Integrating Real-Time Scheduling Theory and Program Refinement

C. Fidge M. Utting P. Kearney I. Hayes

Software Verification Research Centre, Department of Computer Science,
The University of Queensland, Queensland 4072, Australia.

Abstract. We show how real-time schedulability tests and program refinement rules can be integrated to create a formal development method of practical use to real-time programmers. A computational model for representing task scheduling is developed within a 'timed' refinement calculus. Proven multi-tasking schedulability tests then become available as feasibility checks during system refinement.

1 Introduction

There has long been a gulf between formal methods for specifying and developing real-time programs and the needs of real-time programmers 'in the field'.

- Formal methods for specifying concurrent real-time systems typically make unrealistic simplifying assumptions. In particular, 'maximal parallelism' assumes that each task resides on its own processor and is thus never preempted. This is often justified by pointing to the ever-decreasing cost of hardware.
- Embedded systems programmers, on the other hand, constrained by the realities of power, cost and space limitations, try to implement as many tasks on the same processor as possible. Real-time scheduling theory is used to determine whether a given task set can meet its deadlines.

Consequently real-time programmers find that formal specification and development methods do not model their true concerns. If formal methods are to become a useful industrial tool for real-time programming this gap must be bridged.

In this paper we take a first step towards merging these two previously separate streams of activity by representing the computational model used by scheduling theory in a 'timed' refinement calculus. This makes already-proven schedulability results available as a basis for formal development of multi-tasking programs with hard real-time deadlines.

2 Background

2.1 Review of timed refinement

The *timed refinement calculus* [10, 11] is based on predicate transformer semantics with the specification language Z used as a convenient notation for expressing predicates. To make the calculus suitable for expressing parallel, reactive

behaviours, it replaces the familiar pre/post-condition model with one based on environmental *assumptions* and desired *effects*.

A specification statement

$$+\widetilde{v}: [A, E]$$

has three parts. Let $\widetilde{u}, \ldots, \widetilde{y}$ denote mutually disjoint sets of variables.

1. The set of variables $\widetilde{v}$ in the *frame* denote the observable variables which the specification constructs. Variables in $\widetilde{v}$ may appear in E but not A.
2. The assumption A defines knowledge the specification can use about the environment. It is a predicate on a set of variables $\widetilde{u}$ disjoint from $\widetilde{v}$.
3. The effect E is a predicate on variables in $\widetilde{u}$ and $\widetilde{v}$, typically defining the value of those variables in $\widetilde{v}$ in terms of those in $\widetilde{u}$.

Predicates are ordered in the underlying semantics by an entailment relation $\Rrightarrow$ [10, p.3].

Time is introduced into the calculus by a convention on the types of constructed variables. Each variable $v \in \widetilde{v}$ is actually a trace, or history, of values, one for each moment in time. The trace index is absolute time. For instance, a *timed variable* v of 'base' type V is declared as a function

$$v : \mathbb{A} \rightarrow V \ ,$$

where $\mathbb{A}$ is the absolute time domain. Specifications thus define the values of each variable in the frame over all time.

The refinement relation $\sqsubseteq$ on such specification statements is defined using a handful of fundamental rules. Provisos are shown above the line and the refinement rule below.

Law R1: Weaken assumption [10, p.7]

$$\frac{A_1 \Rrightarrow A_2}{+\widetilde{v}: [A_1, E] \sqsubseteq +\widetilde{v}: [A_2, E]}$$

Law R2: Strengthen effect [10, p.7]

$$\frac{A \Rrightarrow (\forall \widetilde{v} \bullet E_2 \Rightarrow E_1)}{+\widetilde{v}: [A, E_1] \sqsubseteq +\widetilde{v}: [A, E_2]}$$

Law R3: Introduce local constructions [10, p.9]

$$\frac{\text{variables in } \widetilde{x} \text{ are fresh}}{+\widetilde{v}: [A, E] \sqsubseteq |[\ +\widetilde{v} \cup \widetilde{x}: [A\ ,\ [E;\ \widetilde{x}\ ]]\ \backslash \widetilde{x}\ ]|}$$

On the right-hand side the signature of schema E is extended to include declaration $\widetilde{x}$. (For brevity we omit types in these generic definitions.) The $|[\ \cdots \backslash \widetilde{x}\ ]|$ construct declares local variables $\widetilde{x}$ that cannot be seen outside its scope [10, p.9].

Law R4: Introduce parallel composition [11, p.8]

$$\frac{A \Rrightarrow (\exists \widetilde{x};\ \widetilde{y} \bullet E)}{+\widetilde{x} \cup \widetilde{y}{:}[A, E] \sqsubseteq +\widetilde{x}{:}[A \wedge (\exists \widetilde{x} \bullet E), E] \parallel +\widetilde{y}{:}[A \wedge (\exists \widetilde{y} \bullet E), E]}$$

The $\parallel$ operator denotes parallel composition of specification statements [11]. The proviso prevents effect E from accidently strengthening assumption A when added to the assumptions of the two parallel components [10, p.8].

Laws **R1** to **R3** are variants of well-known refinement rules. Law **R4** allows a specification to be partitioned into parallel components where each component can assume properties effected by its siblings, as long as the properties do not reference the variables to be constructed by the component.

Significantly, the parallelism operator $\parallel$ is a 'true' concurrency operator. It is not directly suited to modelling the 'interleaved' concurrency found in uniprocessor multi-tasking applications.

2.2 Review of schedulability testing

Uniprocessor scheduling theory offers proven *schedulability tests* for verifying that a system design, with given timing characteristics, can be successfully scheduled under a particular scheduling policy and communication protocol.

To make analysis of complex real-time systems manageable, the theory uses an abstract computational model [1]. In this model a system consists of a set of *tasks*. Each task i *arrives* infinitely often, each arrival separated from the last by at least T_i time units. A *periodic* task arrives regularly with a separation of exactly T_i time units. A *sporadic* task arrives irregularly with each arrival separated from its predecessor by at least T_i time units.

At each arrival, task i issues a nominal *invocation request* for up to C_i units of processor time, its worst-case *computation time* [1]. (For simplicity the model assumes that only tasks consume time. Scheduling overheads such as context switching and shared resource locking are factored into the worst case computation time for each task.) To complete its workload, task i must have this request for processor time satisfied before some *deadline* D_i expires, measured relative to the arrival time of the task invocation. Usually D_i does not exceed T_i.

