# Deadlines are termination

*I. J. Hayes*
*Department of Computer Science and Electrical Engineering, and Software Verification Research Centre, The University of Queensland Brisbane, 4072, Australia (ianh@csee.uq.edu.au).*

*M. Utting*
*Department of Computer Science, School of Computing and Mathematical Sciences, The University of Waikato*
*Private Bag 3105, Hamilton, New Zealand (marku@cs.waikato.ac.nz).*

**Abstract**
We have recently extended the sequential refinement calculus to handle real-time programs. A novel deadline command allows execution time limits to be expressed in a high-level language. The calculus allows refinement steps that separate timing constraints from non-timing requirements. Rules are provided for handling timing constraints, but the refinement of components implementing non-timing requirements is essentially the same as in the standard refinement calculus.

In this paper, we present a new refinement rule for loops that does not require a variant for termination, but uses a deadline command instead. To illustrate the calculus and the new loop introduction rule, we present an example refinement of a program that calculates the size of a kiwifruit from the time it takes to pass through a light beam.

**Keywords**
Real-time, refinement, deadlines, termination

## 1 INTRODUCTION

Formal correctness techniques for real-time programs are less well-developed than those for non-real-time programs, yet the need for them is certainly no less, given that many safety-critical embedded systems involve real-time requirements.

Our goal is to provide a method for the stepwise refinement of sequential, real-time programs from real-time specifications. We follow the refinement calculus approach (Back 1980, Morgan 1994) of devising a wide-spectrum language that encompasses both real-time programs and real-time specifications, and the spectrum in between. To meet this goal, we have found it desirable to:

- use a specification notation which represents variables as traces (functions over time (real numbers)), so that timing requirements can be expressed, including properties of variables whose values change over time.

- distinguish between external inputs, that are not under the direct control of the program, and external outputs and local variables that are.
- add a **deadline** command to our high-level programming language to allow timing requirements to be recorded during development.
- delay the discharging of timing requirements (e.g., **deadline** commands) until after compilation, when the detailed properties of the target machine can be taken into account.

In a previous paper (Hayes & Utting 1997*a*), we developed a calculus for refining an individual real-time process to sequential code. That work was based on the foundations developed by Utting & Fidge (1996), but extended them by introducing a deadline command, which greatly simplifies the treatment of timing constraints. Section 2 gives an overview of this calculus, its real-time, wide-spectrum language and the mechanism for dealing with real-time constraints in the target code. The motivation for our work comes from the real-time refinement calculus of Mahony (1992), which allows not only the specification of real-time systems, but the refinement of a specification into a set of truly parallel processes. Our work complements Mahony's by allowing individual processes to be refined to sequential code.

In this paper, we use the calculus to develop a program that calculates the size of a single kiwifruit from the time it takes to pass through a light beam. Section 3 specifies the requirements, and Sections 4 and 5 show how they can be refined into a real-time program. The timing analysis of the program is discussed in Section 6.

The example illustrates the main features and difficulties of the calculus. In order to develop the obvious program for the task, we needed to develop a novel loop introduction law, that does not require a variant for termination, but uses a fixed time deadline instead.

## 2 THE SEQUENTIAL REAL-TIME REFINEMENT CALCULUS

An obvious difference between our calculus and the standard refinement calculus is that our calculus has a special variable, $\tau$, that represents the *current time*. Each command advances $\tau$ to reflect the passage of time.

We distinguish between three types of variables: external inputs, external outputs and local variables. Input variables correspond to input device registers. They are not under the control of the program, but may be read via a special command. Output variables correspond to output device registers. They are under the control of the program, but differ from local variables in that changes to output variables are externally visible. We write real-time specification commands as $\star\,\vec{v}\colon [A\ ,\ E]$, where

- $\vec{v}$ is a list of variables that may be modified by the command. These variables must be a subset of the outputs and local variables of the program, (variables that correspond to external inputs are read-only). Since $\tau$ is modified by almost every command, it is implicitly in the frame of every command and is not explicitly included in the frame.

- $A$ is a predicate that gives the *assumptions* the command may make about the variables. Within $A$, $\tau$ refers to the time that the command starts execution, but $A$ may also explicitly reference the value of variables at other times.
- $E$ is a predicate that gives the *effect* that the command is to achieve by modifying the variables in the frame. Within $E$, $\tau_0$ and $\tau$ refer to the time that the command starts and finishes execution, respectively.

A key change from the standard refinement calculus is our treatment of variables; they are modelled as functions from time to their value at that time. This allows a real-time specification command to constrain not only the final value of variables, but also their values at other times. Given a variable $v$, its value at time $t$ is written as $v(t)$. However, to recover the *look and feel* of the standard refinement calculus, we follow the convention that an unindexed variable $v$ in a predicate means $v(\tau)$, and $v_0$ means $v(\tau_0)$. The semantics of the specification command are given in Appendix 1.

The basic refinement rules from the standard refinement calculus carry over into our calculus (Hayes & Utting 1997*b*), although some have additional side-conditions to restrict the times at which predicates refer to variables (these side-conditions are satisfied trivially for untimed predicates that use the $v$ and $v_0$ convention described above). We use an extended target language that includes several real-time commands, such as:

- **gettime**$(x) \overset{\text{def}}{=} \star x\colon [\text{true}\ , x \in [\tau_0 \dots \tau]]$.
  This puts a timestamp (an approximation to $\tau$) into the variable $x$. The notation $[\tau_0 \dots \tau]$ stands for the closed interval from the start time of the command, $\tau_0$, to the finish time, $\tau$.
- **read**$(e, x) \overset{\text{def}}{=} \star x\colon [\text{true}\ , x \in e(\!|\ [\tau_0 \dots \tau]\ |\!)]$.
  This copies a snapshot of an external input $e$ into the local variable $x$. The notation $e(\!|\ [\tau_0 \dots \tau]\ |\!)$ stands for the set of values of the input variable $e$ over the time interval $[\tau_0 \dots \tau]$.
- **delay until** $D \overset{\text{def}}{=} \star\colon [\text{true}\ , \tau \geq D]$.
  This is similar to the delay-until command in many programming languages. The delay command finishes execution at or after the (absolute) time $D$.
- **deadline** $D \overset{\text{def}}{=} \star\colon [\text{true}\ , \tau_0 = \tau \wedge \tau \leq D]$.
  The **deadline** command is novel to our approach and allows timing constraints to be expressed abstractly in the extended programming language (Hayes & Utting 1997*a*). It takes no time and must terminate at or before time $D$. Hence, it requires the preceding portion of the program to be complete by time $D$.
- **idle** $\overset{\text{def}}{=} \star\colon [\text{true}\ , \tau_0 \leq \tau]$.
  The **idle** command may take time but does not change any variables. Note that external inputs may change during the time it takes to execute.

