems using extended Hoare triples. 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 252–290. Springer-Verlag, June 1991.

[26] G Jones. *Programming in occam.* Prentice Hall, New York, 1987.

[27] R Sisto, L Ciminiera, and A Valenzano. A protocol for multirendezvous of LOTOS processes. *IEEE transactions on computers*, 40(1):437–446, April 1991.

[28] D J Hatley and I A Pirbhai. *Strategies for Real-Time System Specification.* Dorset House, New York, 1988.

[29] S Bradley, W Henderson, D Kendall, and A Robson. Practical formal development of real-time systems. In *11th IEEE Workshop on Real-Time Operating Systems and Software, RTOSS '94, Seattle*, pages 44–48, May 1994.

[30] J F Groote. Transition system specifications with negative premises. In J C M Baeten and J W Klop, editors, *CONCUR '90, Amsterdam, Lecture Notes in Computer Science 458*, pages 332–341. Springer-Verlag, 1990.

[31] S Bradley, W Henderson, D Kendall, and A Robson. Designing and implementing correct real-time systems. In W-P de Roever, editor, *Formal Techniques for Real-Time and Fault-Tolerant Systems FTRTFT '94, Lubeck(To appear)*. Springer-Verlag, September 1994.

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

[33] 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.

[34] P Puschner and Ch Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1:159–176, 1989.

[35] A Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3):116–128, May 1991.

[36] C Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–52, March 1992.

[37] S Bradley, W Henderson, D Kendall, and A Robson. A formally based hard real-time kernel. Technical Report NPC-TRS-94-3, Department of Computing, University of Northumbria, UK, 1994. To appear in November 1994 edition of Microprocessors and Microsystems, special issue on hard real-time kernels.

[38] H Kopetz, A Damm, C Koza, M Mulazzani, W Swabl, C Senft, and R Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[39] M Kooij. Linking specifications with implementations. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91, Sydney*, pages 99–108. Elsevier, November 1991.