A Formal Design and Implementation Method For Real-Time Embedded Systems

Steven Bradley William Henderson David Kendall
Adrian Robson
Department of Computing, University of Northumbria at Newcastle, UK

Stephen Hawkes
International Research & Development Ltd., Newcastle-upon-Tyne, UK

Abstract

This paper tackles the problem of using formal methods for practical real-time system development and verification, and is based on a real example. Many formal methods for real-time systems have been proposed, but this technique (AORTA) is one of the few to address the issue of how formal designs are to be implemented. Earlier papers on AORTA have been based on providing the formal semantics of the language, and on particular aspects of implementation or verification. This paper concentrates on setting AORTA within the development life cycle, and demonstrating that the approach can be adopted for non-trivial examples.

1. Introduction

Many computer systems are required not only to deliver correct results, but to deliver those results at the correct time — such systems are called *real-time systems*. These systems are most often to be found in control equipment, such as automatic washing machines, fly-by-wire systems, life-support machines, car braking systems, or industrial plant controllers. Many *safety critical* applications, where correct functioning is of vital importance because of the hazardous results of malfunction, fall into the category of real-time systems; many of the applications already mentioned would be classified as safety critical. In addition to the risks of physical harm which might be caused by faulty software, there is the increasing risk of financial loss associated with providing functionality with software. A product recall following the detection of an error in embedded software could be very costly. Software is perhaps the most easily modified of all technical products. However, the correction of faulty software in embedded systems can become just as much a financial burden to the manufacturer as the replacement of a faulty mechanical handbrake in an automobile.

Implementors of real-time systems will be aware of the difficulty of predicting performance at an early stage in the development cycle. Usually, it is only possible to measure performance at a late stage in the implementation of the software and hardware. Even when a system is available for evaluation, verifying that it meets timing requirements will simply involve conducting (possibly many) tests with the system under different loading conditions. It is common experience that concurrent real-time software exhibits complex behaviour with extremely large numbers of states. With anything other than a trivial application, testing is likely to consider only a small fraction of possible behaviours of the system. Clearly, providing confidence that a system will meet deadlines under every circumstance by testing alone will not in general be possible. Cullyer and Storey [7] outline the problems of verifying the real-time behaviour of software and describe testing techniques and strategies for safety-critical software; they conclude that current tool support for testing such software is weak.

A further problem of the testing approach is that measurements made on the system may well point to inadequacies in the implementation or perhaps in the design. Reasoning about where the problem lies can be very difficult since design decisions are often informally linked to the implementation. This encourages developers to avoid performance issues until the implementation stage. The ability to reason about performance at an early stage, during design, seems to be an important goal. Clark et al [6] underline the inadequacy of testing as a means of verifying real-time systems and emphasise the need to provide high confidence *early* in

the development process that timing requirements will be met by the final system.

The quality of software for high profile safety-critical applications such as aviation or automobile traction control is clearly a concern. However, there is an increasing use of embedded controllers in a wide range of products such as kitchen appliances, heating, fire detection and security systems. These systems can also pose a threat if poorly programmed; for example, a faulty controller for a washing machine could conceivably create a hazardous condition by allowing the water heater to remain on when the machine is dry.

Recent European law [8] places the responsibility for producing safe embedded systems on the software developer; it will become essential that developers use reliable methods and tools to demonstrate that they have discharged their responsibility. What methods are deemed *reliable* is changing as the subject develops through research and standards respond; developers might reasonably be expected to be aware of best-practice and use working methods which reflect this. Currently, highly rigorous, mathematically-based methods are mandatory only for defence contract work. Although the use of formal techniques during software development can provide greater confidence in the quality of designs, they do not necessarily lead to a straightforward implementaion as executable code.

An alternative approach to system verification, perhaps best characterised by the phrase "correct by design", challenges the traditional approach of testing. However, formal mathematical techniques, which aim to produce correct systems, are often far removed from practical implementations. In particular, very little work exists on relating real-time performance of systems with the mathematical models used to reason about real-time behaviour. The remainder of the paper discusses a technique which has a formal basis, but which admits verifiable implementation; a real example is used to demonstrate the method in practice.

