SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 98-14

Separating Timing and Calculation in Real-Time Refinement

Ian Hayes

July 1998

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

Separating Timing and Calculation in Real-Time Refinement

Ian Hayes

Abstract.

We consider the specification and refinement of sequential real-time programs. Our real-time specifications describe the allowable behaviours of an implementation in terms of the values of variables over time. Hence within a specification the values of the variables and the times at which they have those values are intertwined. However, in a real-time program some commands are concerned with calculating the right outputs, while other commands, such as delays and deadlines, are concerned with making sure the outputs appear at the right time.

During the refinement process we would like to decompose the overall problem into those aspects dealing with time and those that are purely calculation. We need refinement rules that allow us to separate these concerns. Further, given a component that is only concerned with calculation, the complexities of the real-time calculus that deal with timing behaviour are an unnecessary burden. Such calculational components can be developed more straightforwardly in the standard refinement calculus. We would like to allow the use of the untimed calculus for the development of such components. To do that we need to embed the untimed calculus within the real-time calculus.

1 Introduction

We consider the problem of developing a sequential real-time program via a process of stepwise refinement from a specification of the required real-time behaviour of a system. The program operates in an environment where it can sample *inputs*, perform calculations, and update *outputs*. Of course, the all important extra dimension to the problem is time. Not only must the program calculate the correct values to be output, it must also meet the specified timing requirements.

Following the approach used in the physical sciences for many centuries, we consider inputs and outputs to be traces over time (i.e., functions from time to their value at that time). A specification determines the allowable values of the outputs in terms of the values of the inputs. However, because inputs and outputs are both traces over time, a specification inherently includes both timing and value considerations rolled into one. As part of our refinement process, we would like to decompose the problem into those aspects that deal with real-time constraints and those that perform calculations. The aim of our approach is to be able to develop the components that perform calculations using essentially the standard (untimed) sequential refinement calculus. To do this, we need to be able to perform refinement steps that separate out timing constraints, leaving specifications of calculational components. To use standard refinement calculus derivations for the calculational components, we need to embed the standard calculus within the real-time calculus.

In Section 2 we provide an overview of the sequential real-time refinement calculus [3]. Section 3 introduces an example to illustrate the goals of the rest of the paper, and Sections 4 and 5 look at the relationship between the timed and untimed calculi, and the embedding of the untimed calculus within the real-time calculus.

2 Sequential real-time refinement

Environment In the real-time calculus, variables are modelled as functions over time. For example, the variable v of type T is modelled by a function from $Time$ to T

$$v : Time \rightarrow T$$

where $Time$ is represented by nonnegative reals. In the environment, ρ, of a program we distinguish between inputs, outputs and local variables. Inputs are under the control of the environment, not the program, although the program may make assumptions about the behaviour of inputs. Both outputs and local variables are under the control of the program; collectively we refer to outputs and local variables as the program variables and use the abbreviation $\hat{\rho}$ to stand for the program variables in an environment ρ. Outputs are distinguished from local variables because outputs are externally observable, and hence refinements must take care to preserve their behaviour.

The special variable τ is used to stand for the current time. It is not itself a function of time.

The target programming language consists of the guarded command language [1] extended with the following real-time commands:

- **delay until** D – delays execution until the absolute time D;
- x : **gettime** – assigns the current time to x;
- x : **read**(v) – samples the input v and assigns the value to x; and
- **deadline** D – specifies an absolute time deadline, D, to be met by the preceding code.

Of these the deadline command [2] is novel. It cannot be implemented by simply generating machine code (generating machine code only makes it harder to meet a deadline). Instead, a timing analysis phase is required to ensure that for every valid execution of the machine code, its deadlines will always be met. If the timing analysis is successful then the machine code is a valid implementation, otherwise it must be discarded.

Wide-spectrum language As with the standard refinement calculus, our language is extended with a specification command. However, in our case the specification is dealing with variables that are traces over time as well as the special variable, τ, that stands for the current time. The form of a specification command is $\star\tilde{x}\colon \big[R\big]$. The frame, $\tilde{x}$, is the vector of variables that may be modified by the command. Any program variables in the environment that are not in the frame remain unchanged (stable) for the duration of the execution of the command. We introduce the abbreviation

$$stable(y, S) == (\forall\, t1, t2 : S \bullet y(t1) = y(t2))$$

in order to state such stability properties. We also allow a vector of variables to be used in place of y, in which case all of the variables in the vector are stable over S.

The postcondition, R, of a specification command is a predicate specifying the effect that the command is to achieve. The real-time specification command has been defined by Utting and Fidge [7] by encoding it as a standard specification command that changes only time:

$$\star\tilde{x}\colon \big[R\big] == \tau\colon \big[R \wedge {}^{\backprime}\tau \leq \tau' \wedge stable(\hat{\rho} \setminus \tilde{x}, [{}^{\backprime}\tau \ldots \tau'])\big] \tag{1}$$

where ${}^{\backprime}\tau$ and τ' stand for the start and finish times of the command (equivalent to Morgan's τ_0 and τ convention [6]), and $\hat{\rho} \setminus \tilde{x}$ is the set of program variables in the environment ρ minus the frame $\tilde{x}$. As well as establishing R, the specification also ensures that time does not go backward, and that the program variables that are not in the frame are stable for the duration of the command.

Stable predicates Assertions within real-time specifications may refer to inputs and to the current time. Because of this, one cannot guarantee that just because an assertion, P, holds at time t, that P will hold at any later time, even if the program has not modified any of the variables occurring in P in the intervening period. For example, the assertion $\tau \leq 1$ holds at time 1 but does not hold at any later time. On the other hand, if the assertion $1 \leq \tau$ holds at some time t, it holds for all later times. If v is an input then, if the assertion $v(\tau) = 0$ holds at time 1, that does not guarantee that it holds at later times, because the value of the input may change independently of the action (or lack thereof) of the program.

