Designing and Implementing Correct Real-Time Systems

Steven Bradley, William Henderson, David Kendall, Adrian Robson

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

Abstract. Existing formal methods for real-time largely deal with abstract models of real-time systems, and seldom address implementation issues; they are mainly used for modelling and specification. In this paper we propose an alternative approach, in which a new timed process algebra, AORTA, is used as a design language, which can be verifiably implemented. As well as introducing and formally defining the language, methods for implementation and verification are discussed.

1 Introduction

Much research effort is devoted to real-time systems: theoreticians propose models for real-time systems and techniques to verify them, whilst operating systems designers and hardware manufacturers strive to provide yet better performance from execution platforms. Links between high-level theory and concrete systems are scarce and usually tenuous, and yet it is vital for theoreticians to apply their techniques to concrete systems for their results to be of relevance, and for implementors to see beyond low-level details to be able to provide performance guarantees about complete systems. The work we describe here is an attempt to bridge the gap between formal analysis techniques and implementation methods, by providing a design language with a formal semantics which is verifiably and semi-automatically implementable. By using our process algebra, AORTA (Application Oriented Real-Time Algebra), as a design language, systems can be analysed before they are built, and the systems described can be constructed to be correct, rather than shown to be correct by post hoc analysis.

There are many timed process algebras other than AORTA, some of which are reviewed in [27] and [29], including variants of Timed CCS [10, 11, 18, 22, 40, 43], Timed CSP [35], Timed LOTOS [5, 6, 23, 25, 34], Timed ACP [2, 13], and others such as ATP [28] and CCSR [14]. In Sect. 2.1 we will argue that these algebras are too expressive to be implementable in general. The implementation of untimed process algebras has been considered, for example in [16, 41, 42], but we do not know of any other attempts at direct implementation of a timed process algebra. Similarly, so far as we are aware, other timed formalisms, such as Time Petri Nets [4] and Modecharts [19], TAM [36] and CRSMs [37] have not been used for implementation. Timed logics appear to be most useful for higher-level specification rather than realisation, notwithstanding work done on executable temporal logics for prototyping [3, 15, 26].

In this paper we introduce our design language, AORTA in Sect. 2, and then go on to discuss how AORTA designs can be implemented in Sect. 3. The verification of implementations is addressed in Sect. 4, and Sect. 5 describes how the work fits in to an overall design method, and the directions for further work. Finally, Sect. 6 presents our conclusions.

2 An Application-Oriented Real-Time Algebra

2.1 Why A New Timed Process Algebra?

Untimed process algebras can be used as wide-spectrum languages, i.e. they can be used for specification, design, and to some extent implementation, using bisimulation as a proof technique to show the equivalence of different levels of abstraction. Although timed bisimulations can be defined, the level of detail of the model makes the equivalence relation derived from a timed bisimulation a very fine one; too fine, it would appear, to be useful, because although many people have defined timed bisimulation, no realistic examples exist (see also [29]). It has often been suggested that preorders, relating different levels of abstraction within the same basic language, should be used to allow refinement in the design process. Again, some untimed preorders do exist, but we have not come across any useful timed preorders which deal with concurrency. As many timed process algebras are based heavily on untimed algebras, they have many of the structures which make untimed algebras suitable as wide-spectrum languages, without providing useful proof techniques to go with them. Rather than try to use a timed process algebra as a wide-spectrum language, we have chosen to restrict the algebra to a design language, and to use other techniques, such as model-checking, for verifying correctness between levels of abstraction.

Because existing timed process algebras are meant to be used as specification and modelling languages they are required to be expressive, but the expressivity required precludes implementation in general. For example, most of the languages which can handle concurrency allow dynamic process creation via parallel composition, and it becomes very difficult to verify timing in a system where processes can be created or destroyed. Also, the languages provide a very detailed model of the timing of events within a system, the accuracy of which is very difficult to provide in implementation — giving delays as exact figures, rather than bounds, makes the construction of systems which have that behaviour impractical. Most of the novel features of our language arise from implementation considerations. Two such features are those just mentioned: the number of processes in a system is statically defined, and computation and communication times can be bounded rather than given exactly (the bounds given in languages such as [10, 11, 34] refer to the times at which actions become available and unavailable, and are not able to introduce the nondeterminism associated with computation times). The other major feature of AORTA is that communication between processes can only occur along predefined routes, to aid implementation and to reduce the complexity of the verification problem. Timed process

algebras can be categorised by the time model that they use, some arguing that a dense time model makes verification too complex, and some that a discrete time model does not accurately describe system behaviour. To try and get the best of both worlds, AORTA can use a discrete or a dense time domain (but not both at once).

