SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-11

Coercing Real-time Refinement: A Transmitter

Ian J. Hayes   Mark Utting

June 1996
Revised January 1997

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.uq.edu.au

This paper is published in the *BCS-FACS Northern Formal Methods Workshop, Electronic Workshops in Computing*, Springer-Verlag, 1997

Coercing Real-time Refinement: A Transmitter

Ian J. Hayes[*] Mark Utting[†]

Abstract

Our overall goal is to support the development of real-time programs from specifications via a process of stepwise refinement. One problem in developing a real-time program in a high-level programming language is that it is not possible to determine the detailed timing characteristics of the program until the compiler and target machine are taken into account. To overcome this problem the programming language can be augmented with directives specifying real-time constraints; it is a compile-time error if the compiler cannot guarantee that the generated code will meet them. During the refinement process the timing directives are inserted into the program in order to ensure it meets the specification. The paper introduces the real-time directives, gives a set of laws for real-time refinement, and illustrates their use in the refinement of a simple real-time transmitter.

Keywords Real-time refinement.

1 Introduction

Programming languages such as Ada provide some real-time features such as delay commands. However, there is no facility to specify that a section of code is to be executed by a desired time deadline. To overcome this problem we propose to add a *deadline* directive to the programming language so that time deadlines can be incorporated into programs. The directive is of the form

deadline *exp*

Consider the following fragment of a program, in which dt is a normal program variable of type *Time* and τ stands for the finishing time of the command in which it appears.

$$\left\{\tau \le dt\right\} \tag{1}$$

[*]Department of Computer Science, The University of Queensland, Brisbane, 4072, Australia.

[†]Software Verification Research Centre, The University of Queensland, Brisbane, 4072, Australia.

$$\textbf{delay until}\ dt \tag{2}$$
$$\text{“commands which do not update } dt\text{”} \tag{3}$$
$$\textbf{deadline}\ dt + 1 millisecond \tag{4}$$

We assume the fragment begins execution at a time before dt, as indicated by the assertion (1). The delay command (2) guarantees that execution of the commands following it (3) will begin after time dt. The *lateness* of a delay command is the difference between its actual termination time and the specified delay time. The novel addition to the programming language is the deadline directive (4). It adds the constraint that the time at which it completes is before dt plus 1 millisecond, that is, the lateness of the delay (2) plus the execution time of the intermediate commands (3), takes no more than 1 millisecond.

The deadline directive is rather tricky to implement. If the time taken for the preceding commands (2) and (3) is more than 1 millisecond, then there is no way the deadline directive (4) can make time go backwards. However, if the time taken to execute (2) and (3) is always guaranteed to be less than 1 millisecond, then (4) can simply be implemented by generating no code at all for it.

The deadline command can be viewed as a directive to the compiler for the programming language. If the compiler can determine by analysing the code generated for (2) and (3) that they will always take less than 1 millisecond to execute on the target machine, then the compiler will successfully generate code for them. Because in this case the deadline is guaranteed to be met, it can be discarded. On the other hand, if the compiler cannot guarantee that (2) and (3) will always take less than 1 millisecond, then it cannot guarantee to satisfy (4). In that case the compiler must reject the program because it is unable to guarantee its timing correctness on the given target machine.

The combination of a delay command and a deadline directive allows one to add quite detailed timing constraints to a program, and thus provides an effective real-time programming language. Given this programming language, our overall goal is to develop programs from specifications that include timing requirements, using a process of stepwise refinement.

Our approach to real-time specification and refinement is based on the timed refinement calculus [3, 4, 5, 6], and in particular on the adaption of timed refinement for developing sequential programs by Utting and Fidge [10], although our treatment of time constraints is quite different to that in [10].

We present our approach via an example of the refinement of a simple real-time transmitter. In Section 2 we present a specification of a simple transmitter that includes real-time constraints on the transmitter output. In Section 3 we jump straight to the final program to give the reader a feel for the way in which deadline directives are used to express timing constraints. In Section 3 we also outline the analysis process to calculate the timing constraints for the transmitter program. Section 4 presents the refinement laws for our timed refinement calculus. This includes the definition of the deadline directive in terms of a coercion. The refinement rules for the non-real-time constructs are similar to those of Morgan [7], but with extensions for real-time aspects. Section 5 presents the details of the refinement of the transmitter example.

