A Formally Based Hard Real-Time Kernel

Steven Bradley William Henderson David Kendall
Adrian Robson *

Abstract

In order to demonstrably satisfy hard real-time deadlines, a system must be predictable, and in particular the kernel must be predictable. In this paper we present and analyse a predictable kernel related to AORTA, a formal design language for hard real-time systems. The features of the kernel allow AORTA designs to be verifiably and semi-automatically implemented, and enable verified guarantees to be given about the real-time behaviour of the system.

Keywords: real-time, formal methods, performance prediction, process algebra.

Introduction

The purpose of a hard real-time kernel is to allow concurrent cooperating processes to achieve their specified performance. As the overall aim of a hard real-time kernel is to guarantee performance of the whole system, a simple scheduling mechanism which trades off some efficiency for predictability can be very useful; where hardware costs are relatively small compared with development costs and where guaranteed time performance is critical, such a kernel is of interest. A commoner approach to real-time kernel design is to have a system of priorities (possibly dynamically defined) which ensures that 'important' processes can claim processing time whenever they want it, at the expense of those of lower priority. Unfortunately, high priority processes can often depend on lower priority processes, for instance if they share a critical section of code which may only be accessed by one process at a time; this can lead to a process effectively locking itself out (an extreme consequence of 'priority inversion'). Dynamic assignment of priorities can overcome these problems [1], but gives a kernel whose performance is difficult to predict and analyse in general, and hence difficult to verify.

As well as managing processor allocation, a kernel is expected to deal with inter-process communication and resource management (typically mutual exclusion), and handle these in a way that is demonstrably safe. Once the communications and scheduling arrangements are handled in a predictable way it is then possible to start to predict the behaviour of the whole system in order to show that it satisfies its specification. The complexities of communicating concurrent systems are such that even with a detailed knowledge of the behaviour of all of the individual processes it can be very difficult to form an accurate picture

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

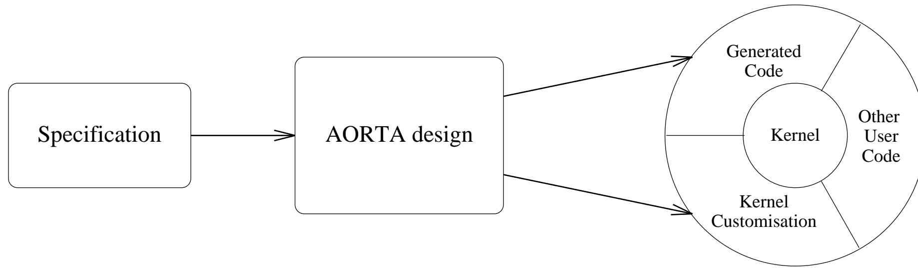

Figure 1: Implementation of an AORTA design using the kernel

of the behaviour of a whole system. Formal methods can be used in such situations, to provide assurance that the picture truly represents the system and that the specification is matched in all instances [2, 3].

If justification is needed for verifying safety-critical systems then we don't have to look very far to find systems whose time performance is crucial: fly-by-wire systems, nuclear and chemical plant controllers, and life support systems all have vital real-time characteristics. Even systems which are not apparently time-critical, but which do have concurrency, can develop faults because of timing. The Therac-25 medical accelerator had, among other problems, timing conditions which caused apparently transient faults, resulting in massive overdoses of radiation [4].

In this paper we present a hard real-time kernel which offers predictable scheduling along with safe communication primitives. It is closely related to AORTA [5], a formal language for hard real-time systems, used to design the timing and communication aspects of a concurrent system, whose language constructs are implemented using the kernel. Having a formal definition of the system via AORTA allows its behaviour to be formally analysed and verified with respect to a formal specification. Through AORTA, a link can be made between the high level reasoning of the theoreticians and the implementation considerations of kernel designers. This link is, unfortunately, not usually made in existing work.

As well as a kernel for AORTA, prototype tools also exist for automatically customising the kernel for an AORTA design, and for generating the framework for the process code (written in C), as shown in figure 1. C was chosen for purely pragmatic reasons, such as availability of timing tools for cross compilers [6], but in principle any language which is amenable to static timing analysis could be used. A simulator tool has also been written, which allows a designer to try out a design before working out the detail of the implementation.

Before considering the kernel in detail, we give an overview of AORTA, to give a specification of the services that must be offered by the kernel. The design and implementation of a kernel which provides such services are described in the following section, after which the performance of the kernel is analysed. In order to indicate how the kernel can be used with AORTA, the next section describes a possible design method before the final, concluding section.

An Overview of AORTA

AORTA (Application Oriented Real-Time Algebra) [5] is a formal design language specifically for concurrent hard real-time systems, related to timed versions of process algebras such as CCS [7], LOTOS [8] and CSP [9]. AORTA deals with the time performance and communication aspects of a system: all sequential computation or other time-consuming activity is represented by the amount of time taken to complete it, making the resulting mathematical model of the system tractable. The few constructs of AORTA are expressive enough to design real systems, and yet restrictive enough to ensure that the system is implementable — it is on the second count that existing timed algebras fall down.

In this section we introduce AORTA by using a semi-realistic example, based on a chemical plant controller. The controller has to monitor and log temperatures within the reaction vessel, and respond to dangerously high temperatures by sounding an alarm. Two rates of sampling must be provided, to be selected by the plant operator, each of which has its own output format for the logging function. In order to ensure safety of the plant, the temperature must be sampled at least every two seconds, and if a reading lies outside the safety threshold the alarm must be sounded. A similar system is described in [10].

