Practical Formal Development of Real-Time Systems

S. P. Bradley W. D. Henderson D. Kendall A. P. Robson

Department of Computing
University of Northumbria at Newcastle
Newcastle-upon-Tyne, NE1 8ST, UK

Abstract

The complexities of real-time systems are such that it is often thought necessary to give a formal justification of their correctness, especially if they are to be used in a safety-critical environment. In this paper we describe our work on a formally based design method for real-time systems which allows the timing aspects of a concurrent system to be mathematically described and verified, as well as semi-automatically implemented. Our design language, AORTA, is a timed process algebra, with features to ensure that all designs can be implemented. A predictable real-time kernel is also described, which is used in the construction of a system from an AORTA design, and which allows the timing of the implementation to be verified.

1 Background and motivation

There is much existing work on methods for real-time systems, both on the theoretical aspects of verifying the correctness of a real-time system [1], and on the practical ways of guaranteeing performance via a real-time kernel [2], but little that links the two. If practical formal techniques are to be found for real-time systems, then both high-level, more theoretical aspects must be considered as well implementation performance issues such as scheduling. There is some work which attempts to link the higher and the lower level, such as the implementation of formal models by compilation of (untimed) LOTOS [3, 4], or the overall system design methodology of the (non-formally based) MARS project [5], but we are not aware of any work that addresses the practical design, implementation, and formal verification of a time-critical system. In response to this apparent lack, we have developed a formal design language based on process algebra called AORTA (Application Oriented Real-Time Algebra) [6], with the specific aim of providing a complete route from a (timed) formal specification to a verified implementation.

The first goal of our project has been to provide a means of producing a real-time system from its formal expression in AORTA, as most of the existing theoretical work covers the verification of a design with respect to a formal specification. Work on real-time kernels has made advances in ensuring that processes will get through their work as quickly as possible. Sometimes, however, this is at the expense of making the scheduling arrangements too complex to be able easily to provide reliable predictions about the performance of an interacting set of processes. Whilst priority-based scheduling algorithms may be provably optimal, it is not always optimality that is important — in particular, predictability of time-critical systems can be crucial. On this basis we have reverted to a very simple yet predictable fixed time-slice round-robin scheduler, so that timing is easier to predict, as the performance of each process does not depend on the performance of others except at explicit communication or synchronisation points. The efficiency sacrificed in using such a scheduling mechanism is balanced with the reduced cost of developing a verified system; as hardware costs are relatively low compared with development costs, we feel that this tradeoff is often justified.

The kernel also provides sound, safe and predictable communication primitives, based on Ada style synchronous communication, which correspond directly to the communication constructs in the AORTA design language. Together with a timeout facility, this provides a direct route to implementation, by C code generation from the AORTA design. Although the AORTA design only deals with the timing and intercommunication of the processes, the sequential code within a process is included in a manageable way, and the timing of non-generated code is verified by a combination of bounds on processing time of the code and the processing distribution figures available for the kernel in [7].

2 The AORTA design language

Timed process algebras are widely known, but are usually used for modelling or specifying real-time systems, rather than designing them. Our language, whilst having some features in common with timed (and untimed) process algebras, is distinguished by its implementability. There are two main points of difference, the first being that the number of processes within a system is constant throughout the lifetime of the system, making processor allocation, and hence computation times, easier to verify. Secondly, all timings in the design can be expressed as upper and lower bounds, rather than exact figures, as in reality bounds can usually be given, where precise figures may not exist. It is this representation of time bounds which is most problematic in existing timed process algebras.

In order to keep the model of the timing tractable, data within a system is not represented in AORTA, with all computation being represented only by the (bounds on the) amount of time required to complete it. Within each process communication is represented by the name of the gate on which communication is to take place, and bounds are placed on the amount of time between both sides of the communication being ready, to the communication actually taking place. Processes are written in a simple equational format, similar to that used in many other process algebras. A process written

```
Convert = in.[100,150]out.Convert
```

will wait for communication on its `in` gate, before doing some computation which lasts for between 100 and 150 milliseconds (any time units, discrete or dense may be chosen for a design — [0.1,0.15] would be an equally valid expression) and offering communication on its `out` gate. Once this second communication has taken place the process will start again waiting for an `in` communication. This is how a process which accepted temperature data and converted it to a different format would be expressed. The bounds on communication times are given by a separate function which takes a gate identifier and returns a time interval.