The presence of deadline commands means that a separate program analysis is required to guarantee that the deadlines will be met by the machine code generated for the program by a compiler. If the program analysis cannot guarantee that a deadline will be met, the program is rejected. Note that it is important to analyse timing *after* compilation, because no analysis of the higher-level program can take into account low-level aspects such as register allocation and code optimisation within a compiler, or instruction pipelining and cache memories within processors, which together can significantly affect the timing characteristics of a program.

## 3 ON MEASURING THE SIZE OF A KIWIFRUIT

Imagine that a single kiwifruit is moving along on a conveyor belt and goes through a light beam sensor that is connected into an embedded microcomputer. A program on the microcomputer polls the boolean *sensor* status and uses a real-time clock to determine the approximate start and end times of when the light beam is broken. When the kiwifruit breaks the light beam the *sensor* rises (to true) and after the kiwifruit passes the light beam is re-established and the *sensor* falls (to false). From those times, and the known speed of the conveyor belt, the size of the kiwifruit can be computed.

The following declarations define the environment in which our program will be used. As well as documenting the type of variables representing physical quantities, we also document their units (Hayes & Mahony 1995). Variables of type `time` are in units of nanoseconds.

**const** $speed \overset{\text{def}}{=} 10\,\text{m/s}$ -- The speed of the conveyor belt
**const** $react \overset{\text{def}}{=} 100\,\mu\text{s}$ -- Desired reaction time after the kiwifruit passes

The *sensor* is an external input to the program. It is derived from the light beam detecting hardware. Its value over time is not under the control of the program, but the program does make assumptions about the behaviour of the input sensor.

**input** $sensor : \mathbb{B}$
**var** $size : \mathbb{N}\,\text{nm}$ -- Returned size of the kiwifruit in nanometres

The logical constants *rises* and *falls* are introduced solely for specification purposes. They denote, respectively, the exact time at which the sensor rises and falls. Logical constants may not be used in the final program, except in assertions and deadlines.

**con** *rises*, *falls* : `time`
**const** $minsep \overset{\text{def}}{=} 1\,\text{ms}$ -- Minimum separation between *rises* and *falls*
-- Corresponds to length of 10 mm

The program may assume that the rise time precedes the fall time by *minsep*.

$$\star\,\{rises + minsep \leq falls\} \qquad (1)$$

Because (1) does not refer to any variables modified by the program, it may be assumed to hold throughout the program. To avoid cluttering the specification and the refinement below, we state (1) once here and assume it where needed. Logically it could be conjoined to the assumption of the specification (2) below and passed through the refinement as necessary. The sensor detects (is true) when the light beam is interrupted by the passing kiwifruit.

$$SENSOR(\tau) \overset{\text{def}}{=} (\forall t : [\tau \ldots falls + react] \bullet sensor(t) = \text{true} \Leftrightarrow t \in [rises \ldots falls])$$

The top level specification is:

$$\star\, size\colon \begin{bmatrix} \tau \leq rises & \tau \leq falls + react \\ SENSOR(\tau) & size \in speed * (falls - rises) \pm 1\,\text{mm} \end{bmatrix} \tag{2}$$

## 4 REFINEMENT OF THE KIWIFRUIT SIZER

Appendix 1 provides a summary of refinement laws used within this paper.

As a first refinement step, it is useful to separate out the initial time assumption, $\tau \leq rises$, and the final deadline requirement, $\tau \leq falls + react$, so that we may concentrate on implementing the remaining functionality.

(2)
$\sqsubseteq$ Law 6 (separate assumption); Law 10 (separate deadline)
$\star \{\tau \leq rises\}$ ;
$\star\, size\colon [SENSOR(\tau)\, ,\, size \in speed * (falls - rises) \pm 1\,\text{mm}]$ ; (3)
**deadline** $falls + react$

The assertion and deadline become part of the final program, and we are left to refine the specification command (3).

The next refinement step is obvious – we want to split the program into three parts: the first two will determine the rise and fall times and the third part computes the size output. We introduce a constant *err* as an error bound on the time to detect the change in the sensor. We leave the actual value of *err* to be determined later, but at this stage we require that it is less than both *minsep* and *react*, so that the sensor is stable for at least a period of *err* after it changes.

$$\textbf{const}\ err \in \{t : \texttt{time} \mid t \leq minsep \wedge t \leq react\} \tag{4}$$

Local variables *riset* and *fallt* are used to communicate the approximations to the rise and fall times between the parts. When introducing a local variable one cannot assume that the allocation and deallocation of the local variable take no time. Hence, an assumption that held immediately before the allocation of a local variable, may not hold immediately after the allocation. However, during the allocation of a new variable the other program variables (not including the external inputs) are stable and time can only increase. A predicate that remains true under these circumstances is referred to as being *idle-stable*. It is invariant over the execution of an **idle** command. The predicate $SENSOR(\tau)$ is idle-stable. It states a property that holds at every instant of time from $\tau$ up to $falls + react$. Hence, $SENSOR(x)$ also holds for all values of $x$ later than $\tau$.

The time taken for the allocation and deallocation of variables may also affect the effect of a specification command. In (3) the effect only refers to constants and the program variable *size*. Because *size* is stable during the deallocation of the variables, the effect will still be true after the deallocation. Note that we follow Morgan's (1994) convention of using a $\lhd$ symbol to mark the specification statement that is refined next – the context of that statement becomes the context of the next refinement.