An important feature of AORTA is that timing information about a process does not have to be exact, but may be expressed as upper and lower bounds on the time (execution time or communication time). This nondeterminism arises in all multitasking systems, from the scheduler, the communication mechanisms, and sections of sequential code, but it is not addressed in other formal methods which deal with time. The number of constructs for building processes in AORTA is relatively small, and the number of processes within a system is fixed, i.e. there can be no dynamic process creation, allowing straightforward timing analysis, and guaranteeing implementability. Each process has a fixed number of communication gates, each of which may connected either externally or internally. All communication between processes must be between linked pairs of gates (as in figure 2), and is synchronous. When a process wishes to communicate on a certain gate, it has to wait until the process at the other end of the link is also ready before the communication takes place, and the process can proceed. This so-called *blocking* communication mechanism is closely related to that of occam [11], and is similar to the Ada rendezvous [12]. One solution to the plant control problem just mentioned involves two processes, `Convert` and `Datalogger`, with two internal connections and a total of five external connections, as shown in figure 2.

Individual processes are defined using four basic language constructs: computation delay, communication (and its extensions of choice and timeout), recursion, and data dependent choice. A summary of the syntax for sequential processes is presented in table 1. Firstly, all time-consuming computation is represented only by the amount of time it takes: no attempt is made to model the data in the system at this stage of the design, so all details of what happens during this time are omitted. The notation used for this is a pair of square brackets enclosing the time taken, where the time can be given as an exact figure or two numbers separated by a comma, giving the lower and upper bounds on the time (`[t1,t2]S`). The second construct is for blocking communication as described earlier, and is written by simply putting the name of the gate followed by a full stop before the subsequent behaviour definition (`a.S`). Recursion (loop-

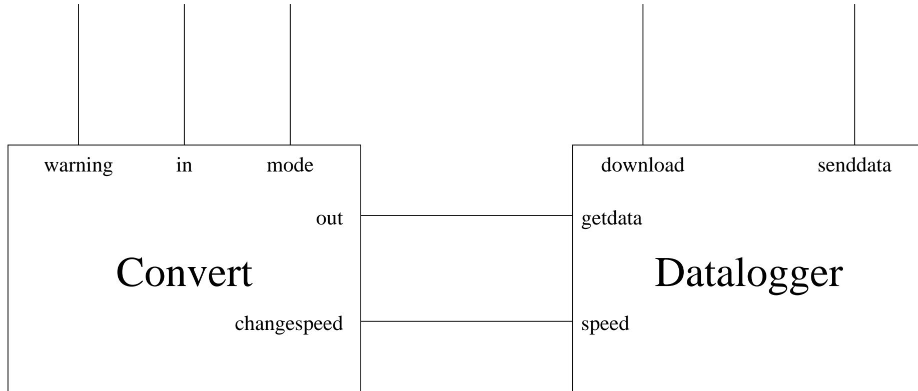

Figure 2: Connectivity of a System Containing Convert

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

Table 1: Summary of AORTA concrete syntax

ing) is the third construct and can be written by using an equational definition of a process and subsequent naming of that process. These three constructs are used in this first attempt at the `Convert` process of our plant controller:

```
Convert = in.[0.001,0.004]out.Convert
```

Here, `Convert` reads its data from gate `in`, and then does the conversion, which takes between 0.001 and 0.004 seconds. Once the conversion is complete, the data is output on gate `out`, and the process loops back to input new data.

The communication construct (`.`) has two extensions available: a choice between offered gates, and timeout continuation if no communication takes place within a certain time. Similar features are offered by Ada and occam, but in AORTA allowances are made for the time taken for communication or timeout to occur. Choices between gates are written using a `+` in between the branches of the choice, and the flow of control of the process depends on which gate is matched first. To allow our conversion process not only to wait for input, but also have the conversion mode changed by the user, a `mode` gate is offered in choice with `in`. If `mode` is matched first then a signal is sent to the `Datalogger` process to tell it to change sampling speed, and a piece of reconfiguration code (recalculation of a look-up table) is executed, which takes between 0.3 and 0.4 seconds. Our `Convert` process would then look like this:

```
Convert = in.[0.001,0.004]out.Convert
          +
          mode.changespeed.[0.3,0.4]Convert
```

As only one branch may be chosen, and the offer of communication on other branches is lost once one is taken up, choice can be used to implement mutual exclusion for access control to a resource of any kind, and in the case of the `Convert` process, it prevents a mode change during a conversion.

Timeouts can be added to a communication or choice between communications, so that if no communication events occur within a certain period of time then control is passed to another branch. As with delays, if timeouts cannot be guaranteed to take place at an exact point in time AORTA allows bounds to be placed on the time taken to time out: timeout will definitely not have occurred before the earliest time, and will definitely have taken place by the latest time if no communication has been possible. To write this down, the communication(s) over which the timeout is taking place is/are bracketed and followed by the bounds (or an exact figure if available) inside `[ .. >`. The behaviour which is to take place in case of a timeout is given after the `>`. For example, to ensure that the `Convert` process is not inadvertently blocked by waiting for communication with `Datalogger`, a timeout of about 1.5 seconds is added to the `out` and `changespeed` communications:

```
Convert = in.[0.001,0.004](out.Convert)[1.5,1.505>Convert
          +
          mode.(changespeed.[0.3,0.4]Convert)[1.5,1.505>Convert
```

Note the use of brackets to indicate which communication the timeout refers to. Used in this way, timeouts can be used to implement watchdogs or polling loops.

