Integrating AORTA with Model-Based Data Specification Languages

Steven Bradley[*], William Henderson[†], David Kendall[†], Adrian Robson[†]

October 20, 1997

Abstract

AORTA has been proposed as an implementable real-time algebra for concurrent systems where event times, rather than values of data, are critical. In this paper we discuss an extension to AORTA to include a formal data model, allowing integration with a variety of model-based data specification languages. An example is given using VDM with AORTA to define a time-critical system with important data attributes, and supporting software tools for AORTA and a simple imperative language are described.

1 Introduction

Although many timed formalisms exist, AORTA [6] (Application-Oriented Real-Time Algebra) is one of the few to consider how designs/specifications of concurrent systems can be implemented in a way that time behaviour can be guaranteed. Supporting tools exist which allow AORTA designs to be simulated, formally verified, and code to be generated [8]. One of the ideas behind the development of AORTA has been that formal methods are good for more than just proof: an unambiguously defined semantics allows early exercising of designs by simulation, and provides a basis for reliable code generation. Whilst proof remains an important aspect of any formal technique, we argue that it is not only the presentation of sound and complete proof theories or automatic verification algorithms which should influence the design of languages, but also the provision of facilities such as code generation and simulation.

AORTA only models formally the order and timing of events, and does not deal with data. Implementation details such as values to be passed during communication and the data transformations to be carried out during a given piece of computation are given as annotations to the AORTA design, in the form of fragments of C [5]. In this paper we examine the problem of introducing formal models of data into AORTA designs, and how this affects the notation, the semantics, the tool support and the development method. The approach given here is different from some other proposals [13, 20, 23, 25], in that rather than integrating with a particular formal specification language, integration within a relatively general framework (described in section 3) is suggested, which allows instantiations with model-based languages such as VDM [18] or Z [21], or with formally defined imperative languages. The formally specified data properties are given as annotations to the basic AORTA design (section 4), in much the same way that fragments of C code are, except that we give a formal semantics to the arrangement (section 5). An example using VDM and AORTA is given in section 6, and tool support for designs in a joint language is discussed in section 7, along with some methodological considerations. Finally, our conclusions are presented in section 8. First of all, though, we introduce the basic language of AORTA.

2 Background to AORTA

AORTA is a timed process algebra which can be used as a design language for communicating concurrent real-time systems. Its main novelty lies in its (semi-automatic) implementability, which is discussed in detail elsewhere [6]. A system is defined as a static parallel composition of processes, linked by explicit communication channels. In its description of processes, AORTA inherits some notation from CCS [19], but other ideas, such as communication channels, are borrowed from elsewhere. Within a (sequential) process, actions can be offered, which must be matched by a communicating partner before the process can proceed, and a choice may be offered between a number of actions. As in CCS, action prefix and choice (sometimes called summation) are represented by `.` and `+` respectively, with `0` for the null process which offers no actions. Recursion can be written using the same equational format as used in CCS (e.g. `A = a.A`), but all recursion must be guarded (i.e. all process names must appear inside an action prefix). The other constructs do not have analogues in CCS, and are concerned with including time information into the process description.

There are two constructs which are used to introduce time, and each of these has a deterministic and nondeterministic form. The first construct is a delay which causes the process to pause for the amount of time specified, during which time no actions are offered — time consuming operations such as computation are represented in this way. As precise times are not always known, the delay may be specified with an upper and lower bound, rather than a precise figure. A process which delays for precisely t time units before behaving like S is written `[t]S`, and if the delay is bounded by times $t1$ and $t2$ the process is written `[t1,t2]S`. The second construct is a 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 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`. As data is not handled by the basic language of AORTA, a data-dependent branch is modelled as a nondeterministic choice between processes. Such a choice is written `P++Q`, and is similar to the nondeterministic choice $P \sqcap Q$ of CSP [17].

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 composition of system from its component processes is kept separate from process definition in AORTA.

Parallel composition of processes in AORTA is defined statically, by listing the names of the processes, with `|` as a separator. Internal communication channels are also defined statically by giving the *connection set*, which lists pairs of gates of processes. Each gate may be connected only once, and a gate may not be connected to another gate of the same process. The parallel composition and connectivity within a system is easily represented graphically. A small example demonstrates most easily how process and system definition works in practice.