In dealing with real-time programs, assertions that are stable over time have special significance. For example, consider the following refinement, for introducing a selection command, that is valid in the standard refinement calculus

$$\begin{array}{ll} \star\big\{P\big\};\ \star x\colon \big[R\big] \sqsubseteq & \textbf{if}\, B \rightarrow \star\big\{P \wedge B\big\};\ \star x\colon \big[R\big] \\ & [\!]\ \neg\, B \rightarrow \star\big\{P \wedge \neg\, B\big\};\ \star x\colon \big[R\big] \\ & \textbf{fi} \end{array}$$

where B is some boolean expression. The above refinement is not necessarily valid in the timed calculus, because evaluating the guards takes time. Although P may be assumed to hold at the start time of the selection command, it may no longer hold at the beginning of a command within a branch because time has passed during the evaluation of the guards.

To avoid this problem, we insist that P is an *idle-stable* predicate, i.e., P does not change its value if only the current time and the value of inputs change. If we assume that, within P, ${}^{\backprime}\tau$ is used to reference the current time, then P is idle-stable provided

$${}^{\backprime}\tau \leq \tau' \wedge stable(\hat{\rho}, [{}^{\backprime}\tau \ldots \tau']) \wedge P \Rightarrow P\,[{}^{\backprime}\tau \backslash \tau']$$

That is, if P holds at time ${}^{\backprime}\tau$, and all the program variables are stable from ${}^{\backprime}\tau$ until some later time, τ', then P still holds at τ'.

A further problem arises with the guards themselves. If the guards refer to inputs or to the current time, then their values are not necessarily stable. Although the guard B may evaluate to true, there is no guarantee that B will still hold at the time the corresponding branch commences execution. To avoid this problem, we insist that guards do not refer to inputs or to the current time, and hence they are idle-stable.

Similarly, just because both branches of the selection establish R, it does not follow that the whole selection will establish R. There are two problems here:

- a branch may have established R, but it takes time for the selection command to exit after completing the branch, during which time R may be invalidated;
- if R refers to initial values of variables, then within the specification command on the left side of the refinement the initial values refer to the values at the start time of the whole selection command, whereas within a branch the initial values refer to the values of the variables when the body of the branch commenced. References to either time or inputs may have changed.

To avoid these problems, we require that such predicates are both *pre-idle-stable* and *post-idle-stable*. Within an effect predicate, R, $`\tau$ is used to refer to the start time of the command, and τ' is used to refer to the finish time of the command. R is pre-idle-stable provided

$$\forall u : \mathit{Time} \bullet u \leq {`\tau} \leq \tau' \land \mathit{stable}(\hat{\rho}, [u \ldots {`\tau}]) \land R \Rightarrow R\,[{`\tau}\backslash u]$$

That is, if the program variables are stable from some time u (representing the start time of the selection command in our example) until $`\tau$, and $R$ holds with respect to start time $`\tau$ and finish time τ', then R also holds with the start time replaced by u.

R is post-idle-stable provided

$$\forall u : \mathit{Time} \bullet {`\tau} \leq \tau' \leq u \land R \land \mathit{stable}(\hat{\rho}, [\tau' \ldots u]) \Rightarrow R\,[\tau'\backslash u]$$

That is, if R holds for start time $`\tau$ and finish time τ', and the program variables are stable from τ' until a later time u (representing the finish time of the whole selection command), then R also holds with the finish time replaced by u.

In the selection command example, in the time periods taken to evaluate the guards and exit the selection, the program variables are stable. If the effect R is both pre and post-idle-stable, then provided R is achieved for each of the branches, it will be achieved for the whole selection command.

3 An example

Consider the example of trying to determine the size of an object passing through a sensor beam on a conveyor belt moving at constant speed. (This example is taken from [4] where a more complete refinement may be found.) The sensor signal rises when the object interrupts the beam and falls when the object has passed through the beam. The size of the object is $(f - r) * s$, where r and f are the times at which the sensor rises and falls, and s is the speed of the belt.

con $r, f : \mathit{Time}$;
const $s == 10\,\mathrm{m/s}$ – speed in metres per second

The logical constants, r and f, denote the exact times at which the sensor rises and falls. We cannot use these directly within the final program. Instead we need to determine approximations, rt and ft, to r and f, and then calculate the approximate size of the object in terms of rt and ft.

$$\begin{array}{l}\mathbf{var}\, size : \mathbb{N}\,\mu\, m; \ \ \text{– size in micrometres} \\ \mathbf{var}\, rt, ft : \mathbb{N}\,\mu\, s \text{ – times in microseconds}\end{array}$$

Although these variables have been declared to be of type natural number (with units), recall that in the underlying model they are viewed as functions from *Time* to natural numbers, and hence to determine the value of a variable at a particular time, one must index the variable by that time.

Let us concentrate on the calculation component of the refinement, assuming the approximations to r and f have been determined to within an error bound of $100\,\mu\, s$:

$$\begin{array}{l}\mathbf{const}\ e == 100\,\mu\,\mathrm{s}; \\ \star\big\{ rt(\grave{}\tau) \in [r \ldots r+e] \wedge ft(\grave{}\tau) \in [f \ldots f+e]\big\}\end{array} \qquad (2)$$

An assertion states a condition that is true at the point at which it appears in a program. An assertion may refer to the current time, but as the assertion takes no time, both the start time, $\grave{}\tau$, and the finish time, τ', of an assertion are the same. To allow for simpler expression of laws later, we follow the convention of only ever using $\grave{}\tau$ to refer to the current time within an assertion.

The task is to calculate the size of the object to within 1 mm, and to complete the calculation within $100\,\mu\, s$ of the object having passed:

$$\star size\colon \big[size(\tau') \in (f-r) * s \pm 1\,\mathrm{mm} \wedge \tau' \leq f + 100\,\mu\, s\big] \qquad (3)$$

To satisfy this specification, one must both calculate the size and meet the deadline. Here we can make use of the deadline command to separate calculation and timing requirements. This is a crucial step in the process as it allows one to concentrate on the calculation of the size without having to worry explicitly about the timing constraint. We make use of the *separate deadline* law [3].