Choice by using `+` allows the flow of control of a process to be changed depending on which of several communications takes place. Sometimes choices are not made on the basis of where the process has communicated, but rather on the values that are passed during communication, or on values computed by the process. AORTA views all such branches as data dependent, and as it has no information about the data of the system, the choice appears to be nondeterministic. Such choices are written using the fourth and final construct, called data dependent choice. To define branches which may depend on data, a `++` symbol is placed in between the possible behaviours. In order to send a `warning` alarm for dangerously high readings, our `Convert` process has to use a data dependent choice, which would be based on the value read from gate `in`. The final version of `Convert` then looks like this:

```
Convert = in.(Convert2 ++ warning.Convert2)
           +
           mode.(changespeed.[0.3,0.4]Convert)[1.5,1.505>Convert
Convert2 = [0.001,0.004]
            (out.Convert)[1.5,1.505>Convert
```

Notice the use of `Convert2` within the definition, which is defined along with `Convert`. These definitions are mutually recursive (they refer back and forth to each other), and are used for ease of reading and expression of a single sequential process. As many simultaneous definitions as required may be given, with flow of control passing between any of them by recursion.

Having introduced our language constructs in the development of the `Convert` process, we now use some of them in the definition of the `Datalogger` process.

This process periodically requests data from the `in` gate of `Convert`, and may be requested to change the speed of sampling via the `changespeed` gate. It may also get requests for the data to be downloaded to an external machine. The basic process samples every second (the speed of sampling depends on the `Datalogger` process, as `Convert` will always offer data as soon as it is ready), and in between samples waits for a speed change command, which changes the sampling period to 0.25 seconds:

```
Datalogger = getdata.[0.01,0.015]
                (speed.Datalogger2)
                   [1.00,1.005>Datalogger
Datalogger2 = getdata.[0.001,0.015]
                (speed.Datalogger)
                   [0.25,0.255>Datalogger2
```

In order to add a download facility, a `download` communication is offered in choice with `speed`; if this is requested, then some formatting computation is done (for between 0.5 and 1.0 seconds) before the data is sent out on `senddata`. This final version of `Datalogger` is defined as follows:

```
Datalogger = getdata.[0.01,0.015]
                (speed.Datalogger2
                 +
                 download.[0.5,1.0]senddata.Datalogger)
                 [1.00,1.005>Datalogger
Datalogger2 = getdata.[0.001,0.015]
                (speed.Datalogger
                 +
                 download.[0.5,1.0]senddata.Datalogger2)
                 [0.25,0.255>Datalogger2
```

The four constructs (computation delay, communication, recursion and non-deterministic choice) are used to define individual processes, which can then be connected together into a system by using the parallel operator, written `|`. To allow inter-process communication, pairs of gates may be connected by specifying a connection set in angle brackets after the parallel processes have been listed. Externally connected gates are also listed, and the delays for all communications (which will depend on the scheduling period) are also specified.

The system definition of our plant controller is then written

```
(Convert | Datalogger)
<(Convert.changespeed,Datalogger.speed:0.001,0.003),
 (Convert.out,Datalogger.getdata:0.001,0.003),
 (Convert.in,EXTERNAL:0.001,0.003),
 (Convert.warning,EXTERNAL:0.001,0.003),
 (Convert.mode,EXTERNAL:0.001,0.003),
 (Datalogger.download,EXTERNAL:0.001,0.003),
 (Datalogger.senddata,EXTERNAL:0.5,10.0)>
```

which is the textual representation of figure 2. Notice that all communication delays are the same size, except for `senddata`, as this involves the transfer of a substantial amount of data.

Once a system has been written in this way its behaviour has been precisely defined. AORTA has a formal mathematical definition and semantics, and hence allows mathematical reasoning about the timed behaviour of systems, but we will not touch on these here for lack of space. The interested reader should see [5].

The Kernel Design and Implementation

Many kernels could be used to implement AORTA designs, but the kernel that we present is designed so that it is relatively easy to produce the figures for minimum/maximum execution and communication times. The scheduling mechanism is just about the simplest available, with the processes being preemptively scheduled on a fixed time slice in a fixed order (i.e. no priorities). Inter-process communication is managed by the kernel, and implemented using shared memory. Some round-robin schedulers rely on every process completing its required activity within its allotted slice [13], but we make the scheduling transparent to the process, so that there is no such notion of required activity within a slice, and assume all activity is preemptible. In other words, rather than having tasks executing periodically/synchronously, our processes do not have any fixed period, and achieve all necessary synchronisation via timeouts and communication. If a process has to be excluded from accessing some resource at the same time as another process, this is achieved by using the communication primitives, which makes for a safer and more comprehensible design, and one which is much more easily changed. The important point about having a fixed time slice is that the performance of each process is independent of the behaviour of other processes, except where made explicit at a communication point, so that the processor is effectively split up in a number of smaller processors, one for each process. As well as making performance prediction for each process much simpler, this transparency means that decisions about whether a distributed or multitasking implementation is more appropriate can be deferred until a more appropriate point. Indeed, AORTA processes need not be software processes, but could equally well be built in hardware, allowing formal analysis to be done on hardware/software codesign of a system.