(3)

$\sqsubseteq$ Law 11 (introduce variable)

$$\begin{array}{l} [\![\textbf{var}\ riset, fallt : \texttt{time}; \\ \quad \star\, size, riset, fallt\text{:}\ \left[SENSOR(\tau)\ ,\ size \in speed * (falls - rises) \pm 1\,\text{mm}\right] \qquad \lhd \\ ]\!] \end{array}$$

$\sqsubseteq$ Law 9 (simple sequential composition) x 2; Law 5 (remove from frame) x 3

$$\star\, riset\text{:}\ \left[SENSOR(\tau)\ ,\ \begin{array}{l} SENSOR(\tau) \\ riset \in [rises \ldots rises + err] \end{array}\right]; \tag{5}$$

$$\star\, fallt\text{:}\ \left[\begin{array}{l} SENSOR(\tau) \\ riset \in [rises \ldots rises + err] \end{array}\ ,\ \begin{array}{l} riset \in [rises \ldots rises + err] \\ fallt \in [falls \ldots falls + err] \end{array}\right]; \tag{6}$$

$$\star\, size\text{:}\ \left[\begin{array}{l} riset \in [rises \ldots rises + err] \\ fallt \in [falls \ldots falls + err] \end{array}\ ,\ size \in speed * (falls - rises) \pm 1\,\text{mm}\right] \tag{7}$$

The refinement of (5) is more interesting. The goal is to determine the time at which the light beam sensor rises. Our first step is to massage the specification to reflect this more specific goal. Firstly, the effect $SENSOR(\tau)$ is immediate from the assumption because $SENSOR(\tau)$ is an idle-stable predicate that does not refer to the variable in the frame, *riset*. Secondly, we weaken the assumption so that we only consider the rising phase of the signal.

(5)

$\sqsubseteq$ Law 4 (strengthen effect); Law 3 (weaken assumption)

$$\star\, riset\text{:}\ \left[\begin{array}{l} \forall\, t : [\tau \ldots rises + err] \bullet \\ \quad sensor(t) = \text{true} \Leftrightarrow rises \le t \end{array}\ ,\ riset \in [rises \ldots rises + err]\right] \tag{8}$$

We make use of the following definition which is written to allow detection of either a rising edge (the new sensor value we are required to detect, *srqd*, is true), or a falling edge change (*srqd* = false).

$$CHNG(chngs, srqd) \overset{\text{def}}{=} \forall\, t : [\tau \ldots chngs + err] \bullet sensor(t) = srqd \Leftrightarrow chngs \le t$$

To refine the detection of the rise of the sensor, we make use of a procedure, *detect_chng*. The first step is a refinement equivalence ($\sqsubseteq$).

(8)
$\sqsubseteq \star\, riset$: $\left[CHNG(rises, \text{true})\ ,\ riset \in [rises \ldots rises + err]\right]$
$\sqsubseteq$ parametrized procedure
*detect_chng*(*rises*, true, *riset*)

The procedure *detect_chng* can be used to detect either a rising or falling edge of the sensor value depending on the parameter *srqd*, which is the required new value of the sensor.

$$\textbf{proc}\ detect\_chng(\textbf{con}\ chngs : \texttt{time};\ \textbf{value}\ srqd : \mathbb{B};\ \textbf{result}\ chngt : \texttt{time}) \overset{\text{def}}{=}$$
$$\star\, chngt\text{: } \left[CHNG(chngs, srqd)\ ,\ chngt \in [chngs \ldots chngs + err]\right] \qquad (9)$$

Before giving a refinement of *detect_chng* (Section 5) we complete the refinement of the program. Detecting the falling edge also makes use of the procedure *detect_chng*.

(6)
$\sqsubseteq$ Law 4 (strengthen effect); Law 3 (weaken assumption)
$\star\, fallt$: $\left[CHNG(falls, \text{false})\ ,\ fallt \in [falls \ldots falls + err]\right]$
$\sqsubseteq$ *detect_chng*(*falls*, false, *fallt*)

Once both the rise time and the fall time have been determined, the size of the kiwifruit can be calculated.

(7)
$\sqsubseteq$ Law 7 (assignment)
$size := speed * (fallt - riset)$

The units of the right side expression are (m / s) $*$ ns = nm, which matches the units of *size*. The proof obligation for the last step is

$speed * (fallt - riset) \in speed * (falls - rises) \pm 1\,\text{mm}$
$\equiv fallt - riset \in (falls - rises) \pm (1\,\text{mm}/speed)$
$\Leftarrow$ from the assumptions on *riset* and *fallt*
$[falls - (rises + err) \ldots (falls + err) - rises] \subseteq (falls - rises) \pm (1\,\text{mm}/speed)$
$\equiv (falls - rises) \pm err \subseteq (falls - rises) \pm (1\,\text{mm}/speed)$
$\equiv err \le (1\,\text{mm}/speed)$
$\equiv err \le 100\,\mu\text{s}$

## 5 REFINEMENT OF DETECTION OF A SENSOR CHANGE

The following code is the implementation of *detect_chng* (the refinement to this code follows shortly). Note that this code is similar to what a programmer would write for this task, except that *deadline* commands have been added to make the timing requirements of the program explicit.

```
proc $detect\_chng$(con $chngs$ : time; value $srqd$ : $\mathbb{B}$; result $chngt$ : time) $\sqsubseteq$
$[\![$var $sens$ : $\mathbb{B}$ ;
    read($sensor, sens$) ;
    deadline $chngs + err$ ;
    $\star$do $sens \neq srqd \rightarrow$
        read($sensor, sens$) ;
        deadline $chngs + err$
    od ;
    gettime($chngt$) ;
    deadline $chngs + err$
$]\!]$
```

Aside: The above code corresponds to what is usually referred to as busy waiting. In the current context we are assuming a single sequential process, and hence busy waiting is acceptable. It would also be acceptable to include a (sampling) delay in the above code, provided the delay time is sufficiently short to allow the program to meet its deadlines. We have not done so here.

The final deadline ensures that the captured change time is within its allowable error bounds. Given the last deadline, the first two deadlines seem redundant, but they are essential for the correct operation of the program. The final deadline command will not be reached by the program if the loop does not terminate, and if the deadline is not reached, it does not have to be met. Consider the case where we want to detect the rise of the sensor. Without the first deadline the initial code could take so long that the first read command completely misses the period when the sensor is true. (This is unlikely in practice, but not excluded by the meanings we have given to the programming constructs.) If the first read completely misses the sensor when it is raised, then the loop will not be guaranteed to terminate, because our assumptions only guarantee one period when the sensor is raised. Hence the final deadline may never be reached. The deadline within the loop has a similar purpose. It guarantees that the body of the loop will not take so long that the read of the sensor misses the raised period of the sensor. Again, if the raised period were missed the loop would not be guaranteed to terminate, and the final deadline would not be reached. As we shall see shortly, the refinement process needs to introduce the deadlines in order to guarantee that the specification will be met. In addition, the deadline within the loop is also used to guarantee its termination.