Law 1 (separate deadline) *Given an environment, ρ, provided D is a time-valued expression, which may only include references to logical constants and program variables but no references to $\grave{}\tau$*

$$\star x\colon \big[P,\quad R \wedge \tau' \leq (D \mathbin{@} \tau')\big] \sqsubseteq \star x\colon \big[P,\quad R\big]\,;\ \mathbf{deadline}\, D$$

The expression D may contain references to program variables. These are to be interpreted as the value of the variables at the finish time of the command. The notation $D \mathbin{@} \tau'$ stands for the expression D with any reference to a program variable, v, replaced by $v(\tau')$. Refining (3) using this law gives

$$\begin{array}{l}\star size\colon \big[size(\tau') \in (f-r) * s \pm 1\,\mathrm{mm}\big]\,; \\ \mathbf{deadline}\, f + 100\,\mu\, s\end{array} \qquad (4)$$

The specification command can now be refined to an assignment command using the assignment introduction rule of the real-time calculus.

Law 2 (assignment) *Given an environment, ρ, a frame, $\tilde{x}$, such that $\tilde{x} \subseteq \hat{\rho}$, and a vector of idle-stable expressions, $\tilde{E}$, provided*

$$\grave{}\tau \leq \tau' \wedge stable(\hat{\rho} \setminus \tilde{x}, [\grave{}\tau \ldots \tau']) \wedge P \wedge \tilde{x}(\tau') = (\tilde{E} \mathbin{@} \grave{}\tau) \Rightarrow R$$

for all states, then

$$\star\{P\};\ \star\tilde{x}\colon [R] \sqsubseteq \tilde{x} := \tilde{E}.$$

The specification command (4) can be refined using this law to the assignment

$$size := (ft - rt) * s$$

with the following proof obligation

$$\begin{pmatrix} \grave{}\tau \leq \tau' \wedge \\ stable(rt, [\grave{}\tau \ldots \tau']) \wedge \\ stable(ft, [\grave{}\tau \ldots \tau']) \wedge \\ rt(\grave{}\tau) \in [r \ldots r + e] \wedge \\ ft(\grave{}\tau) \in [f \ldots f + e] \wedge \\ size(\tau') = (ft(\grave{}\tau) - rt(\grave{}\tau)) * s \end{pmatrix} \Rightarrow size(\tau') \in (f - r) * s \pm 1\,\text{mm} \tag{5}$$

If we consider the refinement of the specification command (4) in the standard untimed refinement calculus, the assumptions (2) can be written

$$\star\{\grave{}rt \in [r \ldots r + e] \wedge \grave{}ft \in [f \ldots f + e]\} \tag{6}$$

where $\grave{}rt$ stands for the value of the variable at the time at which the assertion is reached. We use the before-state decoration to simplify writing laws involving assertions as preconditions. The specification (4) can be written as

$$\star size\colon [size' \in (f - r) * s \pm 1\,\text{mm}] \tag{7}$$

where $size'$ refers to the value of the variable $size$ on termination of the command. This can be refined to the same assignment as before but with the following proof obligation:

$$\begin{pmatrix} \grave{}rt \in [r \ldots r + e] \wedge \\ \grave{}ft \in [f \ldots f + e] \end{pmatrix} \Rightarrow (\grave{}ft - \grave{}rt) * s \in (f - r) * s \pm 1\,\text{mm} \tag{8}$$

In this case the standard refinement contains all the essential information of the refinement in the timed calculus, but omits the complexities of explicit time indices and stability conditions. More importantly the proof obligation is considerably simpler. The goal of this paper is to show how this simpler approach for calculational components can be embedded within the real-time refinement calculus.

Notational conventions We start by introducing a notational convention that allows us to write the simpler form of specification within the real-time calculus. Within assertions and specifications, we allow a variable, v, to be referenced in one of three ways:

- v refers to the whole trace of v (it may be explicitly indexed to get the value of v at a particular time);
- $\grave{}v$ refers to the value of v at the current time within an assertion, and within a specification to the value of v at the start time of the specification; and
- v', which is only meaningful in a specification, refers to the value of v at the finish of the command.

Using this convention within assertions and specifications, $\grave{}v$ is equivalent to $v(\grave{}\tau)$, and within a specification v' is equivalent to $v(\tau')$. Hence the assertions (2) and (6) are equivalent, and the specifications (4) and (7) are equivalent.

Our choice of notation is motivated by wanting v' and $\grave{}v$ to have the same meaning as v and v_0 in Morgan's refinement calculus [6], while allowing references to the value of v at times other than the start and finish times of a specification using the trace notation v. Our choice of notation for this paper is based on the desire to have an identifier v stand for its whole timed trace, rather than its value at some particular time. Then to avoid ambiguities, we need separate notations for the before and after-state values of variables; the notation chosen is derived from that of Hehner [5]. (Aside: In earlier papers we have used the notation v_0 to stand for the initial value of v, and v to stand for both the whole trace and the final value of v. Unfortunately, expressions like $x = y$, where x and y are variables, are then ambiguous: the x and y could either both be whole traces or final state values, and the context cannot disambiguate the two interpretations.)

For example, the read command samples its input at some time during its execution. It can be defined in terms of a specification command as follows:

$$x : \mathbf{read}(v) == \star x\colon \left[x \in v(\!|\ [\grave{}\tau \ldots \tau']\ |\!)\right]$$

where $v(\!|\ [\grave{}\tau \ldots \tau']\ |\!)$ is the set of values of v over the closed interval from the start of the command, $\grave{}\tau$, to the finish of the command, τ'. Using just the $\grave{}v$ and v' conventions one cannot specify the read command because one can only refer to the values of v at $\grave{}\tau$ and τ', and not over the range in between.