2. The Language of AORTA

AORTA Application Oriented Real-Time Algebra attempts to address some of the problems faced by designers of real-time systems by providing a formal language for system design and verification, which deals with quantitative time, and relates directly to implemented systems. In order to ensure implementability of designs, the language is more restrictive than many other timed formalisms, which can be used to model a wide range of possible behaviours, many of them unimplementable. The basic language is a timed process algebra, related to the timed variants of process algebras like CCS [14], LOTOS [11] and CSP [10]. Features of AORTA which are not found in other timed process algebras include

- static definition of concurrency parallelism, in order to simplify timing analysis of processor multitasking,
- static definition of communication paths, in order to simplify timing analysis of inter-process communication,
- time bounds for delays and time-outs, to cope with the implementation-related problems of jitter and clock discretisation.

These features do, in some cases, restrict the expressivity of the language for modelling and specification purposes, but the language is not meant to address these problems: it is intended purely as a design language. We believe that these restrictions do not seriously hamper the ability of the language to design real real-time systems, and the notation has been exercised on several medium-sized examples, such as the submersible controller considered in this paper. Rather than try to define a wide-spectrum language, which can be used to define systems at different levels of abstraction, each related by a refinement proof technique, we have chosen to use a timed temporal logic for the specification, with model-checking as the proof method. This allows us to focus the language more closely on design, and to avoid proof techniques such as timed bisimulation and timed refinement, which do not extend well from the untimed setting to the timed setting. More detailed justification of the choices made in choosing the language constructs and proof techniques can be found in [3]. The aim of this paper is to describe how AORTA can be applied, but first the language needs to be introduced.

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

2.1 Sequential Processes

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

There are two constructs which are used to introduce time, and each of these has a deterministic and nondeterministic form. The first construct is a delay, which causes the process to pause for the amount of time specified, during which time no actions are offered — time consuming operations like computation are represented in this way. As exact timings are not always known for such delays, 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 time-out 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 time-out is implemented, a precise figure for the time at which control is transferred may not be available, so an interval of possibilities can be given instead. A choice process S which times out to process T if no communication happens within time t is written `S [t> T`, and if the time is bounded by $t1$ and $t2$ it is written `S [t1,t2> T`.

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

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

Table 1. Summary of concrete syntax for sequential processes

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

In summary, a sequential process may be constructed from action prefixes, summations (choices over prefixed processes), time delays, time-outs 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.

2.2 Parallel Composition and Communication

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

existing process algebras.

In all of the common process algebras the communication actions of any process are visible to any other process unless explicitly hidden or restricted, which leads to problems on two fronts. From an implementation point of view this requires some way of broadcasting all available actions to all processes. Even more problems are encountered in implementing the multiway synchronisation of CSP and LOTOS, as witnessed by the restriction to two-way communication in occam [12] and the need for a special protocol in LOTOS [17]. For a small to medium-sized system, which is all we can hope to verify at the moment, the mechanism for providing such communication facilities may be an excessively costly overhead, both in terms of implementation and verification.

The availability of all actions to all processes can also cause problems in verification, as checking for all possible communications requires testing of each pair of processes for communication on each action, leading to an explosion in the number of checks to be made. This explosion can be contained by restricting communication to a named set of channels between processes.

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 some pairs of gates linked, using a connection set written in angle brackets after the processes. An element of the connection set is a pair of gates to be connected, along with bounds on the time for a communication delay along that connection. External events, communications between processes and the external environment, also appear in the connection set. These are distinguished by the name *EXTERNAL* and each has an associated function which implements the event - a device handler. The communication delay for an *EXTERNAL* is the bounded execution time of the device handler.

The formal semantics of AORTA are presented elsewhere [3], and is given as a stratified set of transition rules. This gives rise to a transition system which can be represented as a *timed graph* [1], so that automatic verification via real-time model-checking can take place [15].