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

Table 1: Summary of AORTA sequential process syntax

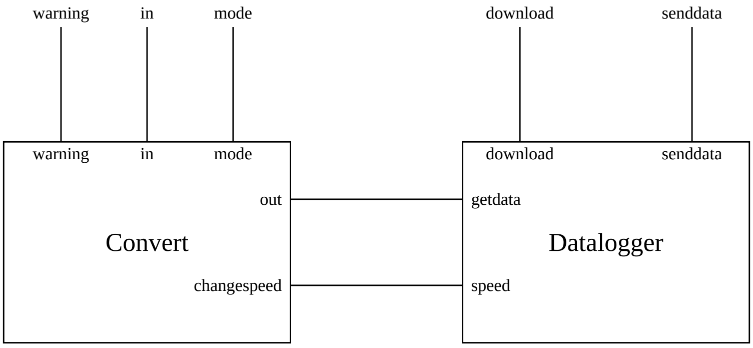

Figure 1: Connectivity of the Chemical Plant Controller

2.1 A Chemical Plant Controller Example

In this section we introduce a semi-realistic example, based on a chemical plant controller. The controller has to monitor and log temperatures within a 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 a 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. This system is described in more detail in [4], and is extended in section 6 to include data information. More complex examples have also been defined in AORTA, including a car cruise controller [6] and a parallel development of part of an industrial submersible controller [5].

The design presented here involves two processes, one of which handles the actual conversion of the data, while the other is used to log the data, and to control the rate at which data is sampled. There are two internal connections, which are used to pass the converted data value, and to indicate a change in the required sampling rate. The graphical representation of this system is shown in figure 1.

The first of the two processes, `Convert`, accepts raw data on the gate `in`, and compares it with a threshold value. Depending on this comparison, the data conversion either takes place straight away, or is preceded by a warning signal. During the actual conversion, which takes place in the `Convert2` part of the process, the calculation is done, and the result offered at the `out` gate, for connection to the `Datalogger` process. This output is timed out, to ensure that fresh data is always available, and that dangerously high input values are noticed within a reasonable time. As well as accepting data input, the `Convert` process allows the conversion mode to be changed, which in this case involves a signal to `Datalogger`, and the recalculation of a lookup table. Again, if no communication is available with the `Datalogger` process within about 1.5 seconds, control is returned to the main sampling loop. Changing mode during conversion is excluded by the choice (+) between `in` and `mode`.

```
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
```

The `Datalogger` process is fairly simple. Data is accepted on the `getdata` gate, which is then stored (requiring a computation delay). The normal sampling loop is driven by a time-out, which regularly requests new data. The period of this loop depends on the current mode of operation (it is either about 1.0 seconds or about 0.25 seconds), and this mode of operation can be changed by a `speed` message from the `Convert` process. As well as accepting mode change commands, the `Datalogger` process accepts requests for the downloading of the current data set to an external machine. In this case, the packet is constructed, and sent out via the `senddata` gate. This may take a considerable period of time, depending on the size of the packet and the nature of the communication link, and is represented by the communication delay associated with `senddata` in the connection set.

```
Datalogger = getdata.[0.001,0.015]
                (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
```

Having defined the individual processes, the full system is defined by the processes which run in parallel, along with connections, both internal and external. As well as providing a textual format for the data presented in Fig 1, communication delays are also associated with each communication channel.

```
(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)>
```

3 Data Model Assumptions

Having described the basic language of AORTA, we can now describe the extensions to handle data. There are two main types of functional specification languages: model-based (such as Z [21], VDM-SL [18] and B [1]) and algebraic (such as ACT

ONE [20] and OBJ [15]). In a model-based language, an abstract formal model of the data in the system is built, and operations are specified and described as transformations on that model. An algebraic approach does not require a complete model to be built, and operations are specified only in terms of each other. These two approaches are not entirely incompatible, as model-based specifications can be written in an algebraic style, and models can be built into an algebraic specification, but for the purposes of this paper the distinction is important. We have chosen to use model-based languages for the pragmatic reason that algebraic languages tend to be biased towards implementation using functional languages, and it is much more difficult to predict the real-time performance of a functional program than an imperative one. There are no pressing reasons for choosing one model-based language over another in our model, and in particular this work is equally applicable to Z, VDM, B, and formally defined imperative languages. Rather than choosing one of these languages arbitrarily, a general presentation is given here. Bowen and Hinchey, in their 'Ten Commandments of Formal Methods' [3] make the point that in industrial application of formal methods it is important to fit in with existing working practices. This point can be extended to the integration of formal methods, where integration with a variety of formal methods has the advantage that as little as possible extra effort has to be made in learning new notations. Therefore we feel that the loose coupling of AORTA with model-based specification languages, rather than a particular language, is a strong point. The specifics of how VDM can be used with AORTA are given with an example in section 6, and tool support for AORTA with a simple imperative language is described in section 7.