Our notational conventions allow us to abbreviate assertions and specifications as illustrated above. However, they do not justify the abbreviation of the proof obligation (5) to (8). If we expand (8) using our notational conventions we get

$$\begin{pmatrix} rt(\grave{}\tau) \in [r \ldots r+e] \wedge \\ ft(\grave{}\tau) \in [f \ldots f+e] \end{pmatrix} \Rightarrow (ft(\grave{}\tau) - rt(\grave{}\tau)) * s \in (f-r) * s \pm 1\,\text{mm} \tag{9}$$

which is simpler than (5). For such calculational components, the additional premisses in (5) are not needed to prove the obligation.

4 Relating the timed and untimed calculi

In order to support the use of the simpler untimed proof obligations for refinement steps for calculational components, we need to examine the relationship between the untimed and timed refinement calculi. In this section, we examine an example refinement law and its proof obligation, and show how the proof obligation for the real-time law can be transformed into a proof obligation in the untimed model.

Consider the following refinement in the real-time calculus

$$\star\{P\};\ \star\tilde{x}\colon[Q] \sqsubseteq \star\{P\};\ \star\tilde{x}\colon[R] \tag{10}$$

where P, Q and R are predicates in which there is no use of the $`v$ and $v'$ notation, except for $`\tau$ and τ', i.e., all references to a variable, v, treat it explicitly as a function of time. In addition, P does not refer to τ', only $`\tau$. The refinement (10) is valid provided that

$$`\tau \leq \tau' \land \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [`\tau \ldots \tau']) \land P \land R \Rightarrow Q \tag{11}$$

holds for all timed states. This is essentially just the strengthen postcondition rule of Morgan [6] adapted to the slightly different notation, and with the additional premisses $`\tau \leq \tau'$ and $\mathit{stable}(\hat{\rho} \setminus \tilde{x}, [`\tau \ldots \tau'])$ derived from the definition of the specification command (1).

We would like to make use of the $`v$ and $v'$ notation to stand for $v(`\tau)$ and $v(\tau')$, respectively. We can always eliminate the abbreviations by replacing $`v$ by $v(`\tau)$ and v' by $v(\tau')$. We define the notation $R\ @\ (`\tau, \tau')$ to be $R$ with all references to $`v$ replaced by $v(`\tau)$ and all references to v' replaced by $v(\tau')$, for all variables, v, occurring in R.

If the predicates P, Q and R in the above refinement rule are allowed to use the abbreviated notation (noting that P is restricted to use only $`v$ and not v'), then (11) has to be rewritten

$$`\tau \leq \tau' \land \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [`\tau \ldots \tau']) \Rightarrow (P \land R \Rightarrow Q)\ @\ (`\tau, \tau') \tag{12}$$

Now consider the special case of a calculational component in which P, Q and R make no explicit references to the trace variable v, i.e., they only reference v via the $`v$ and $v'$ conventions. Further, $P$, $Q$ and $R$ make no references to $`\tau$ and τ'. For this case P is essentially a predicate constraining the pre-state values of variables, and Q and R are predicates relating the pre and post-state values of variables.

The stability predicate implies that for every program variable, v, that is not in the frame ($v \in \hat{\rho} \setminus \tilde{x}$), its value does not change and therefore we may deduce that $`v = v'$. This allows one to replace any occurrences of $v'$ with $`v$, and hence to rewrite (12) as

$$\begin{pmatrix} `\tau \leq \tau' \land \\ \mathit{stable}(\hat{\rho} \setminus \tilde{x}, [`\tau \ldots \tau']) \end{pmatrix} \Rightarrow (P \land R \Rightarrow Q)\, [v' \backslash `v]_{v \in \hat{\rho} \setminus \tilde{x}}\ @\ (`\tau, \tau') \tag{13}$$

where the notation $[v' \backslash `v]_{v \in \hat{\rho} \setminus \tilde{x}}$ stands for the replacement of all occurrences of $v'$ by $`v$, for all identifiers, v, in the set $\hat{\rho} \setminus \tilde{x}$. Now we note that the time progress and stability premisses (to the left of the first implication) in (13) are of no help in establishing the right side of (13) because the predicate makes no references to $`\tau$ and $\tau'$, or to the values of variables other than via $`v$ and v'. Hence we can simplify (13) to

$$(P \land R \Rightarrow Q)\, [v' \backslash `v]_{v \in \hat{\rho} \setminus \tilde{x}}\ @\ (`\tau, \tau') \tag{14}$$

If (14) holds for all values of the timed traces of its variables, then the refinement (10) given above is valid. But note that the predicate to the left of the '@' is essentially that used in the untimed refinement calculus for the strengthen postcondition law (aside from the different notational conventions). If the predicate to the left of the '@' holds for all values of $`v$ and v', then (14) will hold for all timed traces. Hence all we need to show is that for all values of the before and after states

$$(P \land R \Rightarrow Q)\, [v' \backslash `v]_{v \in \hat{\rho} \setminus \tilde{x}} \tag{15}$$

In the next section, we formalise the above reasoning by showing that for any predicate M that does not refer to $\grave{\tau}$ or τ' and only references variables via the $\grave{v}$ and v' convention, that if M holds for all untimed states, then $M \mathbin{@} (\grave{\tau}, \tau')$ holds for all timed trace states.

5 Embedding the untimed calculus

In this section we formalise the relationship between the timed and untimed models. In the timed model, variables are represented by traces over time, while in the untimed model only the before and after-state values of variables are available. The before-state value of a variable, $\grave{v}$, is the value of v at time $\grave{\tau}$ and the after-state value, v', is the value of v at time τ'. Below we formalise timed and untimed states and the relationship between them, and then we show the relationship between untimed and timed proof obligations.