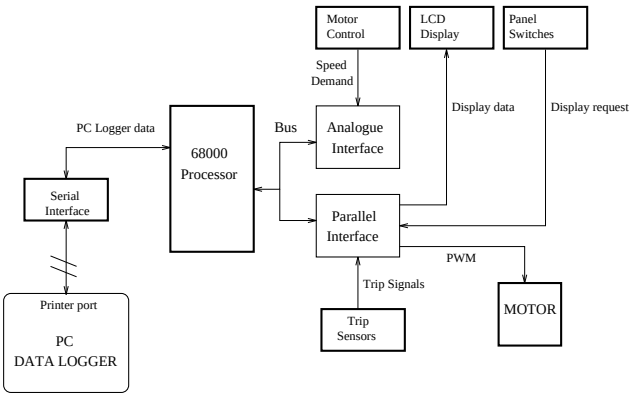

Figure 1. Submersible Control - System Diagram

3. An Example Application

In order to illustrate how AORTA is employed in practice, we describe a simple embedded application and show how a design is expressed in AORTA and implemented to run on a single board computer.

International Research & Development Ltd., a subsidiary of Rolls-Royce PLC, amongst its many activities, designs electric propulsion systems for small submersible vehicles [16], see **Figure 1**. The original controller was implemented using an Intel 8051 micro-controller device and custom interface hardware, but for the purpose of this case-study it was reimplemented on a 68000 microprocessor to make use of an existing AORTA kernel.

The control system performs four main functions, namely speed control, motor overload protection, management of a user interface and the servicing of data requests from an external data logger. The overall structure of the system is illustrated in a 'Process' diagram, **Figure 2**. The propulsion motor (which drives a propellor) is controlled via a pulse-width modulation (PWM) unit. The current propulsion speed demand, a proportional signal, is read from a manual input and is converted into appropriate PWM control values. In order to avoid motor power demands which exceed the rating of the power unit, the control function smoothsout rapid changes in speed demand.

An external data logger is connected to the propulsion controller via a serial interface. The logger is required for field trials and fault finding to allow operating conditions to be recorded whilst the equipment

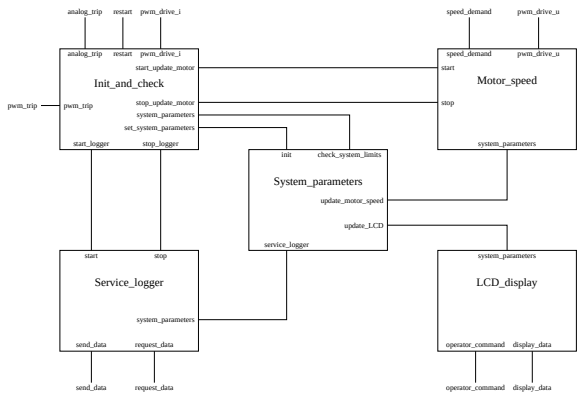

Figure 2. Submersible Control - Process Diagram

is tested. At any time a request may be made by the logger indicating the data required - motor current, speed, etc. The propulsion controller responds by replying with the most recently available data. A display panel allows the operator to view system parameters such as speed demand, motor temperature, etc.; the value displayed is selected by a manual switch.

The system also detects overload conditions of temperature and motor current. Should an overload be detected, the motor will be brought to a controlled stop and the speed controller restarted only after the user has pressed a 'restart' control. Following an overload trip, the system is capable of servicing data logger requests and providing current system parameters via the user interface. Notice that the design includes a process, *System_parameters*, which manages mutual exclusion of the system parameter data.

The application illustrates the following features characteristic of real-time applications:

- Both periodic and sporadic inputs
- Timing constraints
- Concurrency

In common with most control applications, the functions of the propulsion controller are best implemented by concurrent processes. In addition to being a convenient design abstraction, concurrency allows the implementation of polling cycles of arbitrary frequency. It would be difficult to manage the different polling rates and sporadic behaviour of inputs and error conditions using a single process or thread of execution and yet guarantee timely behaviour. Concurrency also allows the system to respond to independent inputs. Clearly, it is necessary that any sporadic data logger requests are handled in a way that does not interfere with overload trip recognition or control over the speed of the motor.