Before continuing, we invite the reader to informally analyse this code (e.g., unroll the loop once or twice) to determine the longest path through the code that can be executed from time *chngs* to the **gettime** command. Note how the analysis depends on consequences of $CHNG(chngs, srqd)$, such as the shape of the *sensor* waveform. Our real-time calculus enables us to make these implicit timing assumptions and invariants explicit, so that we can provide an invariant for the loop and so that the timing analysis phase has a manageable task. With these goals in mind, it turns out (after several attempts!) that a suitable loop invariant is:

$$INV \overset{\text{def}}{=} CHNG(chngs, srqd) \wedge (sens = srqd \Rightarrow chngs \leq \tau)$$

The first step is to introduce a local variable, *sens*, that is used to capture the sensor's values.

$$
\begin{array}{ll}
(9) & \\
\sqsubseteq \text{Law 11 (introduce variable)} & \\
\quad [\![\, \mathbf{var}\ sens : \mathbb{E}; & \\
\qquad \star\, chngt, sens\colon \left[CHNG(chngs, srqd)\ ,\ chngt \in [chngs \ldots chngs + err]\right] & (10) \\
\quad ]\!] &
\end{array}
$$

Next we set up a loop that searches for the change of the sensor and then we capture the time. The introduction of the loop makes use of the following law, that does not have a conventional variant to show termination. Our semantics of the loop guarantees that each iteration takes a minimum amount of time, due for example to loop overheads and guard evaluation. Termination is guaranteed by the fact that time increases by at least that minimum amount on each iteration, and every iteration of the body of the loop is bounded by the same constant time limit. The time limit, $L$, is required to be frame-stable with respect to the frame, $\vec{x}$, of the loop. That guarantees that $L$ will remain invariant (stable) during the execution of the loop. If $L$ does not refer to variables in the frame, or $\tau$, or external inputs, then it is frame-stable.

**Law 1 (iteration with deadline)** *Given an idle-stable invariant property, INV, a deadline expression L, which is frame-stable with respect to the frame $\vec{x}$, and an idle-stable expression G, where none of G, L and INV contain references to $\tau_0$ or zero-subscripted variables,*

$$\star\vec{x}\colon \left[INV \wedge \tau \leq L, \neg\, G \wedge INV\right] \sqsubseteq \star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV, INV \wedge \tau \leq L\right]\ \mathbf{od}$$

The semantics of loops and the proof of this law are contained in Appendix 2. Before introducing the loop, we separate the initialisation, loop, and final capture of the change time.

$$
\begin{array}{ll}
(10) & \\
\sqsubseteq \text{Law 9 (simple sequential composition)} \times 2;\ \text{Law 5 (remove from frame)} \times 3 & \\
\quad \star\, sens\colon \left[CHNG(chngs, srqd)\ ,\ INV \wedge \tau \leq chngs + err\right]; & (11) \\
\quad \star\, sens\colon \left[INV \wedge \tau \leq chngs + err\ ,\ INV \wedge sens = srqd\right]; & (12) \\
\quad \star\, chngt\colon \left[INV \wedge sens = srqd\ ,\ chngt \in [chngs \ldots chngs + err]\right] & (13)
\end{array}
$$

The initialisation establishes the loop invariant. In the second refinement step below, because of the assumption, if the value sensed is equal to *srqd* then the completion time of the command must be after *chngs*. If the sensed value is not equal to *srqd* the completion time may be either before or after *chngs*, but that does not matter.

(11)

$\sqsubseteq$ Law 4 (strengthen effect)

$\star\, sens\colon \begin{bmatrix} \forall t : [\tau \ldots chngs + err] \bullet & sens = srqd \Rightarrow chngs \le \tau \\ \quad sensor(t) = srqd \Leftrightarrow chngs \le t \text{ ' } \tau \le chngs + err & \end{bmatrix}$

$\sqsubseteq$ Law 4 (strengthen effect)

$\star\, sens\colon \begin{bmatrix} \forall t : [\tau \ldots chngs + err] \bullet & sens \in sensor(\!| [\tau_0 \ldots \tau] |\!) \\ \quad sensor(t) = srqd \Leftrightarrow chngs \le t \text{ ' } \tau \le chngs + err & \end{bmatrix}$

$\sqsubseteq$ Law 3 (weaken assumption); Law 10 (separate deadline); Definition of read

**read**$(sensor, sens)$ ; **deadline** $chngs + err$

Now we can introduce the loop.

(12)

$\sqsubseteq$ Law 1 (iteration with deadline)

```
⋆ do $sens \ne sqrd \rightarrow$
      ⋆ $sens\colon [sens \ne srqd \land INV \text{ , } INV \land \tau \le chngs + err]$      ◁
  od
```

$\sqsubseteq$ Law 3 (weaken assumption)

$\star\, sens\colon [CHNG(chngs, srqd) \text{ , } INV \land \tau \le chngs + err]$

$\sqsubseteq$ as for the refinement of (11)

**read**$(sensor, sens)$ ; **deadline** $chngs + err$

On termination the change of the sensor has been detected. It only remains to capture the current time, before the allowed error bound.

(13)

$\sqsubseteq$ Law 3 (weaken assumption)

$\star\, chngt\colon [chngs \le \tau \text{ , } chngt \in [chngs \ldots chngs + err]]$

$\sqsubseteq$ Law 4 (strengthen effect)

$\star\, chngt\colon [chngs \le \tau \text{ , } chngt \in [\tau_0 \ldots \tau] \land \tau \le chngs + err]$

$\sqsubseteq$ Law 10 (separate deadline); Law 3 (weaken assumption); Definition of gettime

**gettime**$(chngt)$ ; **deadline** $chngs + err$

The final program, with the procedure *detect_chng* inlined is shown in Figure 1.