In a specification command, the effect predicate may refer to logical constants, program constants, variables (inputs, outputs, and local variables), and the start and finish time of the command ($\grave{\tau}$ and τ'). In both the timed and untimed models, constants are treated in the same manner, and hence we shall ignore them in this exposition. We represent environments by the following schema

$$
\begin{array}{|l}
\textit{Environment} \\
\hline
\textit{in}, \textit{out}, \textit{lvar}, \textit{pvar}, \textit{vars} : \mathbb{P}\,\textit{Ident} \\
\textit{typeof} : \textit{Ident} \mapsto \mathbb{P}\,\textit{Val} \\
\hline
\textit{pairwise\_disjoint}\langle \textit{in}, \textit{out}, \textit{lvar} \rangle \\
\textit{pvar} = \textit{out} \cup \textit{lvar} \\
\textit{vars} = \textit{in} \cup \textit{out} \cup \textit{lvar} \land \text{`}\tau\text{'} \notin \textit{vars} \\
\operatorname{dom} \textit{typeof} = \textit{vars} \\
\hline
\end{array}
$$

where *Ident* is the set of all identifiers (including ‘τ’) and *Val* is the set of all values (including booleans, integers, *Time*, etc.). The disjoint sets *in*, *out*, and *lvar* represent the names of the inputs, outputs and local variables, respectively. The set *pvar* gives the names of the program variables; for an environment, ρ, $\hat{\rho}$ is an abbreviation for $\rho.\textit{pvar}$. The set *vars* gives the names of all of the variables, and the function *typeof* gives their types.

In the timed model, variables are represented by functions of time, and the start and finish times are just time values. We can model a *timed state* by a mapping from identifiers to values, in which the identifiers ‘$\grave{\tau}$’ and ‘τ'’ are elements of the domain of a timed state and their values are of type time, and variables are modelled by timed traces. Given an environment, ρ, the set of all timed states is given by

$$
\begin{array}{l}
\textit{TState}_\rho == \{\gamma : \textit{Ident} \mapsto \textit{Value} \mid \\
\quad \{\text{`}\grave{\tau}\text{'}, \text{`}\tau'\text{'}\} \subseteq \operatorname{dom}\gamma \land \{\gamma(\text{`}\grave{\tau}\text{'}), \gamma(\text{`}\tau'\text{'})\} \subseteq \textit{Time} \land \\
\quad \rho.\textit{vars} \subseteq \operatorname{dom}\gamma \land (\forall\, \textit{id} : \rho.\textit{vars} \bullet \gamma(\textit{id}) \in (\textit{Time} \to \rho.\textit{typeof}(\textit{id})))\}
\end{array}
$$

where *Value* includes *Val* as well as timed traces of type $\textit{Time} \to \textit{Val}$. For example, the following timed state represents a start time of zero and a finish time of two, and a variable x that has value zero up to time one, and then has value one.

$$
\gamma_1 = \{\text{`}\grave{\tau}\text{'} \mapsto 0, \text{`}\tau'\text{'} \mapsto 2, \text{`}x\text{'} \mapsto (\lambda\, t : \textit{Time} \bullet \mathbf{if}\; t < 1 \;\;\mathbf{then}\;\; 0 \;\;\mathbf{else}\;\; 1)\}
$$

In the untimed model, we only have to model the values of variables in the before and after states. Given a plain identifier, id, we use $\grave{}id$ to refer before-state decorated identifier and id' to refer to the after-state decorated identifier. The sets of all before and after-state decorated identifiers are given by $\grave{}Ident$ and $Ident'$:

$$\begin{array}{l} \grave{}Ident == \{id : Ident \bullet \grave{}id\} \\ Ident' == \{id : Ident \bullet id'\} \end{array}$$

The set of all *untimed states* is given by

$$\begin{array}{l} UState_\rho == \{\sigma : (\grave{}Ident \cup Ident') \pfun Val \mid \\ \quad \{\text{`}\grave{}\tau\text{'}, \text{`}\tau'\text{'}\} \subseteq \mathrm{dom}\,\sigma \land \{\sigma(\text{`}\grave{}\tau\text{'}), \sigma(\text{`}\tau'\text{'})\} \subseteq Time \land \\ \quad (\forall id : \rho.vars \bullet \{\grave{}id, id'\} \subseteq \mathrm{dom}\,\sigma \land \{\sigma(\grave{}id), \sigma(id')\} \subseteq \rho.typeof(id))\} \end{array}$$

For example, the following represents an untimed state with a start time of zero and a finish time of two, and a variable, x, with an initial value of zero and a final value of one.

$$\sigma_1 = \{\text{`}\grave{}\tau\text{'} \mapsto 0, \text{`}\tau'\text{'} \mapsto 2, \text{`}\grave{}x\text{'} \mapsto 0, \text{`}x'\text{'} \mapsto 1\}$$

Given a timed state, γ, one can extract the corresponding untimed state: the before and after times, $\grave{}\tau$ and τ' are the same in both models, and the before and after values of each variable in the untimed model are just the values of the variables in the timed model at the before and after times.

$$\begin{array}{|l} extract == TState_\rho \to UState_\rho \\ \hline extract(\gamma) = \{\text{`}\grave{}\tau\text{'} \mapsto \gamma(\text{`}\grave{}\tau\text{'}), \text{`}\tau'\text{'} \mapsto \gamma(\text{`}\tau'\text{'})\} \cup \\ \quad \{id : \rho.vars \bullet \grave{}id \mapsto \gamma(id)(\gamma(\text{`}\grave{}\tau\text{'}))\} \cup \\ \quad \{id : \rho.vars \bullet id' \mapsto \gamma(id)(\gamma(\text{`}\tau'\text{'}))\} \end{array}$$

For example, if we apply *extract* to the example timed state γ_1 given above, then the result is equal to the example untimed state σ_1.

Given a predicate P that only references variables via the $\grave{}v$ and v' conventions we would like to show that if P holds for all states in an untimed interpretation, then $P \mathbin{@} (\grave{}\tau, \tau')$ holds for all states in the timed interpretation. To formalise this, we need to give both the untimed and timed semantics of predicates. We begin by giving part of the syntax for terms (including predicates).

$$\begin{array}{l} Term ::= constant \mid \grave{}\tau \mid \tau' \mid \grave{}id \mid id' \mid id \mid Term_0 \mathbin{@} (Term_1, Term_2) \mid \\ \quad \neg\ Term \mid Term_0 \land Term_1 \mid Term_0 = Term_1 \mid \ldots \end{array}$$

where *constant* stands for a constant, id is an identifier, and $Term_0$, $Term_1$ and $Term_2$ are terms.