2 The Transmitter Specification

In the timed refinement calculus variables are represented as functions from time to their value at that time. For example, a character variable *ch* is represented by

$$ch : Time \nrightarrow Char. \tag{5}$$

Given a time, t, $ch(t)$ is the value of the variable *ch* at time t. As all variables are represented by such functions of time, we elide the "$Time \nrightarrow$" in their declaration. For example, we declare *ch* via the more conventional

> **var** $ch : Char$,

but emphasise that this is a shorthand for (5).

The environment Before giving the specification of the transmitter we need to set up its environment. The message to be output is a sequence of characters:

> **var** $msg : \mathrm{seq}_0\, Char$.

The sequence *msg* is an input to the transmitter. The number of characters in the message is denoted by $\#msg$, and the indices of *msg* range from 0 to $\#msg - 1$. The message is to be output one character at a time via a single character output buffer:

> **var** $out : Char$,

which is connected to an external output transmission bus. We need to specify the value of *out* over time. The start time for transmission of the message is given by the input variable:

> **var** $st : Time$,

which is not modified by the transmitter. Each character in the message should remain in the output buffer for a contiguous period of at least *chdef* seconds, and the separation between characters is *chsep* seconds:

> **const** $chsep = 100\,\mu\, s;\; chdef = 80\,\mu\, s$.

The time interval during which the ith character should be stable in the output buffer is from $st + chsep * i$ through until $st + chsep * i + chdef$. We introduce the function *interval* to aid in writing the specification. It takes a parameter, i, and returns the time interval (set of times) during which the ith character should be stable in the output buffer:

$$\textbf{let}\ \mathit{interval} == (\lambda\, i : \mathbb{N} \bullet st + \mathit{chsep} * i \,..\, st + \mathit{chsep} * i + \mathit{chdef}).$$

A second auxilliary function, *chout*, takes the same parameter as *interval*, but it returns the set of characters appearing in the output buffer during the ith interval (the image through *out* of the ith interval). If the transmitter is working correctly, this will be a singleton set:

$$\textbf{let}\ \mathit{chout} == (\lambda\, i : \mathbb{N} \bullet \mathit{out}(\!|\ \mathit{interval}(i)\ |\!)).$$

The specification The task of the transmitter is to output the message via the output buffer. The transmission of the first character must begin by time *st*, and after that the characters should appear at intervals of *chsep* seconds, and be stable for at least *chdef* seconds. The transmitter should send the complete message within $\mathit{chsep} * \#\mathit{msg}$ seconds of the starting time. The time at which the transmitter code begins execution is c seconds before the time, *st*, at which the transmission should begin.

$$\mathit{out} : \left[\tau \leq st - c,\ \begin{array}{l}(\forall\, i : 0 \,..\, \#\mathit{msg} - 1 \bullet \mathit{chout}(i) = \{\mathit{msg}(i)\}) \wedge \\ \tau \leq st + \mathit{chsep} * \#\mathit{msg}\end{array}\right]$$

The above specification is given as a specification command [3, 8, 7] of the form $v : [A, E]$. It consists of a list, v, of variables that may be modified by the operation (the frame of the operation); assumptions, A, that the operation may make about the environment of its call; and an effect, E, that the operation is to achieve. Within the assumptions, τ refers to the start time of the command, and within the effect τ_0 and τ refer to the start and finish times, respectively, of the command. We always have $\tau_0 \leq \tau$, so this is not explicitly stated. The assumptions may not reference τ_0. Neither τ_0 nor τ may appear in the frame of a specification or an assignment.

The assumptions may include properties of the values of the variables at the start of the operation (usually referred to as the precondition), but it may also include properties of the values of the environment variables over time. The effect can include properties about the values of the variables in the frame on termination of the operation (usually referred to as the postcondition), but it can also include properties about the values over time. Within a specification command we may refer to the value of a variable v at time t by $v(t)$. If v occurs unindexed by a time, then it stands for $v(\tau)$. Hence, within an assumption, v stands for the value of v at the start time of the command, and within an effect, v stands for the value of v on completion of the command. Within an effect, v_0 stands for $v(\tau_0)$.

The specification above states that the program fragment begins execution before $st - c$ and must terminate before $st + \mathit{chsep} * \#\mathit{msg}$. During this time interval the output buffer is to be updated regularly so that it contains the ith character of the message during $\mathit{interval}(i)$.