The AORTA design of the propulsion control application is provided in full in [5] along with an example of a device driver. We shall provide a detailed commentary on just one process, *Init_and_check*, since this exploits all features of the AORTA notation. The process *Init_and_check* is responsible for starting other system activity, checking for overload conditions and halting other proceses if necessary following an overload. *Init_and_check* begins by communicating the initial system status to the process *System_parameters*, initialising the PWM drive and starting the processes which control the motor and service the data logger:

```
Init_and_check =
 set_system_parameters@!INITIAL_STATUS@.
 pwm_drive_i.start_update_motor.
 start_logger.Check_safety
```

The process behaviour is then given by *Safety_check* which initially offers a choice of the external events, *analog_trip* and *pwm_trip* subject to a time-out:

```
Check_safety =
 (analog_trip.Close_down
  +
  pwm_trip.Close_down)
  [100.0,110.0>
  (system_parameters@?parameters@.
  [0.0112,0.0293
    @check_parameters(parameters,&over_limit);@]
  (Check_safety ++ @over_limit@ Close_down))
```

These events would take place should the analog interface or pulse-width-modulation units fail. The event *pwm_trip* is defined in the connection set as an EXTERNAL which is implemented by the C function *pwm_trip_event*, also listed in [5].

The choice of the external events is offered for a period of between 100 and 110 ms using a time-out. If neither event occurs in this period, the process continues to read the latest values of motor current, temperature, speed, etc. from *System_parameters*. A check is performed on the data by the C function *check_parameters* to determine if all readings are within their control limits. Further process behaviour is decided by a data-dependent choice (++) between *Check_safety* and *Close_down*. If *over_limit* is false, the behaviour is defined by *Check_safety* which repeats the safety check. On the other hand, if *over_limit* is true, events are communicated with the motor controller and data logger to

close them down and no further action will take place until a *restart* event occurs:

```
Close_down =
 stop_update_motor.stop_logger.
   restart.Init_and_check
```

The process *Motor_speed* controls the speed of the drive motor by reading a speed demand, calculating new PWM parameters and outputting these to the drive unit. The process can be halted following an overload since it polls the event *stop* about every 50 ms. *Service_logger* waits for requests for data from the external logger, reads the required system parameters and replies with the data. Like *Motor_speed*, this process can be halted following an overload. *LCD_display* waits for operator commands, reads the required system parameters and displays the values. The remaing process *System_parameters* provides mutually exclusive access to system data from the other processes.

It is necessary that additional details are attached to the AORTA design prior to code generation. The *annotations* introduce the names of functions which perform computation or input and output and other details. Annotations are introduced within @...@ delimiters for the following reasons:

- in delays to define what code is used to perform the computation,
  e.g.,
  $[0.56, 0.62@check(parameters, \&over\_limit); @]$,
  where *check* is a C function.

- in communication, annotations are used to define what values are to be passed, e.g., $speed@?demand@$ causes a value to be received using the event 'speed' and stored in the variable, *demand*. The ? annotation communicates with a matching ! which sends a value.

- in EXTERNAL event connections, annotations are used to define the names of device drivers used to effect communication,
  e.g., $(Init\_and\_check.trip, EXTERNAL : 0.05, 0.1@pwm\_trip; @)$. The names are simply C functions.

- in data-dependent choice, annotations are used to define how the choice is resolved using a C language conditional statement,
  e.g., $++ @pressure > critical@$.

- in the definition of processes to declare variables used within computations and to define functions used for I/O.
  e.g., $@\#include < adc\_io.h > @$.