The more interesting part of the kernel is the communication mechanism. Each process may be ready to communicate on any number of gates and may also have a timeout in effect. We require that when a pair of gates in the given connection set are both ready to communicate they must do so within a certain amount of time, and that if any other gates are offered in choice, they must be disabled so that only one gate in the choice may communicate. As any process may be descheduled at any point, careful attention has to be paid to keeping a consistent and complete set of information available to the kernel and the processes, to ensure that the mechanism is sound.

To describe the algorithm used to implement communication, we first of all have to introduce the various flags and pointers associated with each process and with each gate of the system. Let us assume that we have an indexing set I for the processes, and a set J for the gates, and that i and j range over I and J respectively. For each gate j there are two flags: $gate_j.ready$ and $gate_j.has_comm$, which correspond to the readiness of the gate to communicate (i.e. the process to which it belongs has offered it by itself or in a choice), and the successful completion of communication on that gate. There is also an entry $choice_addr_j$ which contains a pointer to a list of the gates in choice with j at

variable	*set*		*reset*	
name	*what*	*when*	*what*	*when*
set_up_i	process i	wishes to comm	kernel	done set up
$choice_list_i$	process i	before set_up_i	N/A	N/A
$gate_j.ready$	kernel	in set up	kernel	after comm
$gate_j.has_comm$	kernel	after comm	process	after noting comm

Table 2: Access to Communication Variables in the Kernel

the current time, and a choice list $choice_list_i$ for each process. Finally, each process has a flag set_up_i which is used to indicate when that process wishes to offer a communication. Figure 3 shows how these flags are changed when a simple choice between two gates is offered.

There are two parts to the algorithm for communication, which effectively run concurrently: the code executed by the kernel on every reschedule, and the code executed by a process when it wishes to offer a communication. In order for a process to offer a communication (with or without choice), it signals to the kernel that it wishes to communicate, and on which gates it wants to communicate. This is done by storing the gate identifiers in $choice_list_i$ (for process i), and then setting the set_up_i flag. This corresponds to figure 3a. During its reschedule, the kernel checks the set_up_i flag for every process, and if it is set, it goes through the corresponding choice list, setting the $gate_j.ready$ flag for every gate it finds there, as a signal to itself that gate j wishes to communicate, as illustrated by figure 3b. After having checked for setups, the kernel looks through the connection set (i.e. the set of pairs of gates that are connected in the design). If it finds a pair in which both gates are ready to communicate (have their $gate_j.ready$ flags set), it resets all of the $gate_j.ready$ flags in the corresponding choice lists (as pointed to by $gate_j.choice_addr$), and sets the $gate_j.has_comm$ flag for each of the communicating gates, as illustrated by figure 3c. The setting of the $gate_j.has_comm$ flags is the signal back to the processes which initiated the communication that it has been successfully completed, and where there is a choice, which of the gates was successful, and it resets the $gate_j.has_comm$ flag before continuing. Table 2 summarises how the various variables are accessed, and under what conditions they are set and reset, while figures 4 and 5 give the pseudo-code forms of the algorithms used by the communicating process and the kernel respectively. For the process part, the pseudo-code is presented as a function, which takes as its arguments the process identifier and a set (or list) of gates, and returns the identifier of the gate on which communication takes place. In the case of the kernel, the code is executed every time a deschedule takes place.

Figure 6 gives an example of how the kernel works, by showing the order of flags being set and reset for a communication between the `Datalogger` process, which is offering a choice between `download` and `speed`, and the `Convert` process, which is offering the gate `changespeed`. As `Convert.changespeed` and `Datalogger.speed` and linked in the connection set, they trigger a communication when both are ready, as indicated on the diagram.

The algorithms as presented in figures 4 and 5 do not describe the way that timeouts and value-passing (as opposed to pure synchronisation) are handled, but they require only a little modification to deal with these features. Timeouts can be considered as choices with an extra possible branch, where instead of a communication event triggering the branch, a timeout event is used instead. By