Table 1 gives the untimed semantics for these constructs with the exception of variable traces and the '@' notation, neither of which make sense in the untimed interpretation. The untimed semantic function, M_U, given a term T and an untimed state σ, gives the value of the term in that state. For example,

Table 1. Untimed semantics of terms

$$
\begin{array}{|l}
M_U : \mathit{Term} \rightarrowtail (\mathit{UState}_\rho \rightarrow \mathit{Val}) \\
\hline
\forall\, \sigma : \mathit{UState}_\rho \bullet \\
M_U(\mathit{constant})(\sigma) = \mathit{constant} \\
M_U(\grave{}\tau)(\sigma) = \sigma(\text{`}\grave{}\tau\text{'}) \\
M_U(\tau')(\sigma) = \sigma(\text{`}\tau'\text{'}) \\
M_U(\grave{}\mathit{id})(\sigma) = \sigma(\grave{}\mathit{id}) \\
M_U(\mathit{id}')(\sigma) = \sigma(\mathit{id}') \\
M_U(\neg\ T)(\sigma) = \neg\ M_U(T)(\sigma) \\
M_U(T_0 \wedge T_1)(\sigma) = (M_U(T_0)(\sigma) \wedge M_U(T_1)(\sigma)) \\
M_U(T_0 = T_1)(\sigma) = (M_U(T_0)(\sigma) = M_U(T_1)(\sigma))
\end{array}
$$

where T_0 and T_1 are terms.

$$
\begin{aligned}
& M_U(\grave{}\tau = \tau' \wedge \neg\ (\grave{}x = x'))(\sigma) \\
={}& M_U(\grave{}\tau = \tau')(\sigma) \wedge M_U(\neg\ (\grave{}x = x'))(\sigma) \\
={}& (M_U(\grave{}\tau)(\sigma) = M_U(\tau')(\sigma)) \wedge \neg\ M_U(\grave{}x = x')(\sigma) \\
={}& \sigma(\text{`}\grave{}\tau\text{'}) = \sigma(\text{`}\tau'\text{'}) \wedge \neg\ (M_U(\grave{}x)(\sigma) = M_U(x')(\sigma)) \\
={}& \sigma(\text{`}\grave{}\tau\text{'}) = \sigma(\text{`}\tau'\text{'}) \wedge \neg\ (\sigma(\text{`}\grave{}x\text{'}) = \sigma(\text{`}x'\text{'}))
\end{aligned}
$$

Table 2 gives the semantics of terms in the timed model. The semantic function, M_T, given a term and a timed state, gives the value of the term in that state. In this case all constructs can be given a meaning. An (undecorated) identifier, *id*, stands for its whole timed trace value, $\gamma(\mathit{id})$, and before and after decorated identifiers stand for the value of the trace, $\gamma(\mathit{id})$, at the start and finish times, $\gamma(\text{`}\grave{}\tau\text{'})$ and $\gamma(\text{`}\grave{}\tau\text{'})$.

Table 2. Timed semantics of terms

$$
\begin{array}{|l}
M_T : \mathit{Term} \rightarrowtail (\mathit{TState}_\rho \rightarrow \mathit{Value}) \\
\hline
\forall\, \gamma : \mathit{TState}_\rho \bullet \\
M_T(\mathit{constant})(\gamma) = \mathit{constant} \\
M_T(\grave{}\tau)(\gamma) = \gamma(\text{`}\grave{}\tau\text{'}) \\
M_T(\tau')(\gamma) = \gamma(\text{`}\tau'\text{'}) \\
M_T(\mathit{id})(\gamma) = \gamma(\mathit{id}) \\
M_T(\grave{}\mathit{id})(\gamma) = \gamma(\mathit{id})(\gamma(\text{`}\grave{}\tau\text{'})) \\
M_T(\mathit{id}')(\gamma) = \gamma(\mathit{id})(\gamma(\text{`}\tau'\text{'})) \\
M_T(T_0\ @\ (T_1, T_2))(\gamma) = M_{TT}(T_0)(\gamma)(M_T(T_1)(\gamma), M_T(T_2)(\gamma)) \\
M_T(\neg\ T)(\gamma) = \neg\ M_T(T)(\gamma) \\
M_T(T_0 \wedge T_1)(\gamma) = (M_T(T_0)(\gamma) \wedge M_T(T_1)(\gamma)) \\
M_T(T_0 = T_1)(\gamma) = (M_T(T_0)(\gamma) = M_T(T_1)(\gamma))
\end{array}
$$

where T_0, T_1 and T_2 are terms.

In the definition of the ‘@’ notation, we allow terms to be used to specify the start and finish times. Hence, within the term to the left of the ‘@’, the start and finish times are interpreted as the values of these two terms. To define this, we introduce in Table 3 the auxiliary semantic function M_{TT} which takes an additional pair of parameters specifying the start and finish times. (The definition of M_{TT} for the ‘@’ operator is given here for completeness, but it corresponds to the case where an ‘@’ operator is used within the left operand of another ‘@’ operator and is not used in the remainder of this paper.)