We now describe a fairly general framework for the description of model-based languages, and explain our assumptions. The basic model is that each process has a set of possible states, *States*, over which the variable Φ may range. The state Φ includes evaluations for a set of state variables. Each variable A has a set of values over which it may range, given by $values(A)$. Variables can be read using a projection $\Phi.A$, and may be updated using the standard notation $\Phi[A = v]$ where A is a variable name and $v \in values(A)$. Operations are represented using a three-place relation on states, so an operation Δ which can act on state Φ to give state Φ' is written

$$\Phi \overset{\Delta}{\Longrightarrow} \Phi'$$

The operation which changes nothing is then the identity relation on states Ξ, where

$$\Phi \overset{\Xi}{\Longrightarrow} \Phi$$

As well as accessing individual variables and performing operations on states, decisions have to be made based on the data state, which requires the definition of predicates on states, written $p(\Phi)$. Finally, we will need two distinguished state variables: $\mathcal{A}$, with $values(\mathcal{A}) = None = \{\perp\}$, and $\mathcal{T}$, with $values(\mathcal{T})$ as the time domain in use (positive reals or natural numbers).

4 Extension of Syntax

According to [6], the abstract syntax for AORTA sequential expressions is

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

where t, t_1 and t_2 ($t_1 < t_2$) are time values taken from the time domain (either the positive reals or the naturals), a_i are gate names, and X is taken from a set of process names used for recursion. A system expression, is written as a product of processes with a connection set K

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

On the whole, the translation from concrete syntax to abstract syntax is straightforward, but some restrictions are imposed. Choice, with or without time-out, can only take place between communication events, otherwise parallel execution of computation within a single process is required, or a counterintuitive form of time nondeterminism must be adopted. $\sum$ is used to represent choice, and $\triangleright$ for time-outs. The syntax is extended for each of these constructs apart from recursion, so rather than give the whole new syntax at once, the extensions are dealt with in turn.

4.1 Communication

In the original abstract syntax, communication (and its extensions to choice and time-out) uses only gate names, reflecting the pure synchronisation model of the semantics [6]. Extending communication to include value-passing can be achieved by associating a different gate name with each data value to be offered or received (see [19]). While attractive from a theoretical point of view, as this requires only a little syntactic sugaring, it does raise some practical difficulties in implementation. Also, the abstract specification of data state transformation via computation is difficult to incorporate into this model.

The approach adopted here is more akin to that adopted by LOTOS, with its inclusion of the ACT ONE data language for value-passing [20]. Variable names can be attached to communications as input or output parameters, using a question mark for input and an exclamation mark for output. If a value is to be read from gate a into variable A, this is written $a?A.S$, and if the value held in the variable B is to be output on gate a, this is written $a!B.S$. In the general case a gate may have input and output, written $a?A!B.S$, so the abstract syntax form for choice is

$$\sum_{i \in I} a_i?A_i!B_i.S_i$$

If no data is associated with a communication then the input and output variables are both given as the distinguished variable $\mathcal{A}$ (which always has value $\perp$), so that $a.S$ is an abbreviation for $a?\mathcal{A}!\mathcal{A}.S$. Similarly, $a?B.S$ is an abbreviation for $a?B!\mathcal{A}.S$ and $a!B.S$ is an abbreviation for $a?\mathcal{A}!B.S$. The variable $\mathcal{T}$ is used to represent a perfect clock, and so cannot be used as a communication variable. Communications within a time-out are adapted in exactly the same way as for choice, giving the abstract syntax forms

$$\sum_{i \in I} a_i?A_i!B_i.S_i \triangleright_{t_1}^{t_2} S$$

and a corresponding deterministic form.

4.2 Computation

Within AORTA, computations are represented only by a time delay, but during such delays a change of data state will usually take place. Operations which change state are represented by transformation functions $\varDelta$, which are attached to the time delay construct using braces. If an operation $\varDelta$ takes between t_1 and t_2 time units to complete, this is represented by the abstract syntax form

$$[t_1, t_2\{\varDelta\}]S$$

Some computations will require access to a real time clock, for time-stamping or time-averaging, so a special state variable $\mathcal{T}$ is used to represent a perfect clock. In practice, a physical clock will not be perfect, as it may run at the wrong speed, and may have its values discretised. This is modelled by defining a physical clock function on the perfect clock, which gives a set of values related to the perfect clock within some level of accuracy. During computations, time can only be accessed via the physical clock function.