The scheduler places each task making a request in a notional *ready queue* [8]. It decides which task in the queue is currently *running* using the *priority* π_i of each ready task i and the particular scheduling policy it implements. In static-priority scheduling there is a fixed *base* priority associated with each task, although a higher *effective* (or *active*) priority may be temporarily allocated to the task at run time. In dynamic-priority scheduling a run-time metric is used to determine priorities during execution. Tasks of higher priority can pre-empt some task i, resulting in a degree of *interference* I_i to the progress of i.

So that communications overheads can be predicted accurately, the model assumes that all inter-task communication occurs through mutually-exclusive access to shared variables. This allows *a priori* knowledge of the worst-case

blocking time B_i that task i may experience due to lower-priority tasks having locks on resources that it wishes to access, for a known locking protocol.

In general, schedulability tests can be divided into two classes, both based on analysis of worst-case scenarios. Tests that measure 'processor utilisation' check a bound on the total percentage of time that all tasks occupy the processor. For instance, the following test applies to a set of tasks under the *earliest deadline first* scheduling policy, using the *stack resource* locking protocol. Earliest deadline first scheduling is a dynamic-priority scheduling policy in which the task with the earliest *absolute* deadline from the current moment has the highest priority. The stack resource protocol [2] guarantees that task invocations begin executing only when all resources they may wish to access are free, and that each task invocation is blocked by a lower-priority task at most once.

Under these conditions a set of tasks $1..n$, ordered by increasing size of their deadlines, is schedulable if [2]

$$\forall i : 1..n \bullet \sum_{j \in 1..i} \frac{C_j}{D_j} + \frac{B_i}{D_i} \leqslant 1 \,. \tag{1}$$

The test checks, for each task i, that the processor utilisation by that task, plus that by higher-priority tasks that may pre-empt i, plus that by lower-priority tasks that may block i, is less than 100%. The first term is the processor utilisation by all tasks j of priority equal or higher than i. The second term is the degree of blocking that may be experienced. Under the stack resource protocol B_i is the execution time of the longest critical section executed by some task of lower priority than i.

Test 1 is a useful feasibility test in general; earliest deadline first scheduling is 'optimal' in the sense that if a task set is schedulable by any policy then it is schedulable by earliest deadline first.

The second class of tests works by precisely characterising system 'response times'. The worst case *response time* R_i for an invocation of task i defines how long it may take the task to complete its computation, measured from its arrival time. For instance, the following test applies to any assignment of base priorities to tasks [1], under a *static-priority* scheduling policy and using the *ceiling locking* protocol. A static-priority scheduler is one which makes the running task the one in the ready queue with the highest effective priority (favouring the currently running task in the case of ties to avoid unnecessary context switching). The ceiling locking protocol [8] is a special case of the stack resource protocol in which (a) each shared variable has an associated 'ceiling' value as great as the highest base priority of any task that may access it, and (b) each task that locks a variable has its effective priority set to that variable's ceiling value.

Under these circumstances, a set of n tasks, with a static assignment of *unique* base priorities, is schedulable if [1]

$$\forall i : 1..n \bullet R_i \leqslant D_i \quad \text{where} \quad R_i = C_i + B_i + I_i \tag{2}$$
$$\text{and} \quad I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \,.$$

Here $hp(i)$ is the set of tasks with higher base priorities than task i. For each task i the test checks that the response time R_i of the task does not exceed its deadline D_i. The worst case response time for i is the sum of its own worst case computation time C_i, plus its worst case blocking time due to lower-priority tasks B_i, plus the worst case interference due to higher-priority tasks I_i. Assuming use of the ceiling locking protocol B_i equals the longest critical section of any lower priority task accessing a shared variable with a ceiling as great as the priority of task i [1].

The interference term I_i determines how much pre-emption task i will experience during the interval of time defined by R_i due to higher-priority tasks j. For each task j this is its execution time C_j multiplied by the number of arrivals $\lceil R_i/T_j \rceil$ that j may have in R_i time units. Thus interference up to time R_i is defined in terms of the number of pre-emptions that may occur during the interval of time defined by R_i: the definition is recursive!

Fortunately, however, the equation can be solved iteratively [1]. Let R_i^x be the x^{th} approximation to the value of R_i. Starting with $R_i^0 = 0$, equation

$$R_i^{x+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^x}{T_j} \right\rceil C_j$$

converges to R_i. Evaluation stops either when the equation has converged, i.e., $R_i^{x+1} = R_i^x$, or, because $R_i^{x+1} \geqslant R_i^x$ for any x, iteration can stop as soon as $R_i^{x+1} > D_i$ in which case the test has failed.

3 A real-time multi-tasking refinement model

Our aim is to represent the computational model used by scheduling theory (Section 2.2) in the timed refinement calculus (Section 2.1) in such a way that proven schedulability results benefit the refinement process. To do this refinements must introduce those computational entities of interest in scheduling theory, namely tasks, protected shared variables and the scheduler itself, in order to capture the behaviour of a multi-tasking system.

This is illustrated in Figure 1. Boxes denote parallel specification components and arcs the flow of information, via the named variables. A top-level specification, defining values of output variables in terms of inputs, is refined to a description known to map to the scheduling behaviour of our target programming language.

The development procedure from top-level specification to multi-tasking system can be described in the following ten steps. A detailed example illustrating each step is given in the next section.

1. Requirements specification. This defines the functional requirement and its absolute time constraints, expressing the 'effect' variables $\widetilde{out}$ in terms of the 'assumption' variables $\widetilde{in}$.