3 The final program

Before going through the details of the development of the program, we skip ahead to the final program (Figure 1) in order to see how the real-time directives are used. The program achieves its goal by ensuring that the transmission of the nth character begins before the start of the interval on which it must be stable: $st + chsep * n$. It then waits to ensure the output buffer is stable until at least the end of the nth interval: $st + chsep * n + chdef$.

```
A :: $\{\tau \leq st - c\}$
    |[ var $n : \mathbb{N}$ •
        $n := 0$;
        do $n \neq \#msg \rightarrow$
            $out := msg(n)$;
         B :: deadline $st + chsep * n$;
         C :: delay until $st + chsep * n + chdef$;
            $n := n + 1$
        od
    ]|;
D :: deadline $st + chsep * \#msg$
```

Figure 1: Program for the transmitter

In order to be able to safely discard the deadline directives the compiler needs to analyse the code it generates to check that it will satisfy the time constraints. The analysis can be broken down by identifying paths between significant timing points in the program. We demonstrate this analysis for the transmitter program by decomposing the program into its different timing paths and calculating a time constraint on each of these paths. The analysis that the machine code generated by the compiler meets the timing constraints is beyond the scope of the current paper; the interested reader is referred to [2]. In practice, such checks may be performed by a separate analysis phase that has access to the generated code and details of the timing constraints.

Consider the execution path from A to B in Figure 1. This path includes the allocation of the local variable n and its initialisation, evaluating the loop guard (to *true*), and sending the first character. The start time for this path is assumed to be before $st - c$ and the finish deadline is $st + chsep * n$. At the finish time, for this path the value of n is 0, so the maximum execution time allowed for the path A–B is

$$st + chsep * 0 - (st - c) = c.$$

If the message is empty then the loop guard is false on its first evaluation. This gives a path A–D, which excludes the body of the loop. It contains the allocation and initialisation of n, evaluating the guard (to *false*), exiting (or more precisely, not entering) the loop, and deallocating the local variable n. The start time for this path is before $st - c$ and the finish deadline is $st + chsep * \#msg$. Because the loop guard fails on its first evaluation we know that $\#msg = n = 0$, and hence the maximum execution time allowed for the path A–D is

$$st + chsep * 0 - (st - c) = c.$$

The deadline directive (B) guarantees the time at which the delay command (C) commences is before the delay time specified in C. Hence the delay time specified in C becomes the effective time for the start of paths beginning at C.

There is a path from C cycling back to B. The path includes any lateness of the delay command (C), the increment of n, branching back[1] to the start of the loop, evaluating the guard (to *true*) and sending the next character. The start time of the path is $st + chsep * n + chdef$ and its finish deadline is $st + chsep * n$, but in the meantime n has been incremented, so that the maximum execution time allowed is

$$st + chsep * (n + 1) - (st + chsep * n + chdef) = chsep - chdef.$$

After the last character of the message has been sent, there is a path from C to D, which includes any lateness of the delay command (C), the increment of n, branching back to the start of the loop, evaluating the guard (to *false*), exiting the loop, and deallocating the local variable n. The start time of the path is $st + chsep * n + chdef$ and the finish deadline is $st + chsep * \#msg$. Because the loop guard is *false*, we know that $n = \#msg$ on termination of the loop, but n is incremented before the loop is exited. Hence the maximum execution time allowed for the path is

$$st + chsep * \#msg - (st + chsep * (\#msg - 1) + chdef) = chsep - chdef.$$

4 Refinement laws

The timing related commands in our language are introduced via equivalent specification commands. The standard guarded commands have rules that are adapted to allow for time; if the commands do not refer to time these rules are just the standard ones.

The specification command has already been introduced in Section 2. The rules for weakening an assumption and strengthening an effect carry over to the real-time refinement calculus.

[1]The term 'branching back' refers to the programmer's view of the high-level program; in the generated code there may not be a branch instruction at this point.

Law 1 (weaken assumption) *Provided $P \Rrightarrow P'$,*

$$x : [P, R] \sqsubseteq x : [P', R].$$

Law 2 (strengthen effect) *Provided $P_0 \wedge R' \Rrightarrow R$,*

$$x : [P, R] \sqsubseteq x : [P, R'].$$

where P_0 stands for the predicate P with all occurrences of τ replaced by τ_0, and all unindexed occurrences of x replaced by x_0.

The proviso could also be written $P_0 \Rrightarrow (R' \Rightarrow R)$. Often this law is used to replace an effect R by another R' that is equivalent under the assumptions P_0.

A delay command guarantees that its completion is after the specified time.