4.3 Data dependent choice

Data dependent choice is represented as nondeterministic choice in AORTA, using the $\bigoplus_{i \in I} S_i$ notation. In order to give the conditions under which each branch of the choice is to be taken, a predicate on the state is attached to each, again using braces

$$\bigoplus_{i \in I} S_i\{p_i\}$$

Sometimes a degree of nondeterminism is helpful, so the predicates are allowed to overlap (i.e. there can be j and k such that $p_j^{-1}(true) \cap p_k^{-1}(true)$ is nonempty). There must, however, always be one predicate which is true (i.e. $\forall \Phi . \bigvee_{i \in I} p_i(\Phi)$), to ensure that some branch will be taken up.

Combining the extensions for communication, computation and data dependent choice gives the full abstract syntax for AORTA terms with data information

$$\begin{array}{rcl} S & ::= & \sum\limits_{i \in I} a_i?A_i!B_i.S_i \ \mid \ [t\{\Delta\}]S \ \mid \ \sum\limits_{i \in I} a_i?A_i!B_i.S_i \rhd^t S \\ & \mid & [t_1, t_2\{\Delta\}]S \ \mid \ \sum\limits_{i \in I} a_i?A_i!B_i.S_i \rhd_{t_1}^{t_2} S \ \mid \ \bigoplus\limits_{i \in I} S_i\{p_i\} \ \mid \ X \end{array}$$

where t, t_1 and t_2 ($t_1 < t_2$) are time values taken from the time domain (either the positive reals or the naturals), A_i and B_i are state variable names, Δ is a state transformation function, the p_i are predicates on the state, and X is taken from a set of process names used for recursion.

5 Enriched Semantics for AORTA

The semantics defined in [6] gives a stratified set of operational transition rules for defining a transition relation between AORTA terms. A similar approach is adopted here, except that the transition system is enriched with the data state. An interleaving semantic model is used, with time transitions represented by $\xrightarrow{(t)}$ and action transitions (i.e. communications) represented by $\xrightarrow{a}$. A transition system *stratification* is a technique whereby transition rules with negative premises can be meaningfully included. By evaluating the transition system in layers, or strata, it can be shown that no transition's validity depends on its own negation, as circularities can be removed [16]. In our system, the lowest stratum contains transitions between sequential expressions, the second contains all internal system communications, and the third (and highest) contains system time transitions and external communications. By organising the transition system in this way, the negative premise for the system time delay rule given below can be consistently incorporated. This negative premise is essential to enforce maximal progress, or τ-urgency.

To define the first stratum, we have to consider an important subset of sequential expressions, known as the *regular* expressions, on which the semantics is defined (n.b. the semantics is undefined on non-regular expressions). Regular expressions

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S[\![\Phi]\!] \xrightarrow{(t)} S[\![\Phi_{+t}]\!]}$$

$$\frac{}{\sum_{i\in I} a_i.S_i \rhd^t S[\![\Phi]\!] \overset{a_j?v!\Phi.B_j}{\longrightarrow} [t\{\Xi\}]S'_j[\![\Phi[A_j = v]]\!]} \quad \begin{array}{l} j \in I \\ S'_j \in Poss(S_j, \Phi[A_j = v]) \\ t \in delays(a_j) \\ v \in values(A_j) \end{array}$$

Figure 2: Transition rules for sequential expressions with data

have no nondeterminism or recursion before the next action, and can easily be syntactically characterised. A regular sequential expression is annotated with a data state Φ, written $S[\![\Phi]\!]$, and a set of eight sequential expression transition rules (which are defined only on regular expressions) can be given. The full set of rules can be found elsewhere [7], but two example rules are given in figure 2, where we abbreviate the updating of the perfect clock using the the definition $\Phi_{+t} \triangleq \Phi[\mathcal{T} = \Phi.\mathcal{T} + t]$ which changes the state only by adding t to the perfect clock variable. The semantics of data dependent choice is not given by transition rules, but by the definition of the *Poss* function. Any AORTA term which starts with $\bigoplus_{i\in I} S_i$ is not regular, so has to be regularised when an action transition takes place. Without any data state information, the choice between branches is nondeterministic, but by attaching predicates to the branches, a data dependent choice can be made. The *Poss* function defines possible resolutions of nondeterminism which are used to regularise a process; again details can be found elsewhere [7]. There are three rules for system expressions, based on the transitions of sequential expressions, for internal communication, external communication, and time progress. The rule for time progress is