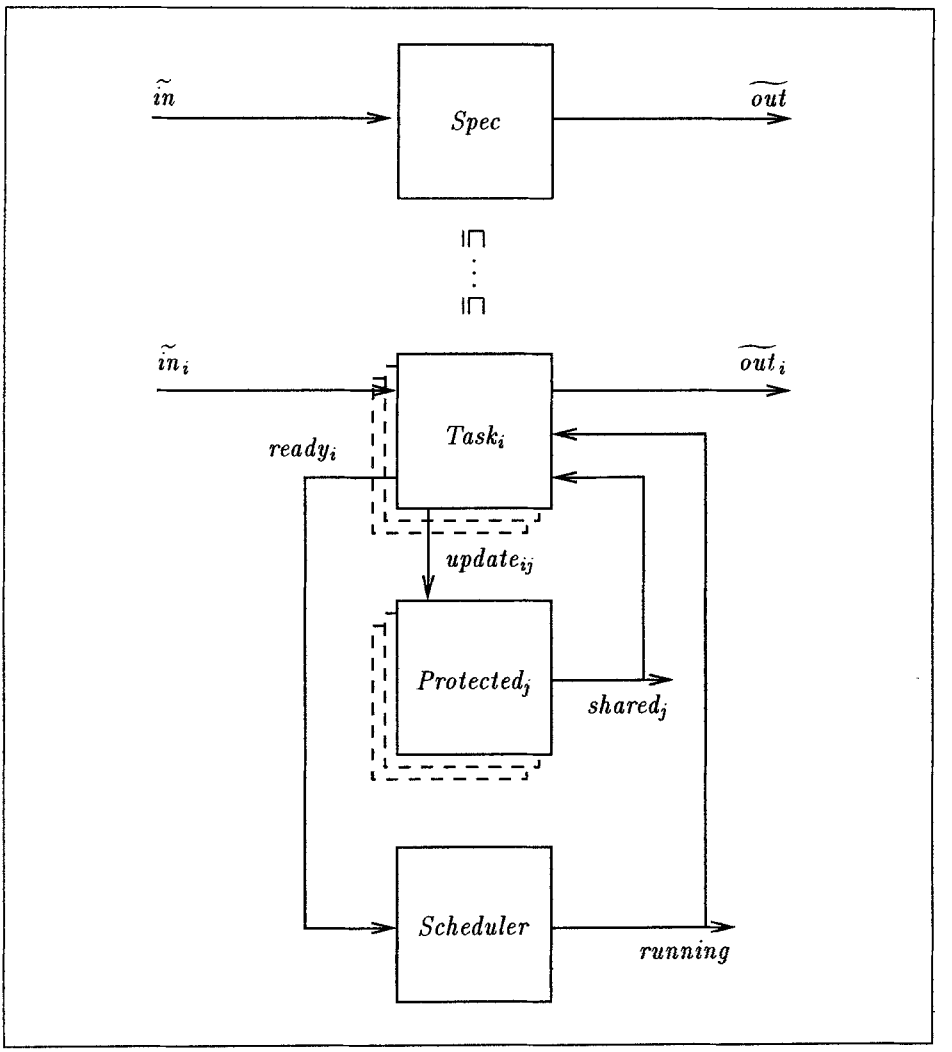

Fig. 1. Overview of the refinement procedure (for static-priority scheduling using the ceiling locking protocol).

2. Introduce shared variables. To allow for later system partitioning, new shared variables $shared_j$ are introduced. Typically this is done so that separate parts of the specification need not have input and output variables in common.
3. Introduce periodic and sporadic requirements. The functional requirements are re-expressed in forms corresponding to the notions of periodic and sporadic tasks. Interarrival times T_i and deadlines D_i are introduced here.
4. Introduce worst case execution times. For each task i a worst case execution time C_i is hypothesised.

5. Introduce feasible schedule. Variable *running*, representing the run-time schedule of executing tasks, is introduced and constrained so that each task i is guaranteed to get sufficient processing time. Each task specification can examine *running* to determine when it is making progress.
   At this point a scheduling *feasibility test* can be undertaken to ensure development is proceeding in a viable direction.
6. Introduce ready indicators. The $ready_i$ variables are introduced so that each task i can indicate its preparedness to run. These variables must ultimately be constrained so that each task i asks for no more than C_i time units.
7. Introduce protected objects. Access to the $shared_j$ variables is restricted so that they are manipulated in a way consistent with the target locking protocol. A protected object specification must accept 'writes' via $update_{ij}$, and allow a task i to 'read' $shared_j$, only when i is known to hold the lock. Worst case blocking times B_i can be expressed at this stage.
8. Introduce scheduler. An abstraction of the scheduling policy, defining *running* in terms of the $ready_i$ indicators, is introduced.
   Sufficient information is now available to perform a full *schedulability test*.
9. Separate tasks, protected objects and scheduler. The specification can now be partitioned into independent $Task_i$, $Protected_j$ and *Scheduler* descriptions. Future refinement of the tasks and protected subroutines can then be performed in isolation.
10. Map to target programming language. The specification should now be in a form known to correspond to a program template in a trusted subset of the target programming language.

As with any development procedure, the exact steps required vary with each application. Not all of the above steps are needed in every case and the order in which the steps are applied may differ. The precise form of the components constructed also varies depending of the target scheduling policy and locking protocol.

For instance, in the example below we target an Ada 95 implementation. Ada 95 supports static-priority scheduling and ceiling priority locking, a combination which is easy to implement. Consequently the *Scheduler* description merely needs to form an imaginary ready queue from the $ready_i$ indicators and use this to set the value of *running* accordingly. Each $Protected_j$ definition is merely a 'merge' function that, when an $update_{ij}$ value appears, sets the value of $shared_j$ to be this new value. However, to allow such simple definitions, each $Task_i$ specification must be suitably well-behaved. The $ready_i$ indicator must always carry the effective priority at which $Task_i$ wishes to run, and i may make computational progress only when *running* indicates that it is executing. Furthermore, $Task_i$ may produce an $update_{ij}$ value, or examine the value of $shared_j$, only when it knows that it is the currently *running* task, and that its $ready_i$ priority is at least as great as the ceiling value for that shared object.

Other target implementations can be handled by our framework, however. For instance, earliest deadline first scheduling could be treated by including the (absolute) deadline of each $Task_i$ invocation in its $ready_i$ request. The *Scheduler* definition can then use these deadlines to determine which task to make *running*.

4 Example

We consider a version of the 'mine shaft' example that has proven to be a popular test-bed for real-time development methods [4]. The system aims to keep the level of ground water seeping into a mine shaft below a certain height as long as atmospheric conditions in the shaft are safe for operating electrical equipment.

4.1 Requirements specification

Firstly we introduce a discrete absolute time domain $\mathbb{A}$ and a type for durations $\mathbb{D}$ of time:

$$\mathbb{A} == \mathbb{N} \qquad \mathbb{D} == \mathbb{N} .$$

Input to the system consists of readings provided by two sensors. A water level sensor continuously provides depth readings.