AORTA is described in more detail in the following subsections, by first giving the concrete syntax and informal semantics, then a small example, before defining the abstract syntax and giving the formal semantics in terms of a timed transition system definition.

2.2 Concrete Syntax and Informal Semantics

AORTA is similar to CCS, both in notation (`.` for action prefixing and `+` for choice) and in semantics (only two-way synchronisation allowed), but is both more expressive, in that timing details can be included explicitly, and more restrictive, to allow for implementation. One of the restrictions placed on the language is that the number of processes in a system may not vary, and this restriction is enforced by insisting that all parallel composition should appear 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 3. As AORTA is in some ways similar to CCS, some familiarity with CCS is assumed in the following.

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 such as 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 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 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 run-time data — either we 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 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.

Parallel Composition and Communication. Apart from fixing the number of processes in a system in order to provide reliable timing predictions (see section 3), there are other steps which can be taken to aid implementability. One area which is crucial to process algebras and real-time systems is interprocess communication, and this is perhaps where AORTA is most different from other process algebras.

In all of the common process algebras the communication actions of any pro-

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`
nondeterministic choice	`S1 ++ S2`
recursion	equational definition

Table 1. Summary of AORTA concrete syntax

cess 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 [21] and the need for a special protocol in LOTOS [38]. For a simple system, which is all we can hope to formally verify at the moment, the mechanism for managing 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.

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 pairs of gates linked, using a connection set written in angle brackets after the processes. The use of AORTA is now illustrated with a small example.

2.3 An Example: A Temperature Conversion Process

In order to demonstrate the language constructs in action, this section develops an AORTA description of a temperature conversion process, which may be used as a component of a plant control system. A first attempt at a conversion process simply takes input on one gate, computes the new format, which takes between 0.1 and 0.15 seconds, and then outputs this new data before restarting. It looks like this:

```
Convert = in.[0.1,0.15]out.Convert
```

A more sophisticated process might have different conversion modes which it can use, which are changed by another controlling process. This would be represented by

```
Convert = in.[0.1,0.15]out.Convert
          +
          mode.[0.3,0.4]Convert
```

where a piece of reconfiguration code, which takes between 0.3 and 0.4 seconds, will execute if communication takes place on the `mode` gate.

In order to ensure that output data is up to date, the process may wish to time out on the `out` communication, to reread and recompute the output value. A timeout of about 1.5 seconds could be added:

```
Convert = in.[0.1,0.15]
           (out.Convert)[1.45,1.55>Convert
          +
          mode.[0.3,0.4]Convert
```

Note the use of brackets to indicate that the timeout takes place over the `out` communication, rather than `in`.

Finally, the process may wish to check the bounds of its input value, and send a warning signal to another process if it lies above a certain threshold. This is written using a nondeterministic (data-dependent) choice, where the right branch should be chosen if there is a problem with the input value.

```
Convert = in.(Convert2 ++ warning.Convert2)
          +
          mode.[0.3,0.4]Convert
Convert2 = [0.1,0.15]
            (out.Convert)[1.45,1.55>Convert
```

A plant control system incorporating the `Convert` process with a `Control` process and a `Datalogger` process is shown in Fig. 1, and may written as follows:

```
( Control | Convert | Datalogger )
<(Control.changem,Convert.mode),
 (Control.temphigh,Convert.warning),
```

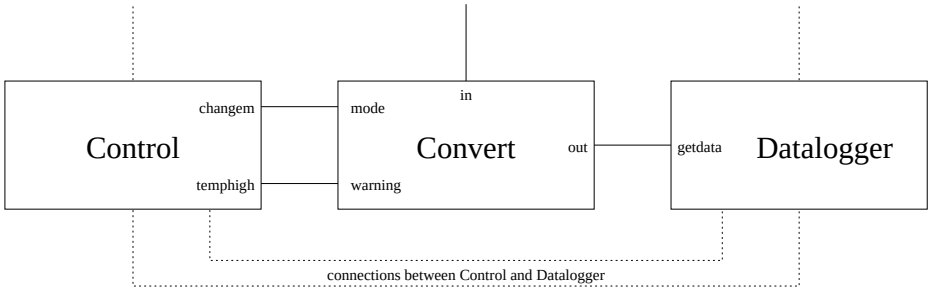

Fig. 1. A Plant Control System

```
 (Convert.out,Datalogger.getdata),
    (* connections between Control and Datalogger *)
>
```

Here the `Control` process may request a change of conversion mode via its `changem` gate, and notes any warning signals on its `temphigh` gate. The output of `Convert` is sent to `Datalogger`, for possible future analysis. A description of this system is also given in [8]. Common solutions to real-time problems can be expressed in AORTA, in particular, resource contention can be handled with choice, and polling loops can be implemented with timeouts. More involved examples have been developed using AORTA, including a car cruise controller.