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

The negative premise $\overset{\tau}{\not\rightarrow}$ is used here to enforce the maximum progress principle, and a simple priority on communication — internal communication is preferred to external communication. A more sophisticated prioritisation can be achieved by making each communication dependent on all higher priority communications being impossible. To retain the consistency of the transition system, a more complex stratification must be used, with a different stratum for each priority level. The lowest priority level will always be for the time delay, so as to enforce the maximum progress principle. Within the rule for time progress, the function *Age* is used to represent the a process after a given amount of time has passed. More formally, we define

$$Age(E, t) = E' \;\Leftrightarrow\; E \xrightarrow{(t)} E'$$

In [6] a direct syntactic interpretation of *Age* is given, along with a theorem relating it to the definition just given, which indirectly demonstrates that *Age* is well-defined (i.e. it is a function).

6 An Example Using VDM

The chemical plant controller example of [4] is given here as an example of how data specifications can be built into AORTA. VDM is used as the specification language here, although other languages can equally well be used. Addressing the data model assumptions given in section 3 in turn, we first have to consider how the set of possible states of a process can be defined. In VDM this can be done by defining a composite type, including fields for each of the state variables of the process (including $\mathcal{A}$ and $\mathcal{T}$). Invariants on the datatype can be used to restrict the state space. The set of values for each state variable is defined by its type. Selectors are used to provide projections for individual variables, and the μ function gives an easy mechanism for updating:

$$\Phi[A = v] = \mu(\Phi, A \mapsto v)$$

Operations are simply VDM operations which take no argument and return no result, but have the process state as a writable external, and no (i.e **true**) precondition. The identity function on states Ξ is simply the operation

ID

ext wr s : *States*

post $s = \overleftharpoonup{s}$

Finally, predicates on states are defined simply as boolean valued functions on states (i.e. of type $States \to \mathbf{B}$).

To construct the set of (data) states for the `Convert` process, we use five state variables, including the perfect clock $\mathcal{T}$ and the dummy $\mathcal{A}$. There are two gates of the `Convert` process which carry data, namely `in` and `out`: the state variables associated with these gates are *input:Rawdata* and *output:Temp* respectively. A lookup table is used for the conversion, and this is stored in the state variable *table:Lookuptable*. With the time domain represented as the type *Time*, the composite type representing the state of `Convert` is given by

$$\begin{array}{lrcl} Convert :: & input & : & Rawdata \\ & output & : & Temp \\ & table & : & Lookuptable \\ & \mathcal{T} & : & Time \\ & \mathcal{A} & : & None \end{array}$$

Within `Convert`, there are two computations: the first converts raw data to a temperature, using a lookup table, and the second recalculates the lookup table for a different conversion mode. Assuming that we have the function *evaluate* then the conversion operation is defined as

DOCONVERSION

ext wr $conv$: *Convert*

post $conv = \mu(\overleftharpoonup{conv}, output \mapsto evaluate(\overleftharpoonup{input}, \overleftharpoonup{table}))$

Changing conversion mode depends on a function *newtable* which recalculates the lookup table, so the operation for changing mode is defined as

CHANGEMODE

ext wr $conv$: *Convert*

post $conv = \mu(\overleftharpoonup{conv}, table \mapsto newtable(\overleftharpoonup{table}))$

To specify the behaviour of nondeterministic choice, a predicate on the state must be attached to each branch of the choice. In the `Convert` process, the behaviour depends on whether the raw data value exceeds a threshold value; if so a warning signal must be sent. The predicates which we are interested in are

$$convertdatahigh : Convert \rightarrow \mathbf{B}$$

$$convertdatahigh(conv) \quad \triangleq \quad input(conv) > threshold$$

and a corresponding predicate *convertdataok.* which assume that we have defined a total order $>$ on *Rawdata* and that the value *threshold:Rawdata* is defined. Attaching these new data constructs to the `Convert` process gives the definition

```
Convert = in?input.
            (Convert2 {convertdataok} ++
             warning.Convert2 {convertdatahigh})
           +
           mode.(changespeed.
                 [0.3,0.4 {CHANGEMODE}]Convert)[1.5,1.505>Convert
Convert2 = [0.001,0.004 {DOCONVERSION}]
            (out!output.Convert)[1.5,1.505>Convert
```