Water

$$H_2O : \mathbb{A} \rightarrow \mathbb{Z}$$

The system will attempt to keep this reading below a certain mark.

$$H_2Omark : \mathbb{Z}$$

A methane sensor generates 'true' whenever the methane level in the mine becomes unsafe and 'false' whenever it falls back to a safe level again. The known rate of change of methane gas, and the callibration of the sensing equipment, guarantees that such values are generated no closer together than a fixed separation. At start-up time 0 some initial value is generated.

$$CH_4sep : \mathbb{D}$$

$$CH_4sep > 0$$

Methane

$$CH_4 : \mathbb{A} \nrightarrow \mathbb{B}$$

$$0 \in \operatorname{dom} CH_4$$
$$\forall x, y : \operatorname{dom} CH_4 \mid x < y \bullet x + CH_4sep \leqslant y$$

Our goal is to pump the water out of the shaft whenever it becomes too deep. However the pump may run only when methane levels are low, for fear of causing an explosion. Also an alarm bell must ring while methane levels are dangerously high. The pump and alarm actuators are controlled by two variables,

$$Pump \mathrel{\widehat=} [\, pumping : \mathbb{A} \rightarrow \mathbb{B}] \qquad Alarm \mathrel{\widehat=} [\, ringing : \mathbb{A} \rightarrow \mathbb{B}] \ .$$

The system is allowed to take a certain amount of time to react to environmental changes.

$$react : \mathbb{D}$$
$$react \leqslant CH_4sep$$

Lambda expressions provide us with a convenient way to express properties of trace variables such as H_2O and CH_4:

$$\begin{aligned}
deep_\delta &== \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge min\, H_2O(\!|\, t - \delta \,..\, t \,|\!) \geqslant H_2Omark \\
shallow_\delta &== \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge max\, H_2O(\!|\, t - \delta \,..\, t \,|\!) < H_2Omark \\
currCH_4 &== \lambda\, t : \mathbb{A} \bullet CH_4(max\, \mathrm{dom}(0 \,..\, t \lhd CH_4)) \\
danger_\delta &== \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge currCH_4(\!|\, t - \delta \,..\, t \,|\!) = \{\mathrm{true}\} \\
safe_\delta &== \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge currCH_4(\!|\, t - \delta \,..\, t \,|\!) = \{\mathrm{false}\}\ .
\end{aligned}$$

For instance, $shallow_\delta$ is true at some time t only if the water level has been constantly below H_2Omark for the last δ time units. Similarly, $safe_\delta$ is true at t if, for the last δ time units, the most recent CH_4 value was always 'false'. (Function $currCH_4$ returns the most recent CH_4 value at any time t.)

The system is required to raise the alarm when the methane level is high, but not when it is low.

ControlAlarm

$$Methane;\ Alarm$$
$$\begin{aligned}
\forall\, t : \mathbb{A} \bullet\ & (danger_{react}(t) \Rightarrow ringing(t)) \\
& \wedge\ (safe_{react}(t) \Rightarrow \neg\ ringing(t))
\end{aligned}$$

If the methane level changed within the last *react* time units the value of *ringing* is unspecified.

Similarly, the pump must be switched on only when needed, and only when conditions are safe to do so.

ControlPump

$$Water;\ Methane;\ Pump$$
$$\begin{aligned}
\forall\, t : \mathbb{A} \bullet\ & (deep_{react}(t) \wedge safe_{react}(t) \Rightarrow pumping(t)) \\
& \wedge\ (shallow_{react}(t) \vee danger_{react}(t) \Rightarrow \neg\ pumping(t))
\end{aligned}$$

The full requirements specification is then

$$+\, ringing, pumping : \begin{bmatrix} Water & ControlAlarm \\ Methane\,' & ControlPump \end{bmatrix} . \qquad (1)$$

4.2 Introduce shared variable

Both *ControlAlarm* and *ControlPump* use the CH_4 sensor variable. In order to achieve independent interfaces to the environmental inputs, we want to prevent the pump controller from directly using the methane sensor and instead have it access a shared variable set by the alarm controller to determine if conditions are safe or not.

In fact, a suitable variable already exists, *ringing*. For brevity we will use *ringing* as not only an 'output' variable, but also as the shared variable used for communication between the alarm and pump controllers.

Again some syntactic conveniences can be defined for referring to the state of this variable in the recent past:

$$noisy_\delta == \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge ringing(\!| \, t - \delta \,..\, t \, |\!) = \{\text{true}\}$$
$$quiet_\delta == \lambda\, t : \mathbb{A} \bullet t \geqslant \delta \wedge ringing(\!| \, t - \delta \,..\, t \, |\!) = \{\text{false}\} \,.$$

Since the pump controller will now use *ringing* to determine if methane levels are high, and it takes some time to update *ringing* whenever methane levels change, we need to shorten the available time to update the output variables in order to ensure that *pumping* is still always set correctly within *react* time units of environmental changes. Let *rt* and *pt* be tighter deadlines on how quickly *ringing* and *pumping* must be updated, respectively.

$$\frac{rt, pt : \mathbb{D}}{rt + pt \leqslant react}$$

Development proceeds by using the stronger update times and having the pump controller refer to shared variable *ringing* rather than the methane sensor.

ControlAlarm2

$$\frac{Methane;\ Alarm}{\forall\, t : \mathbb{A} \bullet (danger_{rt}(t) \Rightarrow ringing(t)) \wedge (safe_{rt}(t) \Rightarrow \neg\, ringing(t))}$$

ControlPump2

$$\frac{Water;\ Alarm;\ Pump}{\forall\, t : \mathbb{A} \bullet (deep_{pt}(t) \wedge quiet_{pt}(t) \Rightarrow pumping(t)) \wedge (shallow_{pt}(t) \vee noisy_{pt}(t) \Rightarrow \neg\, pumping(t))}$$

(1) $\sqsubseteq$ "by **R2**"

$$+ringing, pumping: \begin{bmatrix} Water & ControlAlarm2 \\ Methane\,' & ControlPump2 \end{bmatrix} \qquad (2)$$

4.3 Introduce periodic and sporadic requirements

We will 'implement' the pump controller as a periodic behaviour and the alarm controller as a sporadic one. Interarrival times and deadlines for the sporadic *s* and periodic *p* requirements are introduced as follows.

$$\frac{T_s, D_s : \mathbb{D}}{D_s \leqslant T_s \\ T_s = CH_4sep \\ D_s \leqslant rt} \qquad \frac{T_p, D_p : \mathbb{D}}{D_p \leqslant T_p \\ rt + T_p + D_p \leqslant pt}$$

The second constraint on the right ensures that *pumping* is updated in time. Apart from the reliance on *ringing*, which may be rt time units out of date, two successive updates to *pumping* may be separated by period T_p plus deadline D_p in the worst case (i.e., where the first update occurs immediately after arrival in one period and the next occurs just before the deadline in the following period).