Definition 3 (delay) *Provided neither τ_0 nor τ occur free in exp,*

$$[\mathit{exp} \le \tau] = \mathbf{delay\ until}\, \mathit{exp}.$$

The deadline directive allows a time deadline to be specified. It is the compiler's responsibility to ensure the deadline is met by the generated code. If it cannot it should report a compile-time error. In refinement calculus terms, a deadline directive is a coercion.

Definition 4 (deadline) *Provided neither τ_0 nor τ occur free in exp,*

$$[\tau_0 = \tau \le \mathit{exp}] = \mathbf{deadline}\, \mathit{exp}.$$

A requirement that a variable in the frame remains stable for the duration of a command can be achieved by removing the variable from the frame.

Law 5 (contract frame)

$$x, v : [P, R \wedge \mathit{stable}(x, \tau_0 \mathinner{..} \tau)] \sqsupseteq v : [P, R]$$

where $\mathit{stable}(x, S) = (\forall\, t, u : S \bullet x(t) = x(u))$.

For sequential composition we introduce an intermediate predicate Q which holds on termination of the first component.

Law 6 (sequential composition) *Provided u is a fresh name, and neither Q nor R makes use of zero subscripted variables other than τ_0,*

$$w : [P, R] \sqsubseteq [\![\ \mathbf{con}\, u : \mathit{Time} \bullet w : [P \wedge u = \tau, Q];\ w : [Q\,[\tau_0 \backslash u]\,, R\,[\tau_0 \backslash u]]\ ]\!].$$

We assume that the start time of the second component is identical to the finish time of the first component. Hence τ in the effect of the first component is identical to τ in the assumptions of the second component. However, τ_0 in the effect of the first component refers to the start time of the first component. To cope with this we introduce a constant u to stand for the start time of the first component, and replace all occurrences of τ_0 by u within the assumptions of the second component. In the desired overall effect, R, τ_0 refers to the commencement of the whole sequential composition (not the commencement of the second component). Hence the references to τ_0 in the effect of the second component are replaced by u. (Any reference to a zero subscripted variable, x_0, can be replaced by $x(\tau_0)$ before applying this rule.)

During the refinement process it is useful to be able to separate out time constraints from other requirements.

Law 7 (separate timing constraints) *If R is a predicate that does not explicitly involve τ, and V does not involve τ_0 nor zero subscripted variables then,*

$$w : [P \wedge U, R \wedge V] \sqsubseteq \{U\};\ w : [P, R];\ [V].$$

Note that assertions, such as $\{U\}$, take no time. There may be references to v (and v_0) in P and R; these are really references to $v(\tau)$ (and $v(\tau_0)$). These implicit references to τ are allowed.

Law 8 (introduce variable) *Provided neither x, τ_0 nor τ occur free in P or R,*

$$w : [P, R] \sqsubseteq |[\ \mathbf{var}\, x \bullet x, w : [P, R]\ ]|.$$

Because allocation and deallocation of local variables may take time, one often precedes a step which introduces a local variable with the law for separating out time constraints.

Law 9 (assignment) *Provided $P \Rrightarrow R\,[x \backslash exp]$ and neither τ_0 nor τ occur free in exp,*

$$x : [P, R] \sqsubseteq x := exp.$$

Law 10 (iteration) *For the introduction of a loop one needs to supply a variant expression V (not involving zero subscripted names) which evaluates to an element of a well-founded set with ordering $\prec$. Provided neither τ_0 nor τ occur free in INV or G,*

$$w : [INV, INV \wedge \neg\, G] \sqsubseteq \mathbf{do}\{INV\}\, G \rightarrow w : [G \wedge INV, INV \wedge V \prec V_0]\ \mathbf{od}\,.$$

5 The Transmitter Refinement

We assume that the time used in the specification refers to that provided by the clock of the implementation. If this were not the case, we would need to assume that there was some bound on the clock drift error and factor this into the refinement.

We also assume we are dealing with a real-time program that has sole control of the processor for the duration of its execution. There is no allowance for interrupts, task swapping, etc. See [1] for discussion of these issues.

Our refinement begins from the specification given in Section 2. We repeat the specification here, but remind the reader that the environment of the specification, as detailed in Section 2 is necessary to be able to interpret the specification.

$$out : \left[\tau \leq st - c, \begin{array}{l} (\forall\, i : 0 \,..\, \#msg - 1 \bullet chout(i) = \{msg(i)\}) \wedge \\ \tau \leq st + chsep * \#msg \end{array} \right] \tag{6}$$

Separate out the finishing time constraint The general approach that we take is to factor out explicit time constraints when they arise in the development. Otherwise the refinement is similar to a standard non-real-time refinement.