Table 3. Auxiliary function for timed terms

$$
\begin{array}{|l}
M_{TT} : \mathit{Term} \twoheadrightarrow (\mathit{TState}_\rho \rightarrow (\mathit{Time} \times \mathit{Time} \rightarrow \mathit{Value})) \\
\hline
\forall\, \gamma : \mathit{TState}_\rho;\ t_0, t_1 : \mathit{Time} \bullet \\
M_{TT}(\mathit{constant})(\gamma)(t_0, t_1) = \mathit{constant} \\
M_{TT}(\backprime\tau)(\gamma)(t_0, t_1) = t_0 \\
M_{TT}(\tau')(\gamma)(t_0, t_1) = t_1 \\
M_{TT}(\mathit{id})(\gamma)(t_0, t_1) = \gamma(\mathit{id}) \\
M_{TT}(\backprime\mathit{id})(\gamma)(t_0, t_1) = \gamma(\mathit{id})(t_0) \\
M_{TT}(\mathit{id}')(\gamma)(t_0, t_1) = \gamma(\mathit{id})(t_1) \\
M_{TT}(T_0 \mathbin{@} (T_1, T_2))(\gamma)(t_0, t_1) = \\
\qquad M_{TT}(T_0)(\gamma)(M_{TT}(T_1)(\gamma)(t_0, t_1), M_{TT}(T_2)(\gamma)(t_0, t_1)) \\
M_{TT}(\neg\, T)(\gamma)(t_0, t_1) = \neg\, M_{TT}(T)(\gamma)(t_0, t_1) \\
M_{TT}(T_0 \land T_1)(\gamma)(t_0, t_1) = (M_{TT}(T_0)(\gamma)(t_0, t_1) \land M_{TT}(T_1)(\gamma)(t_0, t_1)) \\
M_{TT}(T_0 = T_1)(\gamma)(t_0, t_1) = (M_{TT}(T_0)(\gamma)(t_0, t_1) = M_{TT}(T_1)(\gamma)(t_0, t_1))
\end{array}
$$

where T_0, T_1 and T_2 are terms.

As an example of the application of the timed semantics, we consider the term used earlier, $\backprime\tau = \tau' \land \neg\,(\backprime x = x')$, but with an application of an ‘@’ operator added.

$$
\begin{array}{l}
\quad M_T((\backprime\tau = \tau' \land \neg\,(\backprime x = x')) \mathbin{@} (\backprime\tau, \tau'))(\gamma) \\
= M_{TT}(\backprime\tau = \tau' \land \neg\,(\backprime x = x'))(\gamma)(M_T(\backprime\tau)(\gamma), M_T(\tau')(\gamma)) \\
= M_{TT}(\backprime\tau = \tau' \land \neg\,(\backprime x = x'))(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \\
= M_{TT}(\backprime\tau = \tau')(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \land M_{TT}(\neg\,(\backprime x = x'))(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \\
= (M_{TT}(\backprime\tau)(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) = M_{TT}(\tau')(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \land \\
\qquad \neg\, M_{TT}(\backprime x = x')(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \\
= (\gamma(\text{‘}\backprime\tau\text{’}) = \gamma(\text{‘}\tau'\text{’})) \land \neg\,(M_{TT}(\backprime x)(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) = M_{TT}(x')(\gamma)(\gamma(\text{‘}\backprime\tau\text{’}), \gamma(\text{‘}\tau'\text{’})) \\
= (\gamma(\text{‘}\backprime\tau\text{’}) = \gamma(\text{‘}\tau'\text{’})) \land \neg\,(\gamma(\text{‘}x\text{’})(\gamma(\text{‘}\backprime\tau\text{’})) = \gamma(\text{‘}x\text{’})(\gamma(\text{‘}\tau'\text{’}))) \\
= \mathit{false}
\end{array}
$$

Note that we can deduce in the timed model that this term is false, independently of the state γ, because if the start and finish times are the same then the values of a variable at the start and finish times must be the same. This was not possible for the same example in the untimed model for an arbitrary state, σ. However, if we assume that the untimed state, σ, in the untimed interpretation is the result of extracting an untimed state from some timed state,

γ, then we can also deduce the term is false. Let $\sigma = \mathit{extract}(\gamma)$, then

$$\begin{aligned}
&\sigma(\text{`}{}^{\backprime}\tau\text{'}) = \sigma(\text{`}\tau'\text{'}) \land \neg\,(\sigma(\text{`}{}^{\backprime}x\text{'}) = \sigma(\text{`}x'\text{'}))\\
=\;& \mathit{extract}(\gamma)(\text{`}{}^{\backprime}\tau\text{'}) = \mathit{extract}(\gamma)(\text{`}\tau'\text{'}) \land \neg\,(\mathit{extract}(\gamma)(\text{`}{}^{\backprime}x\text{'}) = \mathit{extract}(\gamma)(\text{`}x'\text{'}))\\
=\;& \gamma(\text{`}{}^{\backprime}\tau\text{'}) = \gamma(\text{`}\tau'\text{'}) \land \neg\,(\gamma(\text{`}x\text{'})(\gamma(\text{`}{}^{\backprime}\tau\text{'})) = \gamma(\text{`}x\text{'})(\gamma(\text{`}\tau'\text{'})))\\
=\;& \mathit{false}
\end{aligned}$$

Note that the second last line is the same as that in the timed semantics case. This example is a special case of the general theorem that we would like to prove.

Theorem 1. *For any predicate P that does not use the '@' operator, and only references variables via the ${}^{\backprime}v$ and v' conventions, if P holds for all states in an untimed interpretation, then $P \mathbin{@} ({}^{\backprime}\tau, \tau')$ holds for all states in the timed interpretation:*

$$(\forall\, \sigma : \mathit{UState}_\rho \bullet M_U(P)(\sigma)) \Rightarrow (\forall\, \gamma : \mathit{TState}_\rho \bullet M_T(P \mathbin{@} ({}^{\backprime}\tau, \tau'))(\gamma))$$

Proof. The proof relies on Lemma 1 below.

$$\begin{aligned}
&\forall\, \gamma : \mathit{TState}_\rho \bullet M_T(P \mathbin{@} ({}^{\backprime}\tau, \tau'))(\gamma)\\
\equiv\;& \text{Lemma 1}\\
&\forall\, \gamma : \mathit{TState}_\rho \bullet M_U(P)(\mathit{extract}(\gamma))\\
\Leftarrow\;& \text{change of variable; } \mathit{range}(\mathit{extract}) \subseteq \mathit{UState}_\rho\\
&\forall\, \sigma : \mathit{UState}_\rho \bullet M_U(P)(\sigma)
\end{aligned}$$