2.4 The Abstract Syntax of AORTA

In the abstract syntax of AORTA, 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. 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

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

where t, t_1 and 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 (a combination of action prefixing and choice), 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 Fig. 2. Informally, regular expressions are those ex-

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

Fig. 2. Definition of *regular*

pressions 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. For reasons already stated, we do not wish to deal with bisimulations, but 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.

2.5 The Formal Semantics

In order to define the semantics of system expressions, the semantics of regular sequential 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 can be resolved into a regular expression. Transition 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. Later on we will use a distinguished action transition $\xrightarrow{\tau}$ to represent internal communication. The rules are given in Fig. 3, and the auxiliary function $Poss$ is defined in Fig. 4.

When an action transition takes place, a communication delay is prefixed to the process, and all nondeterminism up to the next action is resolved, in accordance with the $Poss$ function. This is not only convenient from a theoretical standpoint, but does correspond to the situation in a real system. 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 Sect. 3 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 specifications are discussed in [17], 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 [17].

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,

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

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

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S \xrightarrow{a_j} S'_j} \; j \in I, S'_j \in Poss(S_j)$$

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

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

Fig. 3. Transition rules for regular expressions

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

Fig. 4. Definition of $Poss$

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 Fig. 6. This function is really meant

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

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

Delay

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

Fig. 5. Transition rules for system expressions

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

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

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

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

Fig. 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 [27].

3 Implementation of AORTA designs

As the most important feature of AORTA is its implementability, we describe in this section work done on implementing AORTA designs. The aim of this part of the work is to provide semi-automatic procedures for producing concrete systems from AORTA designs. We have chosen to implement AORTA designs as software processes multitasking on a single processor, as this is probably the most common solution for small embedded systems, but the process abstraction can model distributed processing, and hardware processes. The software processes are generated from an annotated AORTA design, compiled and statically analysed. When these processes are executed with a dedicated predictable kernel [7], the overall system performance can be analysed, and checked against a high-level formal specification.

Having fixed on a multitasking solution, the next decisions to be made are what kind of scheduling policy should be used, and how to implement message passing between processes; these are the two facets that are most important in a real-time kernel. There are very many scheduling policies available (see [9] for a review), many of which use quite sophisticated techniques to extract maximum performance from the available hardware. For the moment we are content with predictability of performance, rather than good average performance, so we have chosen to use a fixed time slicing round-robin scheduler, as this guarantees that each process will get processing time in all circumstances (even if it doesn't necessarily need it), eliminating the interdependence of processes except where made explicit in a communication request. In adopting this approach we are sacrificing some efficiency for a reduction in the effort required to predict the performance of the system. As hardware costs are relatively small compared with

software development and verification costs, we feel this tradeoff is justified in many cases. If more efficiency is required, then a more complex scheduling policy could be used, but the effort required to predict performance would be higher.

Apart from a scheduling mechanism, a kernel which is to be used to implement AORTA designs must also be able to handle inter-process communication, timeouts, and external communication in a predictable way. Upper bounds can be placed on communication and timeout delays by checking for these events at each reschedule point. Resolving choice between communication branches can also pose problems if it is not managed by a central arbiter, so by having the kernel deal with this at reschedule time many difficulties are alleviated. (It is this matter of resolving choice which poses the largest practical problem to implementing designs on a distributed system.)

The software processes which are executed by the kernel are generated from the design, using kernel calls to set up internal communications, timeouts and external communications. Pieces of sequential code which are to be executed during a computation delay are given as annotations to the design. These annotations are treated as comments as far as the formal semantics of the language is concerned, but are vital in constructing a working system. Annotations are also used to specify branching conditions for data-dependent choices, and to control data transfer in communications. Currently, the software processes are generated in C, for purely pragmatic reasons, such as the availability of cross-compilers, and the existence of code timing analysis tools. Any language which is amenable to best- and worst-case timing analysis could be used in principle.