A choice construct is provided, similar to the `+` of CCS and the `select` statement of Ada, which allows several possible communications to be offered. The future behaviour of the process depends on which is completed first. Timeouts may also be defined, so that if none of the communication choices offered are taken up within a certain time, then control passes to another branch. Again, exact figures are not usually available for occurrences of timeouts, so bounds are used instead. If our `Convert` process is to accept input or allow its conversion mode to be changed, this would be written

```
Convert = in.[100,150]out.Convert
          +
          mode.[300,400]Convert
```

where the reconfiguration procedure takes between 300 and 400 milliseconds. If the data offered on the `out` gates is also to be kept up to date then it may need to be refreshed every 1.5 seconds or so, which is achieved by adding a timeout to the `out` communication:

```
Convert = in.[100,150]
           (out.Convert)[1450,1550>Convert
          +
          mode.[300,400]Convert
```

where 1450 and 1550 are estimates on the bounds which can be placed on a timeout of about 1500 milliseconds.

The last construct which can be used in the definition of individual processes is a data-dependent choice, used where the flow of control of the process depends on the value of some data in the system. Data is not modelled in AORTA, so this is essentially a nondeterministic choice as far as the process algebra is concerned. The choice between two possible behaviours is represented by `++`, so that if our `Convert` process is to give a warning if the value that it finds is outside a certain range, this would be written

```
Convert = in.(Convert2 ++ warning.Convert2)
          +
          mode.[300,400]Convert
Convert2 = [100,150]
            (out.Convert)[1450,1550>Convert
```

A system usually consists of the parallel composition of two or more processes; this is represented using the traditional process algebra bar `|`, with a connection set showing pairs of gates which may communicate. This explicit connection of gates allows for a more efficient implementation, and simplifies verification. The connection set is represented by pairs of gate identifiers written in angle brackets after the processes of the system. A plant control system incorporating the `Convert` process with a `Control` process and a `Datalogger` process, is written as follows:

```
Tempsys =
( Control | Convert | Datalogger )
<(Control.changem,Convert.mode),
```

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

The ordering of the pairs of gates is not important, and not all gates of the processes need be connected: those left free will have to communicate with the environment, like the `in` gate of our `Convert` process. Figure 1 gives a diagrammatic representation of `Tempsys`.

Most features of typical small embedded systems can be designed using this language: resource contention can be handled with the choice construct, and polling loops with a timeout. As well as allowing implementation, AORTA has a formal semantics given in terms of a timed transition system, which allows formal reasoning to be done about the design, and the possible application of model-checking techniques such as [8, 9]. Space does not allow us to go into the details of the semantics here, but see [6] for more details — it remains now to show how AORTA designs can be implemented in practice.

3 Implementation and the kernel

As the main point of the AORTA design language is its implementability, we outline here the kernel which we have written which allows AORTA designs to be verifiably implemented. We mentioned earlier that we have adopted a very simple approach to scheduling in order to be able easily to verify the performance of each of the processes. This is achieved by using a fixed time-slice round robin scheduler, where a fixed schedule of processes is executed on the kernel at a fixed frequency, so that each process has a guaranteed amount of processing time per unit real time, unaffected by the performance of the other processes. For a given amount of processing time required, bounds can be put on the amount of real time required, so that the timing of a piece of sequential computation can easily be verified given the processing requirements of that computation. Bounds on the computation time required for a piece of code can be found using techniques such as described in [10].