The first step in our refinement is to separate out the finishing time requirement. This is necessary because a local variable n is required to step through the characters, and the deallocation of the variable may take time.

(6)
$\sqsubseteq$ separate out the time constraint (Law 7)
$\left\{\tau \leq st - c\right\}$
$out : [true, (\forall\, i : 0 \,..\, \#msg - 1 \bullet chout(i) = \{msg(i)\})]\,;$ (7)
$[\tau \leq st + chsep * \#msg]$ (8)

The final time limit corresponds to a deadline directive.

(8)
$\sqsubseteq$ Definition 4
deadline $st + chsep * \#msg$

We introduce a variable, n, to step through the characters of the message. The effect is strengthened so that on termination, n is the size of the input message.

(7)
$\sqsubseteq$ Law 8; Law 2

var $n : \mathbb{N} \bullet$

$$n, out : \left[true, \begin{array}{l} (\forall i : 0 \mathrel{..} \#msg - 1 \bullet chout(i) = \{msg(i)\}) \wedge \\ n = \#msg \end{array} \right] \tag{9}$$

Set up for loop The loop invariant states that the characters up to $n-1$ have been successfully sent. Our goal is achieved when $n = \#msg$. We split into an initialisation to establish the invariant, and a loop which maintains it.

(9)
$\sqsubseteq$ Law 6
let $INV == 0 \leq n \leq \#msg \wedge (\forall i : 0 \mathrel{..} n-1 \bullet chout(i) = \{msg(i)\}) \bullet$
$n, out : [true, INV];$ (10)
$n, out : [INV, INV \wedge n = \#msg]$ (11)

As there are no occurrences of τ_0 or τ, the local constant u used in Law 6 is redundant.

Initialisation The invariant is established by setting n to 0.

(10)
$\sqsubseteq$ Law 9
$n := 0$

Introduce loop Note that the invariant does not make explicit reference to τ, as required by Law 10. It does, however, make reference to times relative to st. The variant for the loop is $\#msg - n$, which from the loop invariant is guaranteed to be a natural number.

(11)
$\sqsubseteq$ Law 10
do $n \neq \#msg \rightarrow$
    $n, out : [n \neq \#msg \wedge INV, INV \wedge \#msg - n < \#msg - n_0]$ (12)
od

Loop body The body of the loop outputs a single character and increments n. Incrementing n guarantees the variant is decreased.

(12)
$\sqsubseteq$ Law 6; Law 9
$out : [n \neq \#msg \wedge INV, INV\ [n \backslash n+1]];$ (13)
$n := n + 1$

Output nth character To maintain the invariant, the nth character is output by assigning it to the output buffer and ensuring that it is stable for the period from $st + chsep * n$ until $st + chsep * n + chdef$.

(13)

$\sqsubseteq$ expanding INV

$$out : \left[\begin{array}{l} n \neq \#msg \wedge 0 \leq n \leq \#msg \\ (\forall\, i : 0 \mathinner{..} n-1 \bullet chout(i) = \{msg(i)\}) \end{array}, \begin{array}{l} 0 \leq n+1 \leq \#msg \wedge \\ (\forall\, i : 0 \mathinner{..} n \bullet chout(i) = \{msg(i)\}) \end{array} \right]$$

$\sqsubseteq$ Law 2; Law 1

$out : [0 \leq n < \#msg, chout(n) = \{msg(n)\}]$

$\sqsubseteq$ Law 6; Law 9; Definition of $chout$

$out := msg(n);$

$out : [out = msg(n), out(\!|\ interval(n)\ |\!) = \{msg(n)\}]$ (14)

The nth character must remain stable in the output buffer from $st + chsep * n$ through until $st + chsep * n + chdef$. We achieve this goal by ensuring that it remains stable for an interval encompassing this range.

(14)

$\sqsubseteq$ Law 2

$out : \left[out = msg(n),\ \ out(\!|\ \tau_0 \mathinner{..} \tau\ |\!) = \{msg(n)\} \wedge interval(n) \subseteq (\tau_0 \mathinner{..} \tau)\ \right]$

$\sqsubseteq$ Law 2

$out : \left[out = msg(n),\ \ stable(out, \tau_0 \mathinner{..} \tau) \wedge interval(n) \subseteq (\tau_0 \mathinner{..} \tau)\ \right]$

$\sqsubseteq$ Law 5; Law 1

$[true, interval(n) \subseteq (\tau_0 \mathinner{..} \tau)]$

$\sqsubseteq$ Definition of $interval$

$[true, \tau_0 \leq st + chsep * n \wedge st + chsep * n + chdef \leq \tau]$

$\sqsubseteq$ Law 7

$[true, \tau_0 \leq st + chsep * n];$ (15)

$[st + chsep * n + chdef \leq \tau]$ (16)

A deadline directive is used to guarantee the transmission begins on time.