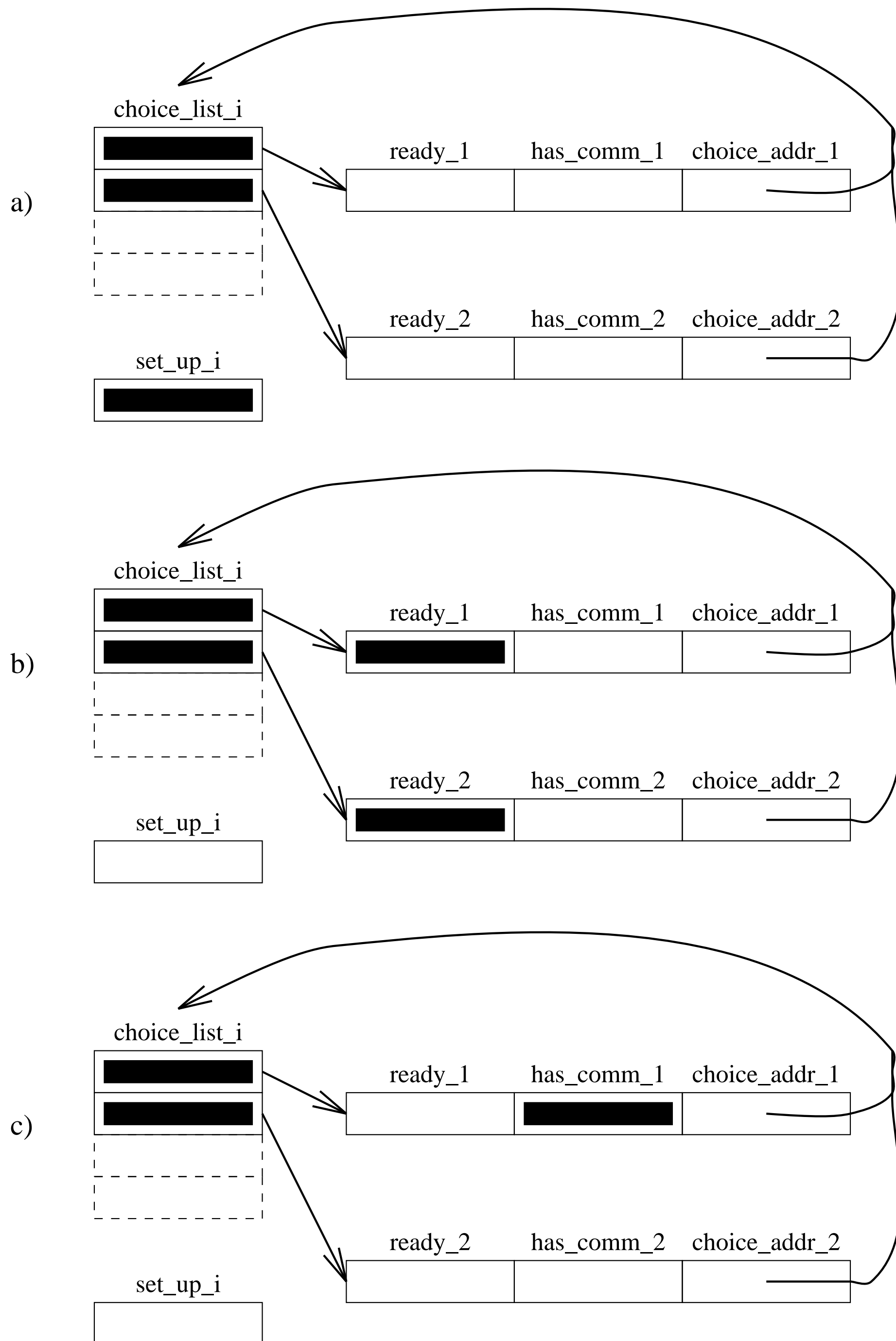

Figure 3: An Illustration of the Communication Flags

```
function communicate(i,choices)
for all j in choices do
  copy j into choice_list_i
end do
set_up_i := true
got_comm := false
do while (not has_comm)
  for all j in choice_list_i do
    if gate.has_comm_j = true then
      j_comm := j
      gate.has_comm_j := false
      got_comm := true
    end if
  end do
end do
return j_comm
```

Figure 4: Communication Function for Process i

```
for i = 1 to number of processes do
  if set_up_i = true then
    set_up_i := false
    for all j in choice_list_i do
      gate_j.ready := true
    end do
  end if
end do
for all (j_1, j_2) in connection set (* i.e./ j_1 and j_2 are connected *) do
  if gate_j1.ready = true and gate_j2.ready = true then
    for all j in list pointed to by choice_address_j1 do
      gate_j.ready := false
    end do
    for all j in list pointed to by choice_address_j2 do
      gate_j.ready := false
    end do
    gate_j1.has_comm := true
    gate_j2.has_comm := true
  end if
end do
```

Figure 5: The Kernel's Communication Test Algorithm

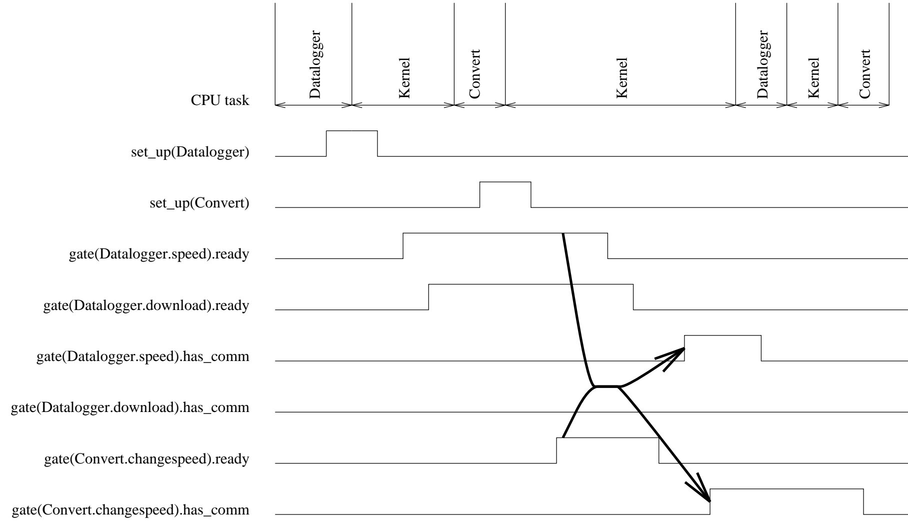

Figure 6: Communication Between the *Convert* and *Datalogger* processes (timings not to scale)

including a timeout in the choice list, the relevant communications are disabled if timeout occurs, and the timeout is disabled if a communication event occurs. A timeout has associated with it a *ready* flag, a *has_comm* flag, and the time at which it should occur. Instead of the kernel checking for a pair of gates to be ready to communicate, it checks to see whether the occurrence time has been passed, and if it has, the choice list is reset and the *has_comm* flag set, as for internal communications.