At each scheduling point, the kernel checks through the list of connected gates to see if there is a pair which is ready; if there is then it effects the communication, signalling to the processes involved that it has taken place, and disables communication on gates which were in choice with the successful gates. It then looks for possible external communications (on gates that are left unconnected) before looking through the list of timeouts to see if any have exceeded their time limit. By checking for communications and timeouts at every reschedule, bounds can be placed on the time for a communication to take place once enabled, and for a timeout to come into effect.

Communication primitives are offered by the kernel as C functions which are called from the processes. The calls to the kernel are generated automatically from the design, along with the parameters of the kernel, such as the number of processes, and the gates which are to be connected. Details such as the code to be executed as part of a computation delay, the data to be passed in the communication, and the conditions for a data-dependent choice are attached as annotations to the design. They have no interpretation in the formal semantics, where they are viewed as comments, but they allow the code to be included in the correct place without having to edit the code generated from the design. Putting the kernel together with the generated code allows an implementation to be generated automatically from an annotated design. The pieces of sequential code still have to be hand constructed, but once written, their timing, and hence the timing of the whole system can be verified.

4 Current and future work

The work on providing a route from design to implementation is complete, so that a system can be built automatically from its design. Although all of the verification methods are manually available, they have not yet been integrated into a single tool. The verification of an AORTA design will be addressed soon, but there is currently a simulator tool, which allows a design (including its timing) to be tested out by a user as the first step in a verification process. It is hoped that existing formal verification methods (such as [8, 9]) may be applicable. Figure 2 shows how the work fits together: 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. Other implementation routes may be the subject of future work, such as distributed implementations or kernels based on other scheduling mechanisms. One particularly interesting piece of future work would be the integration of existing formal methods for developing sequential code (such as Z [11] or VDM [12]) with AORTA, so that the timing of a system and its functional correctness could be verified in a unified way.