(15)

$\sqsubseteq$ Definition 4

deadline $st + chsep * n$

The character must remain stable until $st + chsep * n + chdef$. This is achieved by delaying until the time has elapsed.

$$\begin{array}{l} \quad (16) \\ \sqsubseteq \text{Definition 3} \\ \quad \textbf{delay until}\ st + chsep * n + chdef \end{array}$$

The collected final program is shown in Figure 1 in Section 3.

The approach employed to refine the transmitter program was to separate out timing constraints using Law 7 whenever the timing constraints become explicit. Otherwise the refinement process is like that for a non-real-time development. The very first step in refining the specification (6) was to separate out the overall time constraint on the program. The steps after that were just like normal refinement steps down to the refinement of (14), which separated out the timing constraints on the output of each character.

6 Conclusions

To be able to express real-time constraints in a high-level programming language we have added a deadline directive to existing real-time features such as delays. While the deadline directive is a simple syntactic extension to a programming language, supporting it fully within a compiler is non-trivial:

- the high-level programs need to be analysed to determine timing constraints on execution paths, as was done for the transmitter example in Section 3; and
- the machine code generated for each of these execution paths needs to be analysed to determine whether it meets the constraint [2].

However, timing constraints are essential for genuine real-time program development, and it is just these sorts of analyses that are required to ensure that real-time constraints are met in any language. The use of the deadline directive allows the timing analysis to be deferred until after compilation, when more information is available (register allocations, memory addresses, etc.), which enables tighter timing bounds to be obtained. This can be compared with the approach of Shaw [9], which performs the timing analysis at the high-level language level and hence is only capable of looser timing bounds.

The inclusion of a deadline directive in a programming language allows real-time constraints to be expressed in a high-level language, but our overall goal is the development of real-time programs from specifications with time constraints via a process of stepwise refinement. The approach we have taken is based on the timed refinement calculus [3] as adapted for developing sequential real-time programs [10]. The main advance in this paper over [10] is that the deadline directive allows simpler expression of timing deadlines, and most importantly, it allows the separation out of timing constraints during the development process. That allows many of the development steps to appear just like standard non-real-time refinement steps. For example, in the transmitter example only the refinement steps from (6), (8), (14), (15) and (16) involved real-time constraints. The remaining steps were essentially standard sequential refinements.

For the purposes of this paper we have limited our language and laws to those required to present the essential ideas and to express the transmitter example. The timed refinement calculus is also capable of supporting other real-time features such as reading from a real-time clock, and reading from an external input variable, whose value changes over time independently of the control of the program.

Acknowledgements We would like to thank Colin Fidge, Steve Grundon and Peter Kearney for feedback on earlier drafts of this paper. Mark Utting's contribution was funded by the Information Technology Division of DSTO.

References

[1] C. J. Fidge, M. Utting, P. Kearney, and I. J. 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*, volume 1051 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1996.

[2] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Trans. on Software Eng.*, 21(7):593–604, 1995.

[3] B. P. Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, Department of Computer Science, University of Queensland, 1992.

[4] B. P. Mahony and I. J. Hayes. A case study in timed refinement: A central heater. In *Proc. BCS/FACS Fourth Refinement Workshop*, Workshops in Computing, pages 138–149. Springer, January 1991.

[5] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Trans. on Software Engineering*, 18(9):817–826, 1992.

[6] B. P. Mahony, C. Millerchip, and I. J. Hayes. A boiler control system: Overview of a case study in timed refinement. In Diana Del Bel Belluz and Herbert C. Ratz, editors, *Software Safety: Everybody's Business, Proceedings of the 1993 International Invitational Workshop on Design and Review of Software-Controlled Safety-Related Systems, Ottawa*, pages 189–208. The Institute of Risk Research, 1994.

[7] C. C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[8] C.C. Morgan. The specification statement. *ACM Trans. Prog. Lang. and Sys.*, 10(3), July 1988.

[9] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[10] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, editor, *Proc. 7th BCS/FACS Refinement Workshop*, Electronic Workshops in Computing. Springer, July 1996.