## 6 TIMING ANALYSIS

The final phase is to determine the time constraints on paths through the code in order to guarantee that all deadlines will be met. For each deadline command we

```
$A : \star \{\tau \leq rises\}$ ;
  $[\![$ var $riset, fallt$ : time;
    $[\![$ var $sens : \mathbb{B}$;
        read($sensor$, $sens$);
      $B$ : deadline $rises + err$;
        $\star$ do invariant $CHNG(rises, \text{true}) \wedge (sens = \text{true} \Rightarrow rises \leq \tau)$
        $sens \neq \text{true} \rightarrow$
            read($sensor$, $sens$);
          $C$ : deadline $rises + err$
        od;
        gettime $riset$;
      $D$ : deadline $rises + err$
    $]\!]$;
    $[\![$ var $sens : \mathbb{B}$;
        read($sensor$, $sens$);
      $E$ : deadline $falls + err$;
        $\star$ do invariant $CHNG(falls, \text{false}) \wedge (sens = \text{false} \Rightarrow falls \leq \tau)$
        $sens \neq \text{false} \rightarrow$
            read($sensor$, $sens$);
          $F$ : deadline $falls + err$
        od;
        gettime $fallt$;
      $G$ : deadline $falls + err$
    $]\!]$;
    $size := speed * (fallt - riset)$
  $]\!]$;
$H$ : deadline $falls + react$
```

**Figure 1** Final program with the procedure inlined.

consider the paths through the program that terminate at the deadline. For each such path we need to determine a time constraint on the execution time of the path that guarantees that the deadline will be met. Grundon, Hayes & Fidge (1998) have formalised the details of timing path analysis, but here we present a brief overview of the process for the program in Figure 1.

The first path we consider is A–B, which has a start time before* *rises* and must

*To simplify presentation, 'before' is taken to mean 'no later than' throughout this section.

complete before *rises* + *err*. That gives a time constraint for the path of *err*. Both path A–C, which enters the loop on the first iteration, and path A–D, which does not enter the loop at all, have the same constraint of *err*.

The path starting at A that enters the loop for the first iteration is not the only path that ends at C. The other possibilities come from the repetition of the loop. These paths require some intricate reasoning to determine suitable time constraints. Our goal is to determine timing constraints on paths through the code that guarantee that the deadline at point C is reached before its deadline of *rises* + *err*. Because the deadline is within the loop body, we know on entry to the loop that *sens* is false, which implies that the previous read must have commenced before time *rises*. Hence our constraint is that the path from the previous read, around the loop through the current read, and finishing at the deadline at C, must take time less than $rises + err - rises = err$.

There are paths from D to E, F and G. All start no later than *rises* + *err* and must complete before *falls* + *err*. That gives them a time constraint of *falls* − *rises*. However, we are guaranteed by (1) that $falls - rises \geq minsep$, and hence we can use *minsep* as our time constraint.

The remainder of the paths for calculating the fall time of the sensor are similar to those for calculating the rise. We do not discuss them in detail here.

The final path that we consider is G–H. It has a start time before *falls* + *err* and must complete before *falls* + *react*. This gives a time constraint of *react* − *err*. The specification gives *react* as $100\,\mu$s, and our refinement requires that $err \leq 100\,\mu$s. The constant *err* appears in the constraints of many paths. It can be chosen up to the limit of $100\,\mu$s so that the paths can meet their timing constraints.

Provided we can show that the machine code generated for each of the above paths satisfies the corresponding time constraint, then we can guarantee all deadlines will be met. There has been considerable research in the real-time community on timing analysis of such machine code sequences (Lim et al. 1995).

## 7 CONCLUSIONS

The main advantage of the sequential real-time refinement calculus presented here is that, to developers, it appears to be a straightforward extension of the standard refinement calculus. Although it has a different underlying semantics, most of the standard refinement laws carry over, and the real-time extended programming language is a superset of the standard target language. In practice, a development in the real-time calculus is similar to standard refinement calculus development, but with the addition of steps to separate out timing constraints and refine them into real-time language constructs.

However, even with this strong connection to the standard calculus, our experience so far suggests that programs that rely heavily on timing behaviour for their correctness are quite challenging to develop in our calculus. Finding loop invariants and sequential composition intermediate predicates seems more difficult than in the standard calculus. We suspect that this is partly because we have had more experience with the standard calculus, and partly because real-time programming is intrinsically difficult. Certainly the timing analysis phase is an additional requirement with its own intricacies. Anyway, it is exciting to have a calculus that allows the subtle aspects of real-time programs to be formally proved, just as the standard calculus "dots the i's and crosses the t's" of ordinary sequential programming.

## REFERENCES

Back, R.-J. (1980) Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam.

Grundon, S., Hayes, I. J. & Fidge, C. J. (1998) Timing constraint analysis, *in* C. McDonald, ed., 'Computer Science '98: Proc. 21st Australasian Computer Science Conf. (ACSC'98), Perth, 4–6 Feb.', Springer-Verlag, pp. 575–586.

Hayes, I. J. & Mahony, B. P. (1995) 'Using units of measurement in formal specifications'. *Formal Aspects of Computing* 7(3), 329–347.

Hayes, I. & Utting, M. (1997*a*) Coercing real-time refinement: A transmitter, *in* D. J. Duke & A. S. Evans, eds, 'BCS-FACS Northern Formal Methods Workshop', Electronic Workshops in Computing, Springer Verlag. URL http://www.springer.co.uk/ewic/workshops/NFM96/.

Hayes, I. & Utting, M. (1997*b*) A sequential real-time refinement calculus. Technical Report UQ-SVRC-97-33, Software Verification Research Centre, The University of Queensland, URL http://svrc.it.uq.edu.au.

Lim, S.-S., Bae, Y. H., Jang, G. T., Rhee, B.-D., Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S.-M. & Kim, C. S. (1995) 'An accurate worst case timing analysis for RISC processors'. *IEEE Trans. on Software Eng.* **21**(7), 593–604.

Mahony, B. P. (1992) The Specification and Refinement of Timed Processes. PhD thesis, Department of Computer Science, The University of Queensland.

Morgan, C. (1994) *Programming from Specifications.* second edn, Prentice Hall.

Utting, M. & Fidge, C. (1996) A real-time refinement calculus that changes only time, *in* He Jifeng, ed., 'Proc. 7th BCS/FACS Refinement Workshop', Electronic Workshops in Computing, Springer. URL http://www.springer.co.uk/eWiC/Workshops/7RW.html.

Utting, M. & Fidge, C. J. (1997) Refinement of infeasible real-time programs, *in* 'Proc. Formal Methods Pacific '97', Discrete Mathematics and Theoretical Computer Science, Springer, Wellington, New Zealand, pp. 243–262.