The `Datalogger` process has its own set of states, defined by the composite type

$$\begin{array}{lrcl} Datalogger :: & input & : & Temp \\ & packet & : & Loggerpacket \\ & history & : & (Temp \times Time)^* \\ & T & : & Time \\ & \mathcal{T} & : & Time \\ & \mathcal{A} & : & None \end{array}$$

Two of the variables, *input* and *packet* are used to carry data for communication on gates `getdata` and `senddata`, while *history* is used to record data with time stamps. The variable T is used for the physical clock, as well as the usual $\mathcal{T}$ and $\mathcal{A}$ variables. Two computations are associated with `Datalogger`, which correspond to adding a data item (with time stamp) to the store, and making up a data packet for downloading. To get the time stamp value from the clock, we require the function *possclocks* which returns the possible physical clock values at a given time. The data which is input from the `getdata` port is added to *history* with the operation

$ADDDATA$

ext wr $mk\text{-}Datalogger(h, i, p, t1, t2, a) \; : \; Datalogger$

post $t1 \in possclocks(t2) \wedge h = cons((t1, i), \overleftarrow{h}) \wedge t2 = \overleftarrow{t2}$

Finally, assuming the function *makepacket* we can define the operation

$MAKEPACKET$

ext wr $mk\text{-}Datalogger(i, p, h, t1, t2, a) \; : \; Datalogger$

post $p = makepacket(\overleftarrow{h}) \wedge h = [\,] \wedge t2 = \overleftarrow{t2}$

There are no nondeterministic choices in the `Datalogger` process, so the full version of the process, including data information, is

```
Datalogger = getdata?input.[0.01,0.015 {ADDDATA}]
                (speed.Datalogger2
                 +
                 download.[0.5,1.0 {MAKEPACKET}]
                         senddata!packet.Datalogger)
                  [1.00,1.005>Datalogger
Datalogger2 = getdata?input.[0.01,0.015 {ADDDATA}]
                (speed.Datalogger
+
download.[0.5,1.0 {MAKEPACKET}]
        senddata!packet.Datalogger2)
[0.25,0.255>Datalogger2
```

Having defined the individual processes, the system composition is given as before, using the | operator and a connection set, but with the addition of initial data states for each of the processes within the parallel composition.

7 Tool Support and Methodological Considerations

The emphasis of AORTA is on practicality, in that implementation and simulation issues have been considered alongside verification; designs written in the language can be represented purely in ASCII; implementations are based on generated C. One crucial aspect of a practical design method is the availability of supporting software tools, and research tools for graphical simulation, model-checking via graph generation, and code generation have been provided. These were all originally written for the basic language without a formal data model, where all computational aspects were represented by implementation fragments written in C.

In order to provide support for AORTA extended with a formal data model, some generalisation of the tool set was required. One possible approach would have been to choose a formal language for data, such as VDM, and to attempt a one-off integration of the AORTA tool set with some supporting tools for the data language. This would have the advantage that it might not require too much work, and could provide a fairly tight coupling, but would have the obvious disadvantages of inapplicability to other languages and tools. Instead a more general approach was adopted, whereby an abstract data language interface was specified, (based on the data model assumptions given in section 3) and the integration done at that level. In this way, integration with a new language or tool set involves providing an interpretation of the abstract notions of value, variable, state, computation, predicate and so on. The obvious advantage of this approach is in its flexibility, with the disadvantages that the tools which are to be integrated may need to be adapted to fit the interface provided.

The actual support which is provided for the data enriched language mostly falls into the area of simulation, which we introduced in section 1 as an important part of a formal method. For the basic language the tool set offers simulation as a technique for exercising the semantics, by choosing time and action transitions from a menu. Although this is helpful for a detailed exploration of the behaviour of a design, the more complete description given by a design with data allows a more dynamic simulation to be offered as well; one in which the processes of the design are simulated by concurrently executing threads, which communicate and evolve spontaneously in time. Put another way, we can now provide a direct interpreter for the combined language. The new support provides such a simulator, which allows any AORTA design annotated with formally specified data operations to be executed. Implementation code is provided as a separate annotation to the design, so that if the data formalism is supported by code generation, then the whole of the design (including data parts) can be used to generate complete implementation code directly.