A value-passing mechanism is provided by associating with each gate storage for a data-in value and a data-out value. The value passed could be the data itself, or a reference to the data. When the kernel finds a pair of gates ready to communicate, it copies the data-out of one process to the data-in of the other and vice versa. As far as the calling routine for the process is concerned, an extra parameter is added to the communication function, which is a pointer to an array of values to be passed in. These values are copied to the relevant data-out slots before the *set_up* flag is set, and after communication has occurred the data-in from the gate which has communicated is copied into the relevant entry in the array of values. Note that data-passing is not modelled by AORTA as such, but this does not mean it cannot be used, only that it is not subject to formal analysis within AORTA. The concluding section suggests that other formal methods might be used in conjunction with AORTA to allow reasoning about the data within a system.

The actual implementation of the kernel and communication primitives (i.e. functions for communication and timeout) was written in 68000 assembly language, with the communication primitives callable from C. The code for the whole kernel, including the communication primitives, only occupies about 3 kilobytes of memory, with storage for the variables, flags, and stacks of a small system with four processes and ten connections occupying another 3 kilobytes.

Analysis of the Kernel Performance

Predictability is at a premium in a system which is to be formally verified, so the scheduler described 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 simple round-robin scheduler which switches processes at a fixed rate in time, regardless of their state of execution.

There are three aspects of the timing behaviour of an implementation which need to be verified: the amount of processing time each process gets, the time delay in communication, and the bounds which can be placed on timeout occurrence, which we shall deal with one at a time. Figure 7 gives a scheduling diagram for a system with three processes, and introduces the variables we will use in the analysis. The fundamental variable in the system is the interrupt or scheduling period, which we shall call p, and after each p time units the kernel is activated via a deschedule. Each time the kernel is activated (via an interrupt), a process is descheduled; once the kernel has finished its activity the next process is rescheduled. Depending on how much setting up and communication the kernel has to do, the amount of time it takes to perform all its tasks will vary, but can be bounded. We define the upper and lower bounds on time spent in the kernel at each activation to be k_l and k_u respectively, (actual figures for this implementation are given later,) and the value for a particular activation of the kernel is written k. The point at which each process starts will depend on how long the scheduler took, but the time between deschedules is constant, and is represented by d_i for the process i. For a system with the processes being scheduled one after another, the d_i will be the same for all processes, but where one process occurs twice as frequently as others (e.g. 12131213... as in figure 7) the d_i may be different, although they will all be integer multiples of p. We shall also refer to the process schedule time, which is the amount of time a process remains scheduled. This depends on how long the kernel takes to execute immediately preceding the process being scheduled, but is bounded by $[p - k_u, p - k_l]$.

The easiest way to calculate the amount of real time required for a certain amount of processing time on a process is to consider the amount of time that the process will be in the middle of the processing, but not scheduled. For the minimum execution time we have to consider the best situation, which is where the processing starts at the beginning of a schedule. In this case, if the amount of processing time required is less than the time the process schedule time, then the real time elapsed will be equal to the processing time required, as the processor will have been solely devoted to that processing for its complete duration. If the processing time is greater than the process schedule time then the number of blocks of unscheduled time will be given by $\lfloor r/(p - k_l) \rfloor$, where r_i is the required processing time and $\lfloor x \rfloor$ is the lower integer part of x. This comes from the assumption that this is the best case, so all kernel execution times will be minimal (k_l), giving a process schedule time of $p - k_l$. Having calculated the number of blocks of unscheduled time we get the amount of unscheduled time to be

$$\lfloor \frac{r_i}{p - k_l} \rfloor \times (d_i + k_l - p))$$

as each unscheduled block will last for $(d_i - (p - k_l))$. This gives a total elapsed

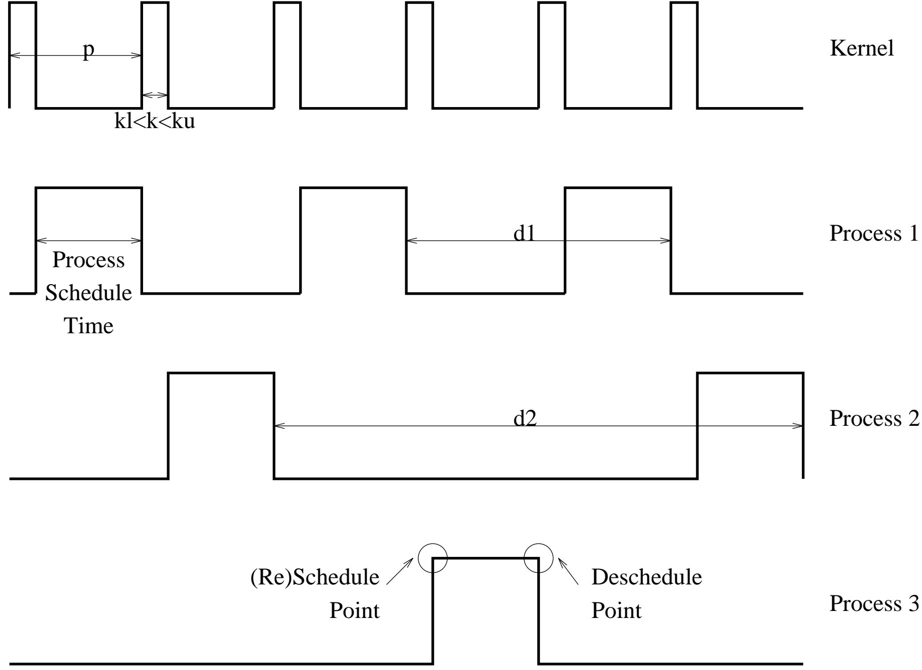

Figure 7: Scheduling diagram for three processes

time of

$$r_i + \lfloor \frac{r_i}{p-k_l} \rfloor \times (d_i - (p - k_l))$$

as the minimum for a required amount of processing r_i. The worst case (which is usually of most interest) is calculated similarly, but this time the computation starts just as the process is being descheduled, and all of the kernel execution times are maximal, giving an elapsed execution time of