## APPENDIX 1 LAWS FOR LANGUAGE CONSTRUCTS

The following laws are extracted from an earlier paper (Hayes & Utting 1997*b*) that gives the semantics of the real-time language constructs, as well as a more comprehensive set of laws.

*Unindexed variables in predicates and expressions* In a predicate, unindexed variables of the form $v$ stand for $v(\tau)$, and variables of the form $v_0$ stand for $v(\tau_0)$. We introduce the notation $R @ (\tau_0, \tau)$ to stand for the predicate $R$ with every unindexed occurrence of a variable, $v$, replaced by $v(\tau)$ and every occurrence of $v_0$ replaced by $v(\tau_0)$. Note that $R$ may contain explicit indexed references to variables at times other than $\tau$; these are not affected by the '@' operator. The operator '@' has a lower precedence than all the normal logical operators, but a higher precedence than '$\equiv$' and '$\Rrightarrow$'. For predicates, such as assumptions, that do not contain any zero-subscripted variables, we use the notation $P @ \tau$. If there are no occurrences of $\tau_0$ or zero-subscripted variables in $P$ then $P @ (\tau_0, \tau) \equiv P @ \tau$. The operator '@' distributes over logical operators.

*Specifications* The assumptions of a specification command determine the range of possible values of variables over time, as well as the start time of the command. The effect further constrains the values of variables over time, as well as constraining the finish time of the command. Program variables (local variables and outputs) not in the frame of a specification are stable over its execution. Given a variable, $v$, and a set of times, $S$,

$$stable(v, S) \overset{\text{def}}{=} (\forall t, u : S \bullet v(t) = v(u))$$

The meaning of a specification command is given with respect to a given environment, where an environment just records the variables (inputs, outputs and local variables) that are in scope. We use $\rho$ to stand for an environment, and $\hat{\rho}$ to stand for the program variables (outputs and local variables) within $\rho$. The meaning function, $\mathcal{M}_\rho$, gives the meaning of a real-time construct in environment $\rho$ in terms of a predicate transformer determined by a standard refinement calculus construct.

**Definition 2 (specification)** *A specification command, $\star\vec{x}$: $[P, R]$, is well formed in an environment, $\rho$, provided (i) the frame, $\vec{x}$, is a vector of program variables (outputs and local variables), (ii) $P$ is a predicate involving the variables in the environment plus $\tau$, and (iii) $R$ is a predicate involving variables in the environment plus $\tau_0$, $\tau$ and zero-subscripted versions of variables in the environment. The meaning of a well-formed specification command is given by the following*

$$\mathcal{M}_\rho\,(\star\vec{x}\colon [P, R]) \overset{\text{def}}{=} \tau\colon [P @ \tau,\, R @ (\tau_0, \tau) \wedge \tau_0 \le \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\tau_0 \ldots \tau])]$$

*where $\hat{\rho} \setminus \vec{x}$ stands for the program variables, $\hat{\rho}$, with any elements in the frame $\vec{x}$ removed.*

Note that $\tau$, unlike other variables, is not itself a function of time. The frame, $\vec{x}$, of the real-time command does not appear in the frame of the equivalent standard command. Instead, those program variables that are not in the frame are constrained to be stable for its duration, and the program variables in the frame are only constrained by the effect of the specification, $R$. In the assumption and effect of a specification command it is permissible to include both explicitly indexed references and unindexed references to the same variable. For more details on the encoding of real-time specifications the reader is referred to Utting & Fidge (1997).

The refinement rules for weakening an assumption and strengthening an effect carry over to the real-time refinement calculus.

**Law 3 (weaken assumption)** *Provided* $P @ \tau \Rrightarrow P' @ \tau$,

$$\star\vec{x}\colon [P, R] \sqsubseteq \star\vec{x}\colon [P', R] .$$

When applying this law we can use the fact that from $P \Rrightarrow P'$ one can deduce that $P @ \tau \Rrightarrow P' @ \tau$. That gives a special case of the law for dealing with properties that are not time dependent.

**Law 4 (strengthen effect)** *Given an environment, $\rho$, provided*

$$(P @ \tau_0) \wedge (R' @ (\tau_0, \tau)) \wedge \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\tau_0 \ldots \tau]) \Rrightarrow R @ (\tau_0, \tau)$$

*then*

$$\star\vec{x}\colon [P, R] \sqsubseteq \star\vec{x}\colon [P, R']$$

*where $\hat{\rho} \setminus \vec{x}$ stands for the program variables (local variables and outputs) of the environment $\rho$, minus the variables in the frame, $\vec{x}$.*

In the case where the properties are not time dependent, a special case of the proviso is, $P_0 \wedge R' \Rrightarrow R$, where $P_0$ stands for the predicate $P$ with all occurrences of $\tau$ replaced by $\tau_0$, and all unindexed occurrences of every variable, $v$, that is in the frame or is an external input, replaced by $v_0$.

**Law 5 (remove from frame)** *Given disjoint vectors of program variables, $\vec{x}$ and $\vec{v}$,*

$$\star\vec{x}, \vec{v}\colon [P, R] \sqsubseteq \star\vec{v}\colon [P, R]$$

*Assertions* Assertions may state assumptions about the variables at the point at which they occur. Hence in an assertion, $\star \{A\}$, an unindexed reference to a variable, $v$, is interpreted as $v(\tau)$. Assertions can also state assumptions about the value of variables at other times by using explicit indices. Assertions take no time, and hence there is no need for $\tau_0$ or zero-subscripted variables, within assertions.

**Law 6 (separate assumption)**

$$\star\vec{x}\colon [U \wedge P, R] \sqsubseteq \star \{U\} ; \star\vec{x}\colon [P, R]$$

*Assignment* As the evaluation of the expressions in an assignment takes time, we require that the expressions in assignments are idle-stable (time-independent).

**Law 7 (assignment)** *Given an environment, $\rho$, a frame, $\vec{x}$, such that $\vec{x}$ is contained in the program variables, $\hat{\rho}$, and a vector, $\vec{D}$, of idle-stable expressions, provided*

$$(P @ \tau_0) \wedge (\vec{x} @ \tau) = (\vec{D} @ \tau_0) \wedge \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\tau_0 \ldots \tau]) \Rrightarrow R @ (\tau_0, \tau)$$

*then*

$$\star\vec{x}: [P, R] \sqsubseteq \vec{x} := \vec{D}.$$

If the properties do not involve time then the following special case of the proviso can be used: $P_0 \wedge \vec{x} = \vec{D}_0 \Rrightarrow R$.