A 68000 based kernel providing this scheduling mechanism and offering primitives for communication and timeouts has been written, and can be used in conjunction with software processes which are automatically generated from an annotated design. The details of the kernel, including a timing analysis of computation and communication, is given in [7]. Having considered the implementation AORTA designs, we now go on to consider the problem of verifying the correctness of that implementation.

4 Verification Techniques for AORTA Designs

The general problem of design and verification of real-time systems is a difficult one, and although much research has gone into various aspects of the problem, little has been done to link high-level formal considerations, with low-level implementation details. To address this problem we have presented a language which tries to bridge the gap, providing an implementable design language which has a formal semantics, and hence can be mathematically reasoned about at a high level. The advantage with this approach is that once verified implementation mechanisms have been put in place, high-level formal reasoning can be brought to bear on actual implementations in a tractable way. AORTA provides a formal virtual machine for real-time systems, in which the domain of interpretation is purely temporal: AORTA deals with the time of occurrence of communications (internal and external) and not data. In this respect we can

provide formal verification of a system which uses C code as we can give guarantees about its time behaviour. For the moment, we address the problem of verifying the provided implementation mechanisms, as there already exists work in the literature which can be used for verifying the correctness of process algebra terms with respect to high-level specifications, and in particular on timed model-checking [1, 12, 24, 28, 30].

From the scheduler, guarantees can be given about upper and lower bounds on the amount of processing time each process has per unit of elapsed time. Combining this information with upper and lower bounds on the processing time for a piece of sequential code (such as are provided by [31, 32, 33]) can give upper and lower bounds on real execution time for a sequential computation, so verifying the bounds given in an AORTA design for a computation delay. Similarly, as the kernel checks for internal communications, timeouts, and external communications at regular intervals, upper bounds can be placed on communication and timeout delays — so verifying the figures given for the *delays* function and the bounds on timeouts. For further details of the detailed timing analysis of the kernel, see [7].

5 A Top-to-Bottom Formal Design Method

Having discussed implementation and verification in the previous two sections, we now look at how the work fits together into a design method. The overall aim of the project is to provide a verifiable route from a formal specification to an implementation, where time can be reasoned about at all levels, and Fig. 7 shows the extent to which we have achieved this, and how further work fits in with that already completed. Solid arrows on the figure indicate routes that are available (automated where appropriate), and dashed lines those which are not yet available or have not been automated. In general, arrows going down are concerned with designing or implementing, and are matched by upward arrows which represent the corresponding verification.

From the figure we can see that implementation techniques for AORTA designs are quite well developed: a kernel has been written and analysed, giving predictable performance of processing and communication; code generation (into C) for each of the processes automates the construction of choices and timeouts, and allows pre-written functional code to be included into the relevant sections. Verification of implementations methods relys on the timing analysis of the kernel described in Sect. 3 and in [7] and the code timing techniques described in [31, 32]. Further work on implementation will try to extend the multi-tasking round-robin scheduling approach described in Sect. 3, by looking at distributed solutions, alternative scheduling strategies, and will possibly examine implementation of some of the processes in hardware, allowing verifiable hardware/software co-design. One important piece of verification that remains to be done is of the functional correctness of the kernel's communication mechanism. The problems of verifying the functional behaviour of sequential code may be addressed by using a formally defined real-time language. Finally, as far as