$$r_i + \lceil \frac{r_i}{p-k_u} \rceil \times (d_i + k_u - p)$$

where $\lceil x \rceil$ is the upper integer part of x.

We measure the communication delay from the point at which the second (i.e later) process starts the communication procedure, to the point at which the process has noted the communication and carries on. This delay will be different for the two sides of a communication, but both sides will have the same upper and lower bounds on the delay. There are three phases to the communication, each of which contributes to the overall delay. Firstly, some processing is required to set up the choice list before the set_up_i flag is set. We define the upper and lower bounds on this time to be pre_comm_l and pre_comm_u respectively. Secondly, once set_up_i has been set, there is a delay during which the kernel will set up the communication, complete it, and eventually reschedule the process. The best case for this delay occurs when the process is descheduled just as it completes its precommunication computation, giving a delay of $d_i - (p - k_l)$. In the worst case, the process only just fails to complete before being descheduled, giving a delay of $2 \times d_i - (p - k_u)$. Thirdly there is a delay from the process next being scheduled, to the final completion of the processing required to reset the has_comm_j flag and continue to the correct branch of the choice. If we write the bounds on this time as $post_comm_l$ and $post_comm_u$, the minimum overall communication delay will be

$$pre_comm_l + (d_i - (p - k_l)) + post_comm_l$$

and the maximum will be

$$pre_comm_u + (2 \times d_i - (p - k_u)) + post_comm_u$$

Both of these figures assume that the very small amount of post-communication processing will be completed in one schedule block, and the maximum time can be reduced to

$$d_i + (p - k_l) - (p - k_u) + post_comm_u = d_i + k_u - k_l + post_comm_u$$

if it can be shown that the pre-communication processing will be completed in one schedule block (as would probably be the case if the communication immediately followed from another).

A timeout of time t, under a similar analysis, yields a minimum occurrence time of

$$pre_comm_l + (\lfloor \frac{t+p}{d_i} \rfloor + 1) \times d_i - (p - k_l)$$

and a maximum of

$$pre_comm_u + (\lceil \frac{t+p}{d_i} \rceil + 1) \times d_i - (p - k_u)$$

which can be reduced to

$$\lceil \frac{t+p}{d_i} \rceil \times d_i + k_u - k_l + post_comm_u$$

if the pre-communication processing can be guaranteed to complete within one schedule block.

In the above analyses, p, the scheduling period and d_i, the time between schedules for process i, may be adjusted by the implementor, but the figures for the kernel k_l and k_u are fixed by the kernel. The actual figures depend on the number of processes in the system, the number of connections, and the maximum number of gates in any choice. The minimum time is of the approximate form

$$k_l = 1222 + 98 \times no_processes + 88 \times no_connections$$

and the upper bound of the form

$$k_u = 2222 + 268 \times no_processes + 88 \times no_connections + 212 \times max_no_in_choice$$

where the figures given are for clock cycles on a 68000 micoprocessor. For a typical system with four processes and ten connections, running on a 68000, the bounds on kernel execution time are $[2406, 5032]$ clock cycles, i.e. $[0.3, 0.6]$ milliseconds for a clock speed of 8MHz.

Using the Kernel in a Design Method

As the main point of interest of the kernel is its amenability to formal analysis, we now look briefly at how the kernel fits into a scheme which provides a verifiable route from specification to implementation. Ultimately the aim is to be able to write a formal specification, and produce a verified AORTA design which can then be implemented semi-automatically. This paper describes how an AORTA design can be implemented: the kernel is customised to the design (using the connection set), and a C program for each process, involving communication, timeout and nondeterministic choice is generated automatically. The

code for the computations concerned (as modelled by delays `[ .. ]`) is written separately by the implementor, attached to the AORTA design as an annotation, and included with the communication code by the code generator. Information about data passing during communication, which lies outside the scope of the AORTA design, is included as an annotation to the design in the same way, and is used to set up and read a value array. Two routines are offered by the kernel, called `communicate` and `communicatet`, for communication and communication with timeout respectively, which are called by the code produced by the code generator when a choice or timeout is required. These are implementations of the algorithm presented in figure 4. As regards verification, the previous section gives an analysis which allows the bounds on the time for delays, communication, and timeout to be calculated; put together with bounds on the processing time required for each computation (using [6] for instance) this provides verification techniques for all of the timing aspects of the implementation.

The route from an AORTA design to an implementation is complete, so that a system can be built automatically from its design, and have its real-time properties verified. There is currently a simulator tool, which allows a design (including its timing) to be explored by a user in an interactive way, which would be used as the first step in a verification process. Figure 8 shows how the work fits together into an overall design method: arrows going downward represent implementation, arrows upwards verification; solid arrows indicate currently available routes, automated where appropriate, and the dashed arrows represent possible future pieces of work.