*Logical constants* Note that logical constants are not implicit functions of time.

**Law 8 (logical constant)** *Provided $(\exists\, u : T \bullet P @ \tau)$ and $u$ does not occur in $C$ or the variables in the environment,*

$$C \sqsubseteq |[\ \textbf{con}\ u : T \bullet \star\{P\}\ ;\ C\ ]|$$

*Refinement to a sequential composition* The refinement of a specification command to a sequential composition of specification commands follows the same approach as in the standard refinement calculus. One must devise an intermediate predicate $Q$ that holds on termination of the first component, and hence also for the assumption of the second component. Because we have assumed that there is no time delay between the execution of the two commands, $\tau$ in the effect of the first component refers to the same time as $\tau$ in the assumptions of the second component.

**Law 9 (simple sequential composition)** *Provided $Q$ and $R$ do not involve $\tau_0$ or zero-subscripted variables,*

$$\star\vec{x}: [P, R] \sqsubseteq \star\vec{x}: [P, Q]\ ;\ \star\vec{x}: [Q, R]$$

A timing deadline in the effect of a specification command may be separated out into a deadline command.

**Law 10 (separate deadline)** *Provided $D$ is a time-valued expression, which may include references to logical constants but no references to $\tau_0$ or zero-subscripted variables,*

$$\star\vec{x}: [P, R \wedge \tau \leq D] \sqsubseteq \star\vec{x}: [P, R]\ ;\ \textbf{deadline}\ D$$

*Local variables* The definition of a local variable block in the real-time language involves expanding the environment for the command within the block. The allocation and deallocation of local variables may take time. Hence we need stability requirements on the assumption and effect in the variable introduction law.

**Law 11 (introduce variable)** *Given an environment, $\rho$, such that $v$ does not occur in the variables of $\rho$ (and hence does not occur free within $P$ or $R$ or in $\vec{x}$), provided $T$ is nonempty, $P$ is idle-stable and $R$ is both pre- and post-idle-stable, then*

$$\star\vec{x}\colon \left[P, R\right] \sqsubseteq |[\ \mathbf{var}\ v : T \bullet \star v, \vec{x}\colon \left[P, R\right]\ ]|$$

A predicate $R$ is pre-idle-stable (post-idle-stable) if when $R$ is used as an effect of a specification command, the specification command can be prefixed (postfixed) by an idle command and the result is a refinement of the original specification command (Hayes & Utting 1997*b*).

## APPENDIX 2 ITERATION

We distinguish between the real-time iteration command ($\star$**do**) and the standard iteration command (**do**) within this section and use the latter in the definition of the former as a way of reusing its predicate transformer.

The guard of an iteration is required to be idle-stable so that it is stable during its evaluation. To account for the delay to evaluate the guard an idle command with a minimum execution time ($d1$) is used,

$$\mathbf{idle}_{\geq d1} \overset{\text{def}}{=} \star\colon \left[\text{true}\ , \tau - \tau_0 \geq d1\right]$$

and to account for exiting the loop, an idle command with a maximum execution time ($d2$) is used,

$$\mathbf{idle}_{\leq d2} \overset{\text{def}}{=} \star\colon \left[\text{true}\ , \tau - \tau_0 \leq d2\right]$$

Of course, both $d1$ and $d2$ are dependent on the implementation, but our definition does not rely on the particular values of these, just that such constants exist. To allow for the case when the guard is initially false, an **idle** is added after the loop to allow the loop to take some time in this case.

**Definition 12 (iteration)** *Given an environment, $\rho$, and an idle-stable expression, $G$, which does not contain references to $\tau_0$ or zero-subscripted variables,*

$$\begin{array}{l}\mathcal{M}_\rho\,(\star\mathbf{do}\ G \rightarrow C\ \mathbf{od}) \overset{\text{def}}{=} \\ \quad [\!]\, d1, d2 : \mathit{Time} \mid d1 > 0 \wedge d2 > 0 \bullet \\ \qquad \mathbf{do}\ G\ @\ \tau \rightarrow \mathcal{M}_\rho\,(\mathbf{idle}_{\geq d1};\ C;\ \mathbf{idle}_{\leq d2})\ \mathbf{od} \circ \mathcal{M}_\rho\,(\mathbf{idle})\end{array}$$

*where '$[\!]$' is generalised nondeterministic choice and '$\circ$' is standard sequential composition.*

The following loop introduction rule does not make use of a conventional variant to show termination. Instead it makes use of a fixed deadline that does not change during the execution of the loop. When combined with the inevitable progress of time due to execution of the loop, this has the same effect as a variant.

**Law 13 (iteration timing with deadline)** *Given an idle-stable invariant property, INV, a deadline expression, L, which is frame-stable with respect to the frame $\vec{x}$, idle-stable expressions G and D, and an environment, ρ, where none of G, D, L and INV contain references to $\tau_0$ or zero-subscripted variables,*

$$\begin{array}{l} \star\vec{x}\colon [INV \wedge D = \tau \leq L, \neg\, G \wedge INV'] \\ \sqsubseteq \star\mathbf{do}\ G \rightarrow \star\vec{x}\colon [G \wedge INV', INV' \wedge \tau \leq L]\ \mathbf{od} \end{array}$$

*where* $INV' \stackrel{\text{def}}{=} INV \wedge D \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [D \ldots \tau])$.