We define some convenient notations for referring to the sporadic and periodic arrival times, and the moment *before* the next arrival after some time t:

$$arrive_s == \mathrm{dom}\ CH_4$$
$$arrive_p == \{n : \mathbb{N} \bullet n * T_p\}$$
$$next_s(t) = \begin{cases} \infty, & t \geqslant max\ \mathrm{dom}\ CH_4 \\ max\{u : \mathbb{A} \mid u < (min\ \mathrm{dom}(0\,..\,t \lhd CH_4))\}, & \text{otherwise} \end{cases}$$
$$next_p(t) = max\{u : \mathbb{A} \mid u < \left\lceil \tfrac{t+1}{T_p} \right\rceil T_p\}\ .$$

The sporadic task arrives at each detected change in methane levels. The periodic task arrives every T_p time units. (There is no need to use T_s in defining $arrive_s$ in this example because CH_4 inputs are already known to be separated by at least this amount.)

It is also helpful to be able to assert that some timed variable v does not change its value at any of the absolute times in some, not necessarily contiguous, set A (where to be 'changed' a variable must have a different value than it had in the previous instant):

$$unchanged(v, A) = \forall\, t : A \bullet t \neq 0 \Rightarrow v(t) = v(t-1)\ .$$

We now re-express the controllers as behaviours that, following each arrival a, achieve their functional requirement by the time of their absolute deadline d, and update their output variables only at some time u between a and d.

Sporadic

Methane; *Alarm*

$$\forall\, a : arrive_s \bullet$$
$$\quad \textbf{let}\ d == a + D_s \bullet$$
$$\qquad (danger_{D_s}(d) \Rightarrow ringing(d))$$
$$\qquad \wedge\ (safe_{D_s}(d) \Rightarrow \neg\ ringing(d))$$
$$\qquad \wedge\ \exists\, u : a\,..\,d \bullet unchanged(ringing, (a\,..\,next_s(a)) \setminus \{u\})$$

Periodic

Water; *Alarm*; *Pump*

$$\forall\, a : arrive_p \bullet$$
$$\quad \textbf{let}\ d == a + D_p \bullet$$
$$\qquad (deep_{D_p}(d) \wedge quiet_{D_p}(d) \Rightarrow pumping(d))$$
$$\qquad \wedge\ (shallow_{D_p}(d) \vee noisy_{D_p}(d) \Rightarrow \neg\ pumping(d))$$
$$\qquad \wedge\ \exists\, u : a\,..\,d \bullet unchanged(pumping, (a\,..\,next_p(a)) \setminus \{u\})$$

(2) $\sqsubseteq$ "by **R2**"

$$+ringing, pumping: \left[\begin{matrix} Water & Sporadic \\ Methane, & Periodic \end{matrix}\right] \tag{3}$$

4.4 Introduce worst case execution times

Constants are introduced to represent the programmer's anticipated worst case computation times for the sporadic and periodic requirements.

$$\frac{C_s : \mathbb{D}}{C_s \leqslant D_s} \qquad \frac{C_p : \mathbb{D}}{C_p \leqslant D_p}$$

The actual computation time required upon each arrival is not yet known (and, indeed, will not be known until the final object code is generated!). At most, therefore, we can say only that at each arrival the notional invocation request will not exceed the worst case execution time.

$$\frac{invreq_s : \mathbb{A} \pfun \mathbb{D}}{\begin{array}{l} \mathrm{dom}\, invreq_s = arrive_s \\ max\ \mathrm{ran}\, invreq_s \leqslant C_s \end{array}} \qquad \frac{invreq_p : \mathbb{A} \pfun \mathbb{D}}{\begin{array}{l} \mathrm{dom}\, invreq_p = arrive_p \\ max\ \mathrm{ran}\, invreq_p \leqslant C_p \end{array}}$$

4.5 Introduce feasible schedule

There are three possible 'tasks' $\mathbb{T}$ that may occupy the processor in our system, the sporadic s and periodic p application tasks, and no activity at all:

$$\mathbb{T} ::= s \mid p \mid idle\ .$$

One of these is running at every moment in time:

$$Run \mathrel{\widehat=} [\, running : \mathbb{A} \fun \mathbb{T} \,]\ .$$

Our overall constraint on the processor is that, for each invocation request, it gives the application tasks as many units of run time as they need.

Processor

$$\begin{array}{l} Methane;\ Run \\ \hline \forall\, a : arrive_s \bullet \#(a\,..\,a + D_s \lhd running \rhd \{s\}) = invreq_s(a) \\ \forall\, a : arrive_p \bullet \#(a\,..\,a + D_p \lhd running \rhd \{p\}) = invreq_p(a) \end{array}$$

(3) $\sqsubseteq$ "by **R3**, **R2**"

$$\left|\left[\begin{matrix} ringing, \\ +pumping, : \\ running \end{matrix}\ \left[\begin{matrix} Water & Sporadic \\ Methane, & Periodic \\ & Processor \end{matrix}\right] \setminus running\right]\right| \tag{4}$$

Here the scheduling model we are constructing first begins to benefit us. How can we be sure that the property of *running* specified in *Processor* is feasible? It is not obvious by inspection, especially in more complex examples. If it is infeasible then the above refinement step makes the effect 'false', i.e., we have refined to 'magic', and any further development effort would be wasted.