Our initial experiment into providing a formal data language has used a simple formally defined imperative language with sequence, choice and iteration, and integer, boolean and enumerated data types. This language is substantially smaller than VDM, for example, but serves to demonstrate that a useful integration is possible. Furthermore, as the computation data relation is a function, direct interpretation is possible, and the language is explicit enough to allow direct code generation. In fact, this is just the sort of language that formal refinements from Z, B or VDM aim to produce, so it may be that two levels of data formalism should be provided: one for an abstract, possibly implicit, specification, and one for an explicit description, closely related to an implementation, and derived by a verified refinement from the specification. However, some approaches, such as that adopted by the IFAD VDM-Toolbox [11] are based on writing explicit specifications in the first place, and hence providing code generation and interpretation facilities directly. In such cases as these, direct integration with AORTA is possible, without the need for an intermediate language.

The discussion about whether implicit specification and refinement, or explicit specification and code generation is better is outside the scope of this paper, but we note that in order to satisfy our earlier criterion of integration with as wide a range of approaches as possible, we should be able to deal with both. This is possible because of a further level of generality built in to the tool support for AORTA, beyond that of a general data language. Not only is the actual type of data language with which designs can be annotated quite general, but the number and type of annotations themselves is general. For instance, for AORTA with the simple imperative language, annotations can be provided at each point in the syntax tree for the textual form of the data part, for its internal representation as a relation on states, or whatever, for the implementation code associated with it, and for information concerning the graphical presentation of the syntax. However, the notion of annotation is general, and the implementation of the tool set modularised such that the addition of new annotations, perhaps for a more abstract data specification, or perhaps for proofs of correctness, or perhaps for timing information about the code, is quite straightforward. Having provided different kinds of annotation, the tool then needs to be configured to say which will be used in code generation, which are to be used in simulation, and which in verification etc.

How then are such tools and languages to be used to develop systems? We suggest that early simulation is important, as it allows problems in the design to be detected before too much of the implementation detail is fixed. Similar arguments are given for the early application of specification and proof techniques during system development. The aim of this work is not necessarily to replace proof in system development, but rather to avoid wasted effort during proof by detecting and eliminating as many errors as possible by simulation, which can be thought of as high-level testing. With the addition of an interpretation for data two kinds of simulation are now possible. In the first, in which the processes evolve spontaneously, a design error may be detected and corrected immediately, or further, more detailed simulation, based on the semantics, may be required to locate the problem. Having satisfactorily tested the design, it may at this point be appropriate to attempt a formal correctness proof. Note that further work is required on proof techniques in a combined language (see section 8). Having verified the correctness of the design, further work will be required to produce the implementation. If code generation of data properties is not automatic then refinement to code, with proofs, will be required. Also, static analysis of code (possibly with user intervention) to extract timing information will we required, as inputs to the scheduling calculations, which are used to verify that the implementation timing will match that given in the design [4].

8 Conclusions

AORTA is a timed process algebra-based design language, so comparison might be made with other timed process algebras; however so many timed process algebras have been defined that even a cursory list of references would be too long for the scope of this paper, so the reader is referred elsewhere [9], and direct references given only for (a version of) Timed CCS [26], Timed CSP [22], and (a version of) Timed LOTOS [2]. At this level the main distinctive feature of AORTA is the ability to generate implementations about which timing guarantees can be made.

This paper has shown how it is possible to build a formal data model into AORTA and how tool support for simulation and implementation generation techniques and tools can be extended. Further work needs to be done on the use of model-checking techniques in association with data properties. One possible approach is to provide a (verified) refinement of the data associated with the state spaces, so that required data properties still hold, but that the state space is finite. Once the state space has been reduced to a finite size, data properties can be represented as propositions labelling timed graphs, so that model-checking of properties like 'The alarm will come on within 5 seconds of receiving a temperature reading above the safe limit' becomes possible. The abstraction to the trivial state space where all data information is ignored has been shown to be equivalent to the original semantics [9], so we can at least still perform simple model-checking with assurance of correctness.

Other research has covered some of the aspects of this work. MOSCA provides a formalism combining CCS, VDM and time, but without providing implementation techniques [25] whilst RAISE [24] and LOTOS [20, 27] provide data modelling in concurrent systems, with some implementation techniques, but no time. Work has also been done with timed extensions to LOTOS [2], which already has the data language ACT-ONE included, but in this case no implementation techniques are provided. A different kind of approach involves introducing time into data specification languages such as Z [10, 13, 14], with the closest work to ours being that by Fidge et al [12], which allows the timed refinement of concurrent systems, including reasoning about implementations by embedding scheduling theory into the Z model. This approach can only be described as 'different' to ours, with the relative merits and demerits associated with the two being the usual ones associated with refinement as opposed to code generation techniques. Also, most of their work has been associated with providing the proof theory (as would be expected for a refinement calculus), whereas our work has focussed on implementation aspects.