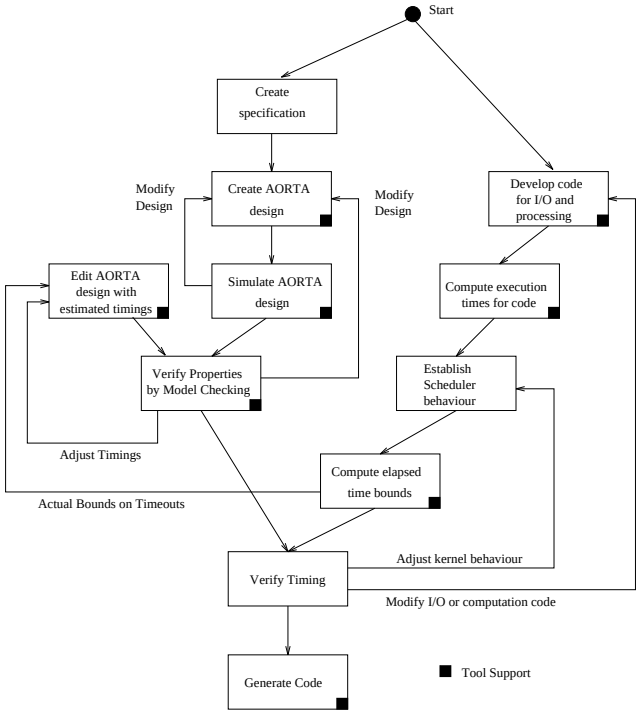

Figure 3. The AORTA Development Process

4. System Development – An AORTA Tool set

A toolset to support the design, validation, verification and implementation of AORTA designs is under development. The design and implementation process involving the use of these tools is summarized in **Figure 3**. Considerable progress has been made in providing tools to support some aspects of AORTA design and implementation - such as code generation, design simulation and the management of code. However, work remains to be completed in the areas of specification and timing verification.

The following tools have been developed:

- Annotation Tool
  AORTA designs are written and edited as ASCII documents. An annotation tool facilitates the editing and inspection of AORTA designs and annotations by managing them as hypertext documents. Thus, a design can be viewed without annotations if desired or the detailed annotations can be inspected by opening hypertext links.

- Graphical Simulator
  A simulator allows designs to be exercised at an early stage to explore their temporal behaviour –

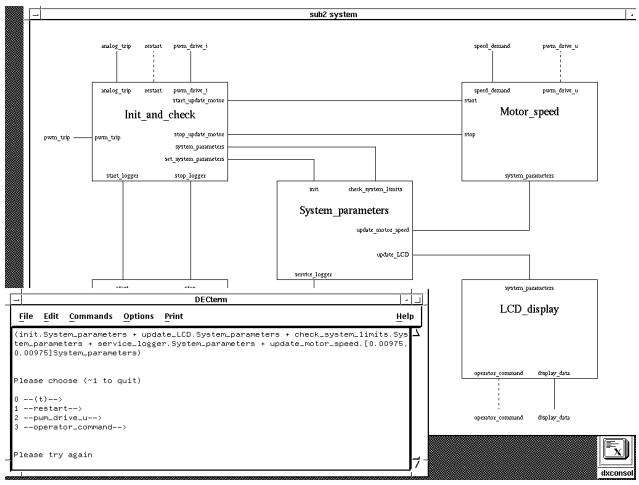

Figure 4. Screen Shot of Simulator

the events that systems offer as time progresses. The simulator allows the behaviour of a design to be stepped through using a simple menu-driven interface. There is also a facility for showing graphically which communications are available - see **Figure 4**. The simulator has proved to be a valuable tool in providing feedback to the designer prior to formal verification or code generation.

- Code Timing
  The bounded duration of computations and other processing activities appear in the AORTA design. The timing of code fragments is currently undertaken by an analysis of assembly language generated by the compiler. Bounded times are derived for computations and for inputs or outputs. A tool is used to calculate the elapsed time of these computations and time-outs, taking into account the time lost in scheduling and in managing events, based on a an analysis of round-robin scheduling [4].

- Verification
  Verification that a design exhibits desired properties is facilitated by a process of timed state graph construction and model checking [15]. A number of different types of properties can be verified. For example, in the context of the earlier example, one might seek to establish a bounded response such that following the input of the motor current which indicates an overload, the motor speed control process will stop within a specified time period.

  Although the submersible control system cannot be regarded as large, only 5 concurrent processes and a total of 19 events, in fact the system can be in any of about 7.10^4 states. The verification process is extremely demanding, requiring considerable computer time to check simple properties; the techniques and algorithms employed are subject to further development.