We require a feasibility test to guarantee that this is not the case. Test 1 from Section 2.2 is suitable. The programmer merely needs to supply anticipated values for the symbolic constants introduced above to see if the development is proceeding along a viable path. For instance, assuming $D_s = 15$, $C_s = 6$, $D_p = 20$ and $C_p = 10$, we can use test 1 to show that the requirement can be met for the sporadic behaviour because

$$\frac{C_s}{D_s} = \frac{6}{15} \leqslant 1 .$$

Similarly, the periodic behaviour can also be satisfied because

$$\frac{C_s}{D_s} + \frac{C_p}{D_p} = \frac{6}{15} + \frac{10}{20} \leqslant 1 .$$

(We have not yet introduced enough detail in the refinement to determine task blocking overheads, so these figures are omitted.)

Of course, this tells us only that a scheduler *exists* that will satisfy the requirement, not that the particular scheduling policy we will ultimately employ can do so.

4.6 Introduce ready indicators

Effective priorities $\mathbb{E}$ for tasks can be represented by natural numbers, with higher values denoting higher priorities,

$$\mathbb{E} == \mathbb{N} .$$

The lowest 'normal' priority is 1; we use 0 to indicate that a task is *not* ready.

In this example there are base priorities π_s and π_p for each of the two tasks, and a ceiling priority π_c for when they access the shared variable.

$$\begin{array}{|l} \pi_p, \pi_s, \pi_c : \mathbb{E} \\ \hline 0 < \pi_p < \pi_s \leqslant \pi_c \end{array}$$

It is considered more critical to note changes in methane than water levels, so π_s is higher than π_p.

Whenever a task is ready to run, its *ready* indicator records this with a value denoting the effective priority at which the task wishes to execute. The periodic task may run at its own base priority π_p, or the higher ceiling priority π_c while it is accessing shared variable *ringing*:

$$Ready_p \mathrel{\widehat{=}} [\, ready_p : \mathbb{A} \rightarrow \mathbb{E} \mid \operatorname{ran} ready_p = \{0, \pi_p, \pi_c\} \,] .$$

Our simple sporadic task does nothing but update the shared variable, so it always runs at the ceiling priority π_c, and π_s is not needed in this case:

$$Ready_s \mathrel{\widehat=} [\, ready_s : \mathbb{A} \rightarrow \mathbb{E} \mid \operatorname{ran} ready_s = \{0, \pi_c\}\,] \ .$$

$$(4) \sqsubseteq \text{“by } \mathbf{R3}, \mathbf{R2}\text{”}$$

$$\left\| \left[\begin{array}{l} ringing, \\ pumping, \\ +running, : \\ ready_p, \\ ready_s \end{array} \right] \left[\begin{array}{l} \\ Water \\ Methane \\ \\ \end{array} , \begin{array}{l} Sporadic \\ Periodic \\ Processor \\ Ready_s \\ Ready_p \end{array} \right] \setminus \left[\begin{array}{l} running, \\ ready_p, \\ ready_s \end{array} \right] \right\| \qquad (5)$$

4.7 Introduce protected objects

In this example we do not need to add any *update* channels, since *Periodic* does not attempt to write to *ringing*. Also, the *Sporadic* specification serves as both a sporadic requirement and custodian of the shared variable so a distinct *Protected* specification is unnecessary.

The task designs are completed by defining how access to the shared variable is controlled through manipulation of effective priorities. In doing so we further contrain the task definitions so that they always complete their work before some worst case response time, no greater than their deadlines.

$$\frac{R_s : \mathbb{D}}{R_s \leqslant D_s} \qquad \frac{R_p : \mathbb{D}}{R_p \leqslant D_p}$$

Our simple sporadic task does nothing but update shared variable *ringing*, so it always runs at effective priority π_c.

Sporadic2

$$\begin{array}{l} Methane;\ Run;\ Alarm;\ Ready_s \\ \hline \forall\, a : arrive_s \bullet \\ \quad \exists\, r : a \,..\, a + R_s \mid r = max\ \mathrm{dom}(a \,..\, a + D_s \lhd running \rhd \{s\}) \bullet \\ \qquad (danger_{r-a}(r) \Rightarrow ringing(r)) \\ \qquad \wedge\ (safe_{r-a}(r) \Rightarrow \neg\ ringing(r)) \\ \qquad \wedge\ (\exists\, u : a \,..\, r \bullet unchanged(ringing, (a \,..\, next_s(a)) \setminus \{u\})) \\ \qquad \wedge\ ready_s(\!|\ a \,..\, r\ |\!) = \{\pi_c\} \\ \qquad \wedge\ ready_s(\!|\ r + 1 \,..\, \mathrm{next}_s(a)\ |\!) = \{0\} \end{array}$$

Time r is the (absolute) time at which a *particular* task invocation completes its work, defined to be the earliest time at which it has received $invreq_s(a)$ units of processor time (see schema *Processor* above). The task is 'ready' from time a until r. After this it will not request any more processor time until the next arrival.

The periodic task, on the other hand, must access the shared variable 'created' by the above schema. In doing so it may block task s if a CH_4 event occurs while the periodic behaviour p is accessing *running*.

$$B_s : \mathbb{D}$$
$$\hline$$
$$B_s \leqslant C_p$$

Activity of the periodic task following each arrival time a can be divided into two parts. We require that it first samples the value of the shared variable *ringing*, before some time x; to do so it must be ready at priority π_c. It then samples the value of H_2O and sets the value of *pumping*, before its worst case response time R_p has elapsed.

Periodic2

$$Water;\ Alarm;\ Run;\ Pump;\ Ready_p$$
$$\hline$$
$$\begin{aligned}
&\forall a : arrive_p \bullet \\
&\quad \exists r : a \,..\, a + R_p \mid r = max\ \mathrm{dom}(a \,..\, a + D_p \lhd running \rhd \{p\}) \bullet \\
&\qquad \exists x : a \,..\, r \mid \#(a \,..\, x \lhd running \rhd \{p\}) \leqslant B_s \bullet \\
&\qquad\quad (deep_{r-x}(r) \land quiet_{x-a}(x) \Rightarrow pumping(r)) \\
&\qquad \land\ (shallow_{r-x}(r) \lor noisy_{x-a}(x) \Rightarrow \neg\ pumping(r)) \\
&\qquad \land\ (\exists u : x \,..\, r \bullet unchanged(pumping, (a \,..\, next_p(a)) \setminus \{u\})) \\
&\qquad \land\ ready_p(\!|\ a \,..\, x\ |\!) = \{\pi_c\} \\
&\qquad \land\ ready_p(\!|\ x + 1 \,..\, r\ |\!) = \{\pi_p\} \\
&\qquad \land\ ready_p(\!|\ r + 1 \,..\, next_p(a)\ |\!) = \{0\}
\end{aligned}$$

The new subscripts and arguments to *deep*, *quiet*, etc., reflect the more precise times at which these properties are tested. The alarm value is sampled within x time units of arrival a. The water sensor is sampled some time in the remaining $r - x$ time units, before the particular response time r.