Lemma 1. *Given any term, P, satisfying the same restrictions as for Theorem 1, the timed meaning of a term $P \mathbin{@} ({}^{\backprime}\tau, \tau')$ in a timed state γ, is equal to the untimed meaning of P in the untimed state extracted from γ.*

$$\forall\, \gamma : \mathit{TState}_\rho \bullet M_T(P \mathbin{@} ({}^{\backprime}\tau, \tau'))(\gamma) = M_U(P)(\mathit{extract}(\gamma))$$

Proof. The proof makes use of the semantics of terms in both the timed and untimed interpretations, and uses structural induction over terms.

$$\begin{aligned}
&M_T(P \mathbin{@} ({}^{\backprime}\tau, \tau'))(\gamma)\\
=\;& \text{timed semantics of `@'}\\
&M_{TT}(P)(\gamma)(M_T({}^{\backprime}\tau)(\gamma), M_T(\tau')(\gamma))\\
=\;& \text{timed semantics of } {}^{\backprime}\tau \text{ and } \tau'\\
&M_{TT}(P)(\gamma)(\gamma(\text{`}{}^{\backprime}\tau\text{'}), \gamma(\text{`}\tau'\text{'}))\\
=\;& \text{structural induction}\\
&M_U(P)(\mathit{extract}(\gamma))
\end{aligned}$$

The last step is proved by structural induction over terms. Selected base cases follow:
Constants:

$$\begin{aligned}
&M_{TT}(\mathit{constant})(\gamma)(\gamma(\text{`}{}^{\backprime}\tau\text{'}), \gamma(\text{`}\tau'\text{'}))\\
=\;& \text{timed semantics}\\
&\mathit{constant}\\
=\;& \text{untimed semantics}\\
&M_U(\mathit{constant})(\mathit{extract}(\gamma))
\end{aligned}$$

Start time:

$M_{TT}(`\tau)(\gamma)(\gamma(‘`\tau’), \gamma(‘\tau'’))$
$=$ timed semantics
$\gamma(‘`\tau’)$
$=$ definition of extract
$\mathit{extract}(\gamma)(‘`\tau’)$
$=$ untimed semantics
$M_U(`\tau)(\mathit{extract}(\gamma))$

Before variable value:

$M_{TT}(`\mathit{id})(\gamma)(\gamma(‘`\tau’), \gamma(‘\tau'’))$
$=$ timed semantics
$\gamma(\mathit{id})(\gamma(‘`\tau’))$
$=$ definition of extract
$\mathit{extract}(\gamma)(`\mathit{id})$
$=$ untimed semantics
$M_U(`\mathit{id})(\mathit{extract}(\gamma))$

The inductive cases are straightforward. For example:

$M_{TT}(T_0 \wedge T_1)(\gamma)(\gamma(‘`\tau’), \gamma(‘\tau'’))$
$=$ timed semantics
$M_{TT}(T_0)(\gamma)(\gamma(‘`\tau’), \gamma(‘\tau'’)) \wedge M_{TT}(T_1)(\gamma)(\gamma(‘`\tau’), \gamma(‘\tau'’))$
$=$ inductive hypothesis
$M_U(T_0)(\mathit{extract}(\gamma)) \wedge M_U(T_1)(\mathit{extract}(\gamma))$
$=$ untimed semantics
$M_U(T_0 \wedge T_1)(\mathit{extract}(\gamma))$

6 Conclusions

The sequential real-time refinement calculus [2] introduced a novel deadline command as a way of including timing deadlines within machine-independent, higher-level, real-time programs. In addition, it introduced laws, such as *separate deadline*, that allow the timing related parts of a specification to be separated from the calculational parts. However refinements of the calculational components in the raw real-time calculus are encumbered by the additional baggage required for the real-time components. The objective of this paper has been to show that the untimed calculus can be embedded within the real-time calculus via the use of some special notational conventions. This allows components of programs that are only responsible for performing calculations rather than meeting real-time constraints, to be developed using the simpler untimed calculus.

The notational conventions allow abbreviated reference to the values of variables at the start and finish times of a specification command, as well as reference to the value of a variable at other times via the use of the trace notation. If a specification only uses the before and after state conventions, i.e., it is a calculational component, then the simpler untimed refinement laws may be used. To justify this approach, we have presented both a timed and untimed semantics for predicates using this convention, and shown that if a predicate P holds for all untimed states, then the corresponding predicate in the timed model, $P \mathbin{@} ({`}\tau, \tau')$, holds for all states in the timed model.

The overall objective of our work is to provide a systematic method for the development of real-time programs, and to provide tool support for such a method. This paper is a step towards a method that supports both the sequential real-time refinement calculus, and embedding within that the standard sequential calculus. The use of the embedded untimed calculus allows simpler developments of program fragments that are not concerned with timing constraints.

Acknowledgements I would like to thank Ray Nickson and Peter Robinson for pointing out possible ambiguities in earlier versions of the notation, and David Carrington, Andy Evans and Luke Wildman for feedback on earlier drafts of this paper.

References

1. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.
2. I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop (NFMW'96).* Springer, 1997.
3. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. Technical Report UQ-SVRC-97-33, Software Verification Research Centre, The University of Queensland, URL http://svrc.uq.edu.au, 1997.
4. I. J. Hayes and M. Utting. Deadlines are termination. In D. Gries and W.-P. de Roever, editors, *IFIP TC2/WG2.2, 2.3 International Conference on Programming Concepts and Methods (PROCOMET'98)*, pages 186–204. Chapman and Hall, 1998.
5. Eric C. R. Hehner. Predicative programming: Parts i and ii. *Comm. ACM*, 27(2):134–151, February 1984. Corrigendum: Comm. ACM Vol. 27 No. 6 (June 1984) p. 593.
6. C. C. Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.
7. 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.