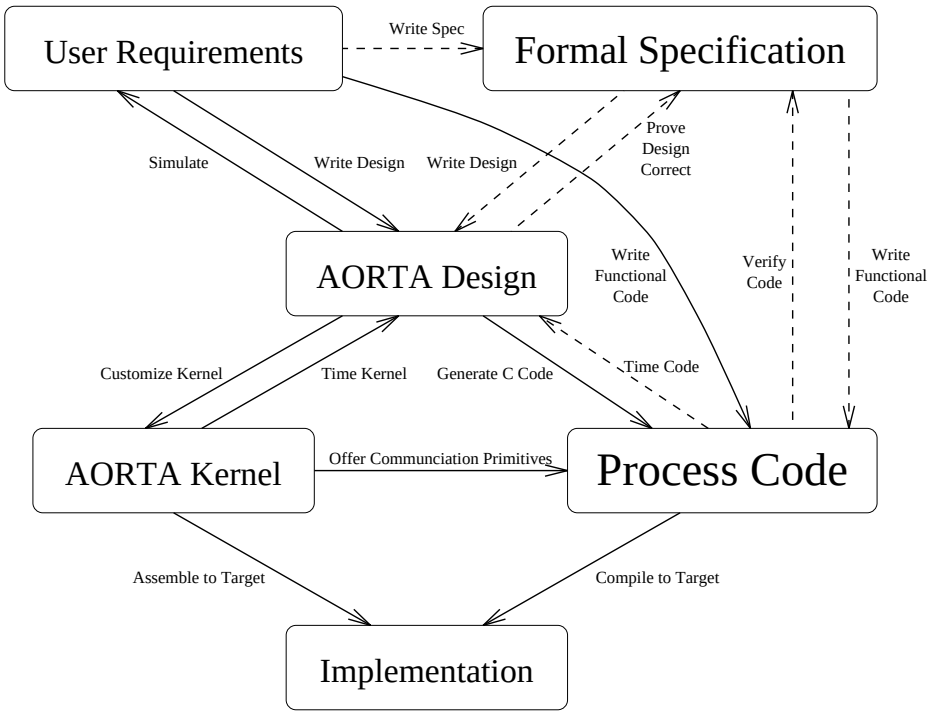

Fig. 7. AORTA Within a Design Method

implementation is concerned, a C code timing tool must be integrated into the verification system to deliver the required guarantees of performance.

Algorithms for verifying the correctness of a formally defined system with respect to a formal specification written in a timed temporal logic are relatively common [1, 12, 24, 30], so we are hoping to use some of these results to provide automatic verification of AORTA designs with respect to timed logic specifications. A translation of AORTA designs into the timed graphs of [1] exists (although it has yet to be proven correct), which could provide the basis for an automatic model-checking tool. As well as a model-checking approach, a mechanical proof-checking approach may also be of interest, given the complexity of the model-checking problem, and the size of the state-space in a concurrent system. A synthesis of the two approaches might be used, with a proof-checker used in the higher level verification, using a model-checker once the problem has been reduced in size, perhaps performing model-checking only on a single process. This last suggestion is seriously hampered, however, by the problem of representation of real numbers within mechanical proof checkers, so is of a lower priority.

Currently there are no techniques for verifying the correctness of an AORTA design with respect to a high-level specification, but a simulator has been written which allows designs to be informally checked against their requirements. Before

systems can be verified as correct, they must actually be correct, so in order to avoid wasting time in trying to prove untrue results, the simulator can be used to test out a design informally before applying the rigour of proof. A simulator is also useful where a system has to be informally assessed by someone inexpert in formal methods, such as the end user of the system.

Functional aspects of system design are not currently considered in AORTA, but this is clearly an important area for future research. At the moment, in order to include functional code, the code generator uses AORTA designs with sections of hand-written C code attached in the relevant places, but these annotations could be used to not only provide the code, but also to specify the variables and computations concerned. It may be possible to adapt techniques such as Z [39] or VDM [20] for this purpose, but the introduction of concurrency into these techniques has proved difficult in the past; also if such an approach is to be adopted, a language other than C must be used.

6 Conclusions

In this paper we have presented a process algebra, AORTA, which has been developed specifically for use as a design language. The restrictions and extra expressivity of AORTA, as compared to other timed process algebras, make it possible to implement time-critical systems and to verify them; tools have been developed to semi-automatically implement designs written in AORTA. The formal semantics of AORTA make high-level formal reasoning about designs possible in principle. Combining implementability with high-level reasoning allows a hard real-time system to be developed and verified from specification to implementation.

Acknowledgements

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

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*, pages 414–425, June 1990.
2. J C M Baeten and J A Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.
3. H Barringer, M Fisher, D Gabbay, G Gough, and R Owens. Metatem: A framework for programming in temporal logic. Technical Report Series UMCS-89-10-4, Department of Computer Science, University of Manchester, Oxford Rd, Manchester, October 1989.