- The Kernel
  The execution of processes comprising an AORTA system is managed by a dedicated software *kernel* [4]. The kernel schedules processes in turn to share access to the CPU. The processes call on the kernel to undertake time-out operations, synchronise, communicate and perform input and output.

  It is of paramount importance that the kernel's functions are fully predictable, so that each process has a known access to the CPU and that event synchronizations have predictable durations. The kernel uses a simple scheduling mechanism which gives each process a fixed time slice in a round-robin sequence, although other methods such as fixed priority scheduling are possible. Thus, processes have a guaranteed share of CPU time and can be treated as independent threads of execution between synchronising points. Processes wishing to synchronise or communicate employ the kernel to achieve this; between each process time-slice, the kernel looks for possible communications and manages their completion. The time bounds of context switching and the management of interprocess synchronization have been determined by an analysis of the kernel [4]. Thus, it is possible to verify that the times expressed in the AORTA design, to complete any processing and undertake synchronizations, are correct.

- Code Generator
  A code generator translates the AORTA design by making the necessary calls to the kernel and incorporates the annotations in the compiler-ready output. Process code, device drivers and the kernel are linked to build a system which is downloaded onto the target processor.

5. Conclusions

There is a pressing need for reliable methods of designing and implementing embedded real-time software. Current methods lack the rigour which admits a quantified analysis of performance prior to implementation.

We have described a formal approach to embedded systems development which does not rely on testing for verification. Instead, the required temporal behaviour of the system is expressed in a design which can be

implemented automatically following annotation. The major benefit of adopting a formal treatment is that implemented systems will have the desired timely behaviour expressed in a design and will meet deadlines under all circumstances. The technique has been illustrated using an example which embodies many of the features characteristic of small embedded systems; we believe this demonstrates that this formal approach can be applied to problems of practical significance. A toolset has been described which supports the development AORTA applications during design, validation, verification and implementation.

Current and future work on AORTA includes the implementation of 'industrial strength' specification, design and timing tools to facilitate AORTA systems development. Also of interest are the application of alternative scheduling techniques to the round-robin mechanism described in this paper, the distributed implementation of AORTA designs using CAN [2] and the use of a commercial real-time kernel to support AORTA.

6. Acknowledgements

The authors are grateful to **International Research & Development Ltd.** (part of Rolls-Royce Industrial Power Group) for permission to publish details of the submersible control system. The work reported in this paper was undertaken with the financial support of both **Northern IT Research** and **The University of Northumbria at Newcastle**.

References

[1] R Alur, C Courcoubetis, and D Dill. Model-checking for real-time systems. In *IEEE Fifth Annual Symposium On Logic In Computer Science, Philadelphia*, pages 414–425, June 1990.

[2] Bosch GmbH. *CAN specification*, version 2.0 edition, 1992.

[3] S. Bradley, W. Henderson, D. Kendall, and A. Robson. Application-oriented real-time algebra. *Software Engineering Journal*, pages 201–212, September 1994.

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

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

[6] J.A. Clark, J.A. McDermid, and A.Burns. Analysing high-integrity systems. *Computing and Control Engineering Journal*, 5(5):18–23, February 1994.

[7] W.J Cullyer and N.Storey. Tools and techniques for the testing of safety-critical software. *Computing and Control Engineering Journal*, 5(5):239–244, October 1994.

[8] D.Davis. Safety-critical systems - legal liability. *Computing and Control Engineering Journal*, pages 13–17, February 1994.

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

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

[11] International Standards Organisation. *Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observable behaviour*, 02-15 edition, 1989.

[12] G. Jones. *Programming in Occam*. Prentice hall, 1987.

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

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

[15] S.Bradley, W.D Henderson, D.Kendall, and A.P Robson. Validation, verification and implementation of timed protocols using AORTA. In *15th International Symposium on Protocol Specification Testing and Verification*, June 1995.

[16] S.Hawkes. *A PWM Motor Controller - Functional Specification*. International Research and Development Ltd, ird/93-4project edition, November 1994.

[17] R. Sisto, L. Ciminiera, and A. Valenzano. A protocol for multirendezvous of lotos processes. *IEEE Transactions on Computers*, 40(1):437–446, April 1991.