(5) $\sqsubseteq$ "by **R2**"

$$\left\| \left[\begin{array}{l} ringing, \\ pumping, \\ +running, \\ ready_p, \\ ready_s \end{array} : \left[\begin{array}{l} Water \\ Methane \end{array}, \begin{array}{l} Sporadic2 \\ Periodic2 \\ Processor \end{array} \right] \setminus \begin{array}{l} running, \\ ready_p, \\ ready_s \end{array} \right] \right\| \qquad (6)$$

4.8 Introduce scheduler

The scheduling policy can be introduced easily. At any time t it makes the running task the highest priority one in a notional ready queue formed from the $ready_s$ and $ready_p$ indicators.

Scheduler

$$Ready_p;\ Ready_s;\ Run$$
$$\hline$$
$$\forall t : \mathbb{A} \bullet running(t) = highpri(t)$$

Function *highpri* returns the highest priority ready task at time t:

$$highpri(t) = \begin{cases} idle, & ready_p(t) = ready_s(t) = 0 \\ p, & ready_p(t) > ready_s(t) \\ & \vee\ (ready_p(t) = ready_s(t) \neq 0 \\ & \quad \wedge\ t > 0 \wedge highpri(t-1) = p) \\ s, & ready_s(t) > ready_p(t) \\ & \vee\ (ready_s(t) = ready_p(t) \neq 0 \\ & \quad \wedge\ (t > 0 \Rightarrow highpri(t-1) \in \{s, idle\})) \end{cases} .$$

In other words, the highest priority ready task is *idle* if neither p or s is ready at time t. It is p if that task is ready with a higher effective priority than s, or both tasks are ready with the same priority and the task running at the last moment in time was p. An arbitrary decision has been made to favour s when both tasks become ready at the same priority at time 0 or following an *idle* period.

(6) $\sqsubseteq$ "by **R2**"

$$\left\| \left[\begin{array}{l} ringing, \\ pumping, \\ +running, : \\ ready_p, \\ ready_s \end{array} \left[\begin{array}{l} Water \\ Methane \end{array}, \begin{array}{l} Sporadic2 \\ Periodic2 \\ Processor \\ Scheduler \end{array} \right] \setminus \begin{array}{l} running, \\ ready_p, \\ ready_s \end{array} \right] \right\| \qquad (7)$$

Again our refinement process benefits from scheduling theory. Although we claim the existence of worst case response times R_s and R_p above we now need to show that satisfactory values do indeed exist under this scheduling policy. Test 2 from Section 2.2 is suitable. For instance, given values of $T_s = 100$, $T_p = 25$ and $B_s = 5$, as well as the values for deadlines and computations times used in Section 4.5, we can determine that the sporadic requirement can be satisfied because

$$R_s = C_s + B_s = 6 + 5 = 11 \leqslant D_s .$$

There are no tasks of higher priority than s to pre-empt it, and the only lower-priority task that can block it is p, which can do so at most once. Hence the sporadic requirement will always meet its deadline of 15 time units from arrival.

To test the periodic requirement we note that no lower-priority tasks exist to block p, so 'B_p' is zero. But p can be pre-empted by s, so interference must be considered and

$$R_p = C_p + \left\lceil \frac{R_p}{T_s} \right\rceil C_s .$$

This recursive equation converges as follows:

$$R_p^0 = 0$$

$$R_p^1 = C_p + \left\lceil \frac{R_p^0}{T_s} \right\rceil C_s = 10 + \left\lceil \frac{0}{100} \right\rceil 6 = 10$$

$$R_p^2 = C_p + \left\lceil \frac{R_p^1}{T_s} \right\rceil C_s = 10 + \left\lceil \frac{10}{100} \right\rceil 6 = 16$$

$$R_p^3 = C_p + \left\lceil \frac{R_p^2}{T_s} \right\rceil C_s = 10 + \left\lceil \frac{16}{100} \right\rceil 6 = 16 \,.$$

Hence R_p is less than the deadline of 20 and we can conclude that the system is indeed schedulable! Intuitively this value is reasonable because, given the longer interarrival time of the sporadic task compared to the periodic one, s can preempt p at most once at any arrival of p.

4.9 Separate tasks, shared objects and scheduler

The task requirements and the scheduler can now be separated, for later individual refinement, by straightforward application of the refinement rules. Firstly the scheduler is separated from the tasks.

$$(7) \sqsubseteq \text{“by } \mathbf{R4}, \mathbf{R2}, \mathbf{R1}, \mathbf{R2}, \mathbf{R1}\text{”}$$

$$[\![\quad + running\colon \left[\begin{array}{l}(\exists\, running \bullet \\ \quad Sporadic2 \\ \quad \wedge\ Periodic2)\end{array} , \ Scheduler\right] \qquad (8)$$

$$\|$$

$$+ \begin{array}{l} pumping, \\ ringing, \\ ready_p, \\ ready_s \end{array} : \left[\begin{array}{l} Water \\ Methane \\ Processor \end{array} , \begin{array}{l} Sporadic2 \\ Periodic2 \end{array}\right] \qquad (9)$$

$$\backslash running, ready_p, ready_s \ ]\!]$$

Part of the first ‘strengthen effect’ step eliminated *Processor* from the effect of the scheduler component by making use of knowledge about the tasks, specifically that they ask for exactly *invreq* time units at each invocation. (The proof relies on our scheduling policy model being deterministic; for particular task behaviours a unique *running* value is defined.) The first ‘weaken assumption’ step then removed unnecessary properties *Water*, *Methane*, *Scheduler* and *Processor* from the assumption of the scheduler component. The second ‘weaken assumption’ removed *Scheduler*, *Sporadic2*, *Periodic2* and the unused existentially-quantified variables from the assumption of the tasks component.