*Proof* Our first step is to introduce the **idle** command at the end of Definition 12 (iteration). The introduction relies on $\neg\, G \wedge INV'$ being post-idle-stable.

$$\begin{array}{l} \star\vec{x}\colon [INV \wedge D = \tau \leq L, \neg\, G \wedge INV'] \\ \sqsupseteq \text{Law 9 (simple sequential composition); Definition of idle} \\ \quad \star\vec{x}\colon [INV \wedge D = \tau \leq L, \neg\, G \wedge INV']\ ;\ \mathbf{idle} \end{array}$$

We now proceed to refine the first component using the standard refinement calculus laws. To emphasise this we use the *s*-subscripted '$\sqsubseteq_s$' to stand for refinement in the standard calculus.

$$\begin{array}{l} \mathcal{M}_\rho\,(\star\vec{x}\colon [INV \wedge D = \tau \leq L, \neg\, G \wedge INV']) \\ \sqsupseteq_s \text{Definition 2 (specification)} \\ \quad \tau\colon [INV \wedge D = \tau \leq L \mathbin{@} \tau, \neg\, G \wedge INV' \mathbin{@} (\tau_0, \tau) \wedge \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\tau_0 \ldots \tau])] \\ \sqsubseteq_s \text{definition of } INV'\text{; strengthen postcondition; weaken precondition; choice} \\ \quad [\![\ d1, d2 : Time \mid d1 > 0 \wedge d2 > 0 \bullet \\ \qquad \tau\colon [INV' \wedge \tau \leq L + d2 \mathbin{@} \tau, \neg\, G \wedge INV' \wedge \tau \leq L + d2 \mathbin{@} \tau] \qquad \lhd \\ \sqsubseteq_s \text{Standard iteration with variant } \left\lfloor \frac{L+d2-\tau}{d1} \right\rfloor \\ \quad \mathbf{do}\ G \mathbin{@} \tau \rightarrow \\ \qquad \tau\colon \left[ G \wedge INV' \wedge \tau \leq L + d2 \mathbin{@} \tau, \begin{array}{c} INV' \wedge \tau \leq L + d2 \mathbin{@} \tau\ \wedge \\ \left\lfloor \frac{L+d2-\tau}{d1} \right\rfloor < \left\lfloor \frac{L+d2-\tau_0}{d1} \right\rfloor \end{array} \right] \\ \quad \mathbf{od} \\ \sqsubseteq_s \text{definition of } INV'\text{; weaken precondition; strengthen with } \tau_0 \leq \tau \\ \quad \mathbf{do}\ G \mathbin{@} \tau \rightarrow \\ \qquad \tau\colon \left[ G \wedge INV' \mathbin{@} \tau, \begin{array}{l} INV' \wedge \tau_0 + d1 \leq \tau \leq L + d2 \mathbin{@} (\tau_0, \tau) \\ \tau_0 \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [\tau_0 \ldots \tau]) \end{array} \right] \\ \quad \mathbf{od} \\ \sqsupseteq_s \text{Definition 2 (specification)} \\ \quad \mathbf{do}\ G \mathbin{@} \tau \rightarrow \mathcal{M}_\rho\,(\star\vec{x}\colon [G \wedge INV', INV' \wedge \tau_0 + d1 \leq \tau \leq L + d2])\ \mathbf{od} \end{array}$$

We now concentrate on the body of the loop. The following step relies on the fact that $G \wedge INV'$ is idle-stable, and that $INV'$ is both pre- and post-idle-stable.

$$\star\vec{x}\colon [G \wedge INV', INV' \wedge \tau_0 + d1 \leq \tau \leq L + d2]$$

$\sqsubseteq$ Law 9 (simple sequential composition) $\times$ 2; Definitions of idles
$\mathbf{idle}_{\geq d1};\ \star\vec{x}\colon \left[G \wedge INV', INV' \wedge \tau \leq L\right];\ \mathbf{idle}_{\leq d2}$

Combining the above together we get,

$\| \ d1, d2 : Time \mid d1 > 0 \wedge d2 > 0 \bullet$
$\quad \mathbf{do}\ G @ \tau \rightarrow \mathcal{M}_\rho \left(\mathbf{idle}_{\geq d1};\ \star\vec{x}\colon \left[G \wedge INV', INV' \wedge \tau \leq L\right];\ \mathbf{idle}_{\leq d2}\right)$
$\quad \mathbf{od}\ \circ\mathcal{M}_\rho\,(\mathbf{idle})$
$\sqsubseteq_s$ Definition 12 (iteration)
$\mathcal{M}_\rho\left(\star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV', INV' \wedge \tau \leq L\right]\ \mathbf{od}\right)$

□

A simpler rule for iteration does not involve all timing aspects.

**Law 1 (iteration with deadline)** *Given an idle-stable invariant property, INV, a deadline expression L, which is frame-stable with respect to the frame $\vec{x}$, and an idle-stable expression G, where none of G, L and INV contain references to $\tau_0$ or zero-subscripted variables,*

$$\star\vec{x}\colon \left[INV \wedge \tau \leq L, \neg\, G \wedge INV\right] \sqsubseteq \star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV, INV \wedge \tau \leq L\right]\ \mathbf{od}$$

*Proof* We let $INV' \overset{\text{def}}{=} INV \wedge D \leq \tau \wedge stable(\hat{\rho} \setminus \vec{x}, [D \ldots \tau])$, where $D$ is fresh, and make use of Law 13 (iteration timing with deadline).

$\star\vec{x}\colon \left[INV \wedge \tau \leq L, \neg\, G \wedge INV\right]$
$\sqsubseteq$ Law 8 (logical constant); Law 6 (separate assumption); Law 4 (strengthen effect)
$|[\ \mathbf{con}\, D : Time \bullet \star\vec{x}\colon \left[INV \wedge D = \tau \leq L, \neg\, G \wedge INV'\right]\ ]|$
$\sqsubseteq$ Law 13 (iteration timing with deadline)
$|[\ \mathbf{con}\, D : Time \bullet \star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV', INV' \wedge \tau \leq L\right]\ \mathbf{od}\ ]|$
$\sqsubseteq$ Equivalent effect by Law 4 (strengthen effect); Law 3 (weaken assumption)
$|[\ \mathbf{con}\, D : Time \bullet \star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV, INV \wedge \tau \leq L\right]\ \mathbf{od}\ ]|$
$\sqsubseteq$ Law 8 (logical constant)
$\star\mathbf{do}\ G \rightarrow \star\vec{x}\colon \left[G \wedge INV, INV \wedge \tau \leq L\right]\ \mathbf{od}$

□

## BIOGRAPHIES

**Ian Hayes** received his PhD from the University of New South Wales, Australia in 1983. He has been on the academic staff of the University of Queensland since 1985, and is an associated academic of the Software Verification Research Centre there.

**Mark Utting** received his PhD from the University of New South Wales in 1992. He then worked at the Software Verification Research Centre in Brisbane, developing the Ergo theorem prover and this real-time refinement calculus. In late 1996 he became a lecturer at Waikato University, and is now enjoying teaching programming languages and formal methods there. He is married to Lynda and they have four young children and a cat.