AORTA has been tried on some small examples, including the car cruise controller, but there is not yet enough information to evaluate fully whether our assumptions about the tradeoff between efficiency and predictability are sound. Other questions may be raised about the applicability of AORTA to safety-critical systems, given its use of C, interrupts, and concurrency. The choice of C is not really an issue, as we mentioned earlier that any language which can be analysed for timing could be used instead. Interrupts and concurrency, however, do raise more interesting points, as they are sometimes prohibited because of the unpredictable behaviour they can cause. We would argue that the aim of this work is to provide techniques for reliably, predictably and verifiably using concurrency where appropriate — the use of fixed period interrupts is a secondary issue. It is our job to prove that concurrency using interrupts can be used safely within critical hard real-time systems, as long as a proper analysis is performed.

Conclusion

In a hard real-time system it is not just good average performance that is required, but guaranteed performance in all cases. By using a predictable scheduler coupled with a formal analysis technique, we can provide sound guarantees about system performance, guarantees which refer to the actual implementation rather than to an abstract model of it. The techniques described here are as applicable to distributed processing as to multitasking, so that is possible to defer design decisions about how many and what kind of processors may be required until after the AORTA design is complete. In addition, there is no reason why a process need be restricted to software: an AORTA process could equally well be implemented in hardware, although this would require further investigation.

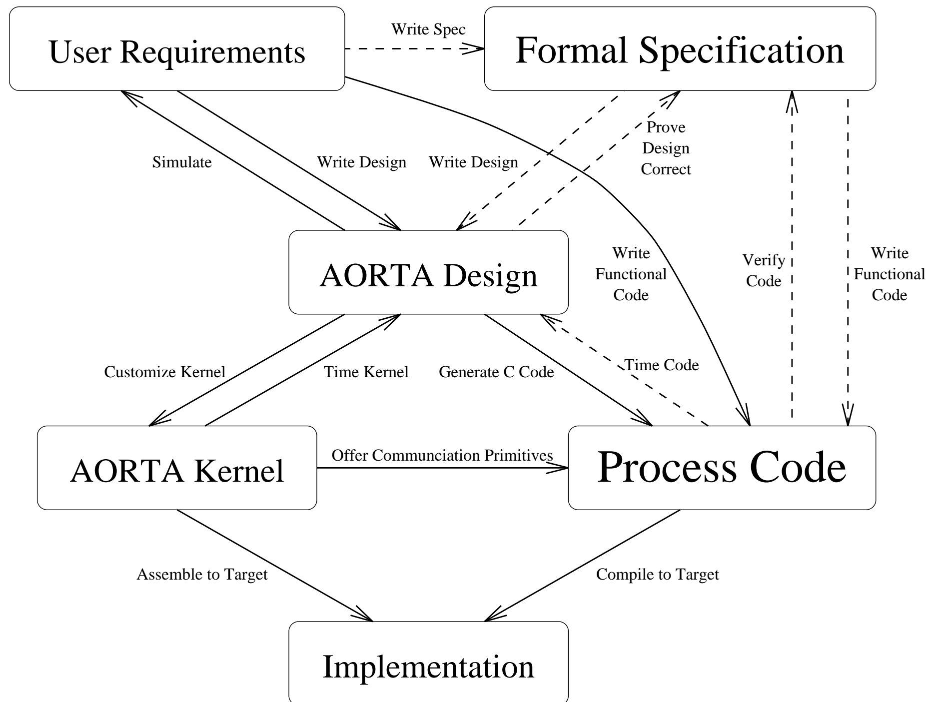

Figure 8: AORTA within a real-time design methodology

Referring to figure 8, we can see where future work will fit in to a design method. Techniques for formally verifying an AORTA design with respect to a formal specification are currently being looked into, and it is hoped that existing work such as [14, 15] may be useful in providing automatic verification procedures. One particularly interesting piece of work would be the integration of existing formal methods for developing sequential code (such as Z [16] or VDM [17]) with AORTA, so that the timing of a system and its functional correctness could be verified in a unified way, giving a complete verification technique for hard real-time systems.

Acknowledgements

The authors would like to thank the University of Northumbria at Newcastle and Northern IT Research for their financial support, and acknowledge the helpful comments of Mike Bradley and the anonymous referees on earlier versions of the paper.

References

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

[2] G Bruns and S Anderson. A case study in the analysis of saftey requirements. In H H Frey, editor, *IFAC symposium on safety of computer control systems (SAFECOMP '92)*, pages 1–6, 1992.

[3] M Thomas. The industrial use of formal methods. *Microprocessors and Microsystems*, 17(1):31–36, 1993.

[4] N G Leveson and C S Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[5] 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 '94 (FTRTFT '94) (To appear)*. Springer-Verlag, 1994.

[6] 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.

[7] R Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[8] 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.

[9] C A R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] 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.

[11] G Jones. *Programming in occam*. Prentice Hall, 1987.

[12] Department of Defense. *Reference manual for the Ada programming language*. springer-Verlag, 1983.

[13] 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.

[14] 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.

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

[16] B Potter, J Sinclair, and D Till. *An Introduction to formal specification and Z*. Prentice-Hall, 1991.

[17] C B Jones. *Systematic software development using VDM*. Prentice-Hall, 1986.