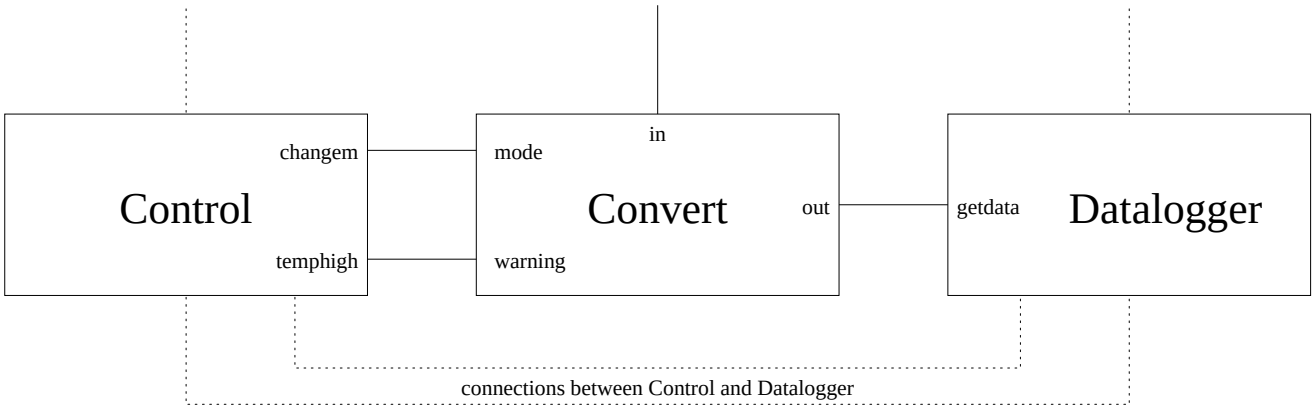

Figure 1: Connectivity of `Tempsys`

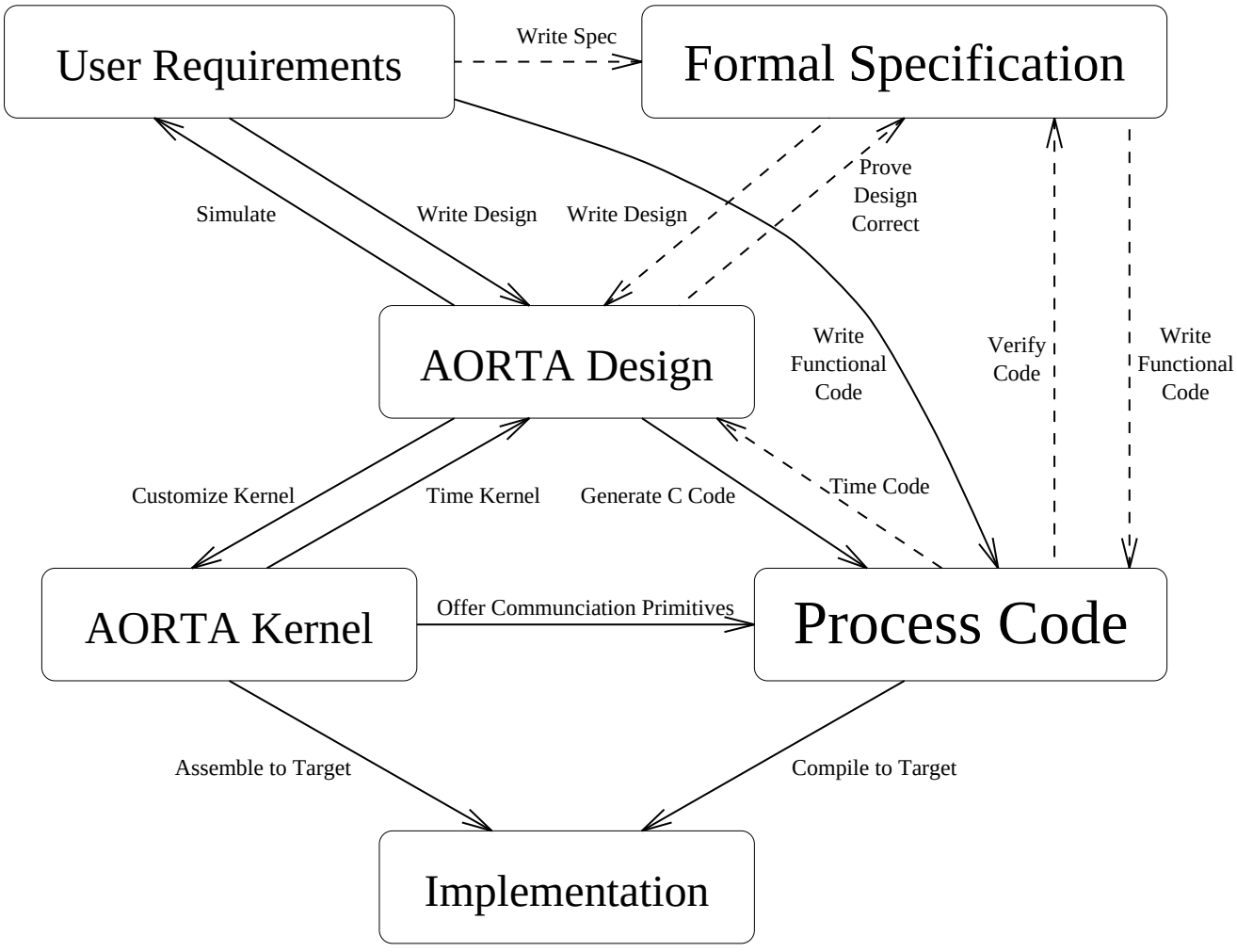

Figure 2: AORTA within a real-time design methodology

Acknowledgements

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

References

[1] 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.

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

[3] 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.

[4] A Valenzano, R Sisto, and L Ciminiera. Rapid prototyping of protocols from LOTOS specifications. *Software — Practice and Experience*, 23(1):31–54, January 1993.

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

[6] S Bradley, W Henderson, D Kendall, and A Robson. An Application Oriented Real-Time Algebra. Technical Report NPC-TRS-93-3, Department of Computing, University of Northumbria, UK, 1993.

[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.

[8] 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.

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

[10] 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.

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

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