Then the second component, the individual task requirements, is further refined to give three parallel components in total.

$$(9) \sqsubseteq \text{“by } \mathbf{R4}, \mathbf{R2}, \mathbf{R1}, \mathbf{R2}, \mathbf{R1}\text{”}$$

$$+ ringing, ready_s\colon \left[\begin{array}{l} Methane \\ Processor \end{array} , \ Sporadic2\right] \qquad (10)$$

$$\|$$

$$+ pumping, ready_p\colon \left[\begin{array}{l} Water \\ Processor \\ Sporadic2 \end{array} , \ Periodic2\right] \qquad (11)$$

In both components the effect is strengthened to eliminate the 'non-constructed' variables. The assumption for the sporadic requirement was weakened to remove *Water*, *Sporadic2* and *Periodic2*, and that of the periodic task to remove *Methane* and *Periodic2*. *Sporadic2* remains as an assumption for the periodic task due to the role of *ringing* as a shared variable in this example. Both tasks retain assumption *Processor* so that they know they will receive as much processor time as needed.

4.10 Implementation

Our ultimate aim is to apply the above method to development of Ada 95 programs. This is feasible because the Ada 95 language design accounts for recent advances in schedulability theory [8]. The system above adheres to constructs supported by Ada 95 and can be mapped to the following program.

```
with Ada.Real_Time; use Ada.Real_Time;
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);

protected Alarm is -- implements Sporadic2
  pragma Locking_Policy(Ceiling_Locking);
  pragma Priority(Interrupt_Priority'First); -- i.e., π_c
  function danger return Boolean;
private
  procedure CH4high; pragma Attach_Handler(CH4high,...);
  procedure CH4low;  pragma Attach_Handler(CH4low,...);
  -- We assume blocked interrupts remain pending!
  ringing: Boolean := False; for ringing use ...;
end Alarm;

protected body Alarm is
  procedure CH4high is
  begin
    ringing := True
  end CH4high;
  procedure CH4low is
  begin
    ringing := False
  end CH4low;
  function danger return Boolean is -- part of Periodic2
  begin
    return ringing;
  end danger;
end Alarm;

task ControlPump is -- implements Periodic2
  pragma Priority(Priority'First); -- i.e., π_p
```

```
end ControlPump;

task body ControlPump is
  H2Omark: constant Integer := H2Omark;
  H2O: Integer; for H2O use ...;
  pumping: Boolean := False; for pumping use ...;
  period: Time_Span := Tp;
  Next: Time := Clock; -- time "0"
begin
  loop
    delay until Next; -- arrival time
    pumping := not Alarm.danger and then H2O >= H2Omark;
    Next := Next + period;
  end loop;
end ControlPump;
```

Our *Scheduler* specification is implemented trivially by the compiler directive on the second line which requests static-priority, pre-emptive scheduling. Similarly, another compiler directive within protected object `Alarm` requests the ceiling locking protocol introduced in Section 4.7 above.

Our degenerate sporadic 'task', *Sporadic*2, is implemented by two interrupt-handling procedures `CH4high` and `CH4low`. (Ada 95 interrupt-handlers are parameterless, unlike the CH_4 variable which carried a boolean value.) The *Periodic*2 requirement is implemented as an iterative Ada task, with a function `danger` that gives it access to the shared variable `ringing`.

Input and output variables `H2O`, `ringing` and `pumping` are mapped to hardware-specific memory locations which are assumed to be continuously accessable by the environment. The hardware-specific interrupt handlers attached to the `CH4high` and `CH4low` procedures implement the CH_4 input.

Although not shown above, this program is still considered to be accompanied by the calculated timing constraints on each component. These must be retained until formally discharged. Real-time development is not considered complete until it has been shown, either experimentally or through further proof, that a call to function `danger` takes no more than B_s time units, that procedures `CH4high` and `CH4low` execute in under C_s time units, and that each iteration of `ControlPump` takes less than C_p time units (including context switching and interrupt handling overheads!).

We have not discussed in this paper how such computation times are determined, or how sequential code segments are generated. However, refinement rules that achieve both aims have already appeared [6], and extend the methodology above to do this.

5 Conclusion

We have shown how new results in real-time scheduling and refinement theories can be integrated. This was done by representing the computational model

used by pre-emptive scheduling theory in a real-time refinement calculus. The refinement calculus then gained from proven schedulability results.

This work is part of the *Quartz* project, investigating formal methods for the development of hard real-time software. A number of major projects have goals similar to Quartz, especially the safemos [3], *ProCoS* [9] and TAM [13] projects, and Hooman's development method [7], but none makes use of scheduling theory. Previous modelling exercises used Z to define aspects of priority scheduling [5] and the priority ceiling protocol [12], but did not model absolute timing or define refinement methods.

Acknowledgements We wish to thank Andy Wellings for advice on scheduling theory, Graeme Smith for correcting errors in the paper, and the anonymous FME'96 referees for their comments. The Quartz project is funded by the Information Technology Division of the Australian Defence Science and Technology Organisation. Ian Hayes' participation in this work was funded by a University of Queensland Project Enabling Grant.

References

1. N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
2. T.P. Baker. Stack-based scheduling of real-time processes. *Real Time Systems*, 3(1):67–99, March 1991.
3. J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
4. A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison-Wesley, 1990.
5. A. Burns and A.J. Wellings. Priority inheritance and message passing communication: A formal treatment. *The Journal of Real-Time Systems*, 3:19–44, 1991.
6. C. Fidge. Adding real time to formal program development. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 618–638. Springer-Verlag, 1994.
7. J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.
8. Ada 9X Mapping/Revision Team Intermetrics. Ada 9X reference manual, draft version 5.0, June 1994.
9. H. Jifeng. *Provably Correct Systems*. McGraw-Hill, 1995.
10. B. Mahony. Using the refinement calculus for dataflow processes. Technical Report TR 94-32, Software Verification Research Centre, October 1994.
11. B. Mahony. Networks of predicate transformers. Technical Report TR 95-5, Software Verification Research Centre, February 1995.
12. M. Pilling, A. Burns, and K. Raymond. Formal specifications and proofs of inheritance protocols for real-time scheduling. *Software Engineering Journal*, 5(5), September 1990.
13. D. Scholefield. Proving properties of real-time semaphores. *Science of Computer Programming*, 24(2):159–181, April 1995.