4. B Berthomieu and M Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):199–273, March 1991.
5. 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*, pages 249–264. North Holland, November 1991.
6. T Bolognesi, F Lucidi, and S Trigila. From timed Petri nets to timed LOTOS. In L Logrippo, R L Probert, and H Ural, editors, *Protocol Specification, Testing and Verification X*, pages 395–408. North-Holland, 1990.
7. 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.
8. 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*, pages 44–48, May 1994.
9. A Burns. Scheduling hard real-time systems : a review. *Software Engineering Journal*, 6(3):116–128, May 1991.
10. L Chen. An interleaving model for real-time systems. Technical Report ECS-LFCS-91-184, Edinburgh University, November 1991.
11. 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, Lecture Notes in Computer Science 571*, pages 53–71. Springer-Verlag, 1992.
12. E A Emerson, A K Mok, A P Sistla, and J Srinivasan. Quantitative temporal reasoning. *Real-Time Systems*, 4(4):331–352, December 1992.
13. M Felder, C Ghezzi, and M Pezze. High-level timed Petri nets as a kernel for executable specifications. *Real Time Systems*, 5(2/3):235–248, May 1993.
14. 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.
15. C Ghezzi, D Mandrioli, and A Morzenti. Trio: A logic language for executable specifications of real-time systems. Technical Report 89-006, Politecnico di Milano, 1989.
16. D Gilbert. Executable LOTOS. In Rudin and West, editors, *Protocol Specification, Testing and Verification VII*, pages 281–294. North-Holland, North-Holland, 1987.
17. J F Groote. Transition system specifications with negative premises. In J C M Baeten and J W Klop, editors, *CONCUR '90, Lecture Notes in Computer Science 458*, pages 332–341, 1990.
18. H Hansson. A calculus for communicating systems with time and probabilities. In *Proc. 11th real-time systems symposium 1990*, pages 278–287, 1990.
19. F Jahanian, R Lee, and A K Mok. Semantics of modecharts in real time logic. In *Proceedings of 21st Hawaii International conference on system Science*, pages 479–489. IEEE, IEEE Press, 1988.
20. C B Jones. *Systematic software development using VDM.* Prentice-Hall, 1986.
21. G Jones. *Programming in occam.* Prentice Hall, 1987.
22. P Krishnan. A model for real-time systems. In *Proc. Foundations of Computer Science*, pages 298–307, 1991.

23. G Leduc. An upward compatible timed extension to LOTOS. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91*. North Holland, November 1991.
24. H R Lewis. A logic of concrete time intervals. In *IEEE Fifth Annual Symposium On Logic In Computer Science*, pages 380–389, June 1990.
25. A McClenaghan. Mapping time-extended LOTOS to standard LOTOS. In K R Parker and G A Rose, editors, *Formal Description Techniques IV, FORTE '91*, pages 233–248. North Holland, November 1991.
26. B Moszkowski. *Executing Temporal Logic Programs*. C.U.P., 1986.
27. 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. Lecture Notes in Computer Science 600*, pages 526–548. Springer-Verlag, 1991.
28. X Nicollin, J Sifakis, and S Yovine. From ATP to timed graphs and hybrid systems. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Lecture Notes in Computer Science 600*, pages 549–572. Springer-Verlag, June 1991.
29. 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.
30. J S Ostroff. A verifier for real-time properties. *Real-Time Systems*, 4(1):5–36, March 1992.
31. C Y Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, March 1993.
32. C Y Park and A C Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, pages 48–57, May 1991.
33. P Puschner and Ch Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time systems*, 1:159–176, 1989.
34. 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*, pages 105–121. IFIP, North-Holland, 1987.
35. 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), Lecture Notes in Computer Science 600*, pages 640–675. Springer-Verlag, June 1991.
36. D J Scholefield and H S M Zedan. TAM: A formal framework for the development of distributed real-time systems. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Lecture Notes in Computer Science 571*, pages 411–428. Springer-Verlag, 1992.
37. A C Shaw. Communicating real-time machines. *IEEE Transactions on Software Engineering*, 18(9):805–816, September 1992.
38. R Sisto, L Ciminiera, and A Valenzano. A protocol for multirendezvous of LOTOS processes. *IEEE transactions on computers*, 40(1):437–446, April 1991.
39. J M Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.
40. C Tofts. Timed concurrent processes. In *Semantics for Concurrency*, pages 281–294, 1990.
41. I Tvrdy. From LOTOS to OCCAM. In *Second International Conference on Software Engineering for Real Time Systems*, pages 175–179. The Computing and Control Division of the Institution of Electrical Engineers, September 1989.

42. A Valenzano, R Sisto, and L Ciminiera. Rapid prototyping of protocols from LOTOS specifications. *Software — Practice and Experience*, 23(1):31–54, January 1993.
43. W Yi. Real-time behaviour of asynchronous agents. In *CONCUR 90, Lecture Notes in Computer Science 458*, pages 502–520. Springer-Verlag, 1990.

This article was processed using the LaTeX macro package with LLNCS style