In summary, then, this paper has shown how a fairly general formal data model can be integrated syntactically and semantically into AORTA. Tool support for simulation and code generation has been discussed, and an example of using AORTA with VDM has been included. Proof support needs further work, although some suggestions have been made, so some may raise the question as to what purpose a formal semantics serves where no proof support is to be offered. In our introduction, we argued that formal methods and good for more than just proof, and we feel that this has been borne out by the provision of useful simulation tools, and also a clear statement of the necessary assumptions about the data model, which have formed the basis of tool support for the data-enriched language.

References

[1] J-R Abrial. *The B-Book.* Cambridge University Press, 1996.

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

[3] J P Bowen and M G Hinchey. Ten commandments of formal methods. *IEEE Software*, 28(4):56–63, April 1995.

[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 Henderson, D Kendall, A Robson, and S Hawkes. A formal design and implementation method for real-time embedded systems. In P Milligan and K Kuchinski, editors, *22nd EUROMICRO Conference (EUROMICRO 96), Prague*, pages 77–84. IEEE, September 1996.

[6] S Bradley, W D Henderson, D Kendall, and A P Robson. Application-Oriented Real-Time Algebra. *Software Engineering Journal*, 9(5):201–212, September 1994.

[7] S Bradley, W D Henderson, D Kendall, and A P Robson. Modelling data in a real-time algebra. Technical Report NPC-TRS-95-1, Department of Computing, University of Northumbria, UK, 1995.

[8] S Bradley, W D Henderson, D Kendall, and A P Robson. Validation, verification and implementation of timed protocols using AORTA. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV (PSTV '95), Warsaw*, pages 193–208. IFIP, North Holland, June 1995.

[9] S P Bradley. *An Implementable Formal Language for Hard Real-Time Systems.* PhD thesis, Department of Computing, University of Northumbria, UK, October 1995.

[10] A Coombes and J McDermid. Specifying temporal requirements for distributed real-time systems in Z. *Software Engineering Journal*, 8(5):273–283, September 1993.

[11] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[12] C Fidge, M Utting, P Kearney, and I Hayes. Integrating real-time scheduling theory and program refinement. In M-C Gaudel and J Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods*, number 1051 in Lecture Notes in Computer Science, pages 327–346. FME, Springer, 1996.

[13] C J Fidge. Specification and verification of real-time behaviour using Z and RTL. In J Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems Second international symposium, Nijmegen, Lecture Notes in Computer Science 571*, pages 393–409. Springer-Verlag, 1992.

[14] C J Fidge. Real-time refinement. In J C P Woodcock and P G Larsen, editors, *Formal Methods Europe '93: Industrial-Strength Formal Methods, Lecture Notes in Computer Science 670*, pages 314–331. Springer-Verlag, 1993.

[15] J A Goguen and T Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI, August 1988.

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

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

[18] C B Jones. *Systematic software development using VDM.* Prentice Hall, New York, 1986.

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

[20] 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.

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

[22] S Schneider, J Davies, D M Jackson, G M Reed, J N Reed, and A W Roscoe. Timed CSP: Theory and practice. In J W de Bakker, C Huizing, W P de Roever, and G Rozenberg, editors, *Real-Time: Theory in Practice (REX workshop), Mook, Lecture Notes in Computer Science 600*, pages 640–675. Springer-Verlag, June 1991.

[23] G Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J Fitzgerald, C B Jones, and P Lucas, editors, *FME 97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. FME, Springer, 1997.

[24] The RAISE Language Group. *The RAISE Specification Language.* BCS Practicioner Series. Prentice Hall, 1992.

[25] H Toetenel. VDM + CCS + Time = MOSCA. In *18th IFAC/IFIP Workshop on Real-Time Programming — WRTP '92, Bruges.* Pergamon Press, June 1992.

[26] C Tofts. Timed concurrent processes. In *Semantics for Concurrency*, pages 281–294, 1990.

[27] M van Sinderen, L Ferreira Pires, and C A Vissers. Protocol design and implementation using formal methods. *Computer Journal*, 35(5):478–491, 1992.