SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-48

# A Real-Time Refinement Calculus that Changes Only Time

Mark Utting and Colin Fidge

July 1996
(Revision 1.1)

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

**Note:** Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

# A real-time refinement calculus that changes only time

Mark Utting Colin Fidge
Software Verification Research Centre,
Department of Computer Science,
The University of Queensland,
Queensland 4072, Australia.
{marku,cjf}@cs.uq.edu.au

**Abstract**

The behaviour of a real-time system that interacts repeatedly with its environment is most succinctly specified by its possible traces, or histories. We present a way of using the refinement calculus for developing real-time programs from requirements expressed in this form. Our trace-based specification statements and target language constructs constrain the traces of system variables, rather than updating them destructively like the usual state-machine model. The only variable that is updated is a special current-time variable. The resulting calculus allows refinement from formal specifications with hard real-time requirements, to high-level language programs annotated with precise timing constraints.

## 1 Introduction

Refinement rules that preserve the functional requirements of procedural programs are now well established [18]. Such rules do not support real-time requirements, however. The *Quartz* project is defining a refinement method for multi-tasking applications with hard real-time constraints [3]. This paper illustrates one aspect of this work, refinement rules for generating sequential real-time code. (A companion paper describes the procedure for creating concurrent real-time task specifications [4].)

The method is an application of the standard refinement calculus [18], with conventions borrowed from two previous streams of real-time refinement research. It uses timed traces as in the 'timed' refinement calculus [14] and a special time variable inspired by the Temporal Agent Model [24]. Unusual features of our calculus are that code execution is modelled by constraining timed trace variables and updating *only* the special time variable, and the use of specifications that are feasible only in the context of other sequential components.

## 2 Motivation

There have been a number of attempts at producing 'real-time' refinement calculi. In particular, the *timed refinement calculus* [12, 14, 13] offers an elegant method for the design of parallel real-time systems. Variables are modelled as traces over the entire time domain. Refinement rules allow for partitioning these traces into disjoint *processes*. Each process has two sets of variables available to it, the *external* variables, over which it has no control, and the *constructed* variables, which it must create. Accordingly, a specification has separate *assumption* and *effect* predicates, specifying its anticipated environment and required goals, respectively. Constructed variables may appear in the effect, but not in the assumption. Refinement rules restrict the form of these specifications to allow each parallel process to be further refined in isolation from the others, even when there are data dependencies between them. Unfortunately, however, the calculus has no rules for introducing sequential programming constructs because no way is provided of dividing timed traces into sequentially-composed segments.

Conversely, the *Temporal Agent Model* (TAM) [22, 24] offers a natural way of constructing timed traces from sequential components. Its 'chop' operator allows a trace to be defined in terms of adjacent subsequences, and its free

'current time' variables allow the precise timing behaviour of target language constructs to be modelled. This makes refinement rules for introducing timed sequential programming constructs possible. However the TAM model for concurrency is less satisfactory. Its single timing function over all variables, and lack of separate preconditions, makes complete partitioning of variables belonging to concurrent processes impossible; in subsequent refinements there is always the danger of interference between 'independent' processes due to the use of conjunction for representing parallelism.

Our goal therefore is to use the best features from these two calculi to create a practical real-time refinement calculus. Elsewhere we have already shown how the timed refinement calculus can be used to develop skeletal real-time multi-tasking programs [4]. In this paper we complete this program development procedure by adapting the standard refinement calculus with concepts borrowed from both of the above methods to enable refinement of individual task specifications to sequential programming language code.

# 3 Definitions

Z [26] mathematical notation is used in the following definitions and predicates.

## 3.1 Time

For the purposes of this paper we assume discrete time. Let

$$\mathbb{A} \quad == \quad \mathbb{N}$$

denote the *absolute time* domain, and

$$\mathbb{D} \quad == \quad \mathbb{N}$$

denote *durations* of time. The *execution time* of each system component is given as a non-empty set of durations,

$$\mathbb{E} \quad == \quad \mathbb{P}_1 \mathbb{D} \ ,$$

to allow for possible uncertainty in its precise timing behaviour. A 'set addition' operator $\dot{+}$ adds two such sets by summing every pair of elements.

$$\_ \dot{+} \_ : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$$

$$\forall X, Y : \mathbb{E} \bullet X \dot{+} Y = \{x : X; \ y : Y \bullet x + y\}$$

For example,

$$\{7, 10\} \dot{+} \{2, 3\} \quad = \quad \{9, 10, 12, 13\} \ .$$

Not suprisingly, adding two 'uncertain' timings results in an even broader range of possible execution times.

## 3.2 Specification statements

In our calculus, a process creates a complete trace (history) for each of its constructed variables over all time. In this paper, we focus on a single process that constructs a set of variables $\vec{p}$. Each $x \in \vec{p}$ must be a *timed variable* of type

$$x : \mathbb{A} \to T \ ,$$

for some 'base' type $T$, defining the evolution of the variable over all time. A process may also refer to a set of external variables $\vec{u}$ representing inputs from its environment. External variables are typically, but not necessarily, also represented as functions over the time domain.

To allow these traces to be constructed incrementally, as is necessary to model an imperative programming language, we introduce a special *time* variable, $\tau : \mathbb{A}$. This denotes the current absolute time. Its value may be increased by language statements, but cannot be decreased. The purpose of $\tau$ is to partition the traces into contiguous segments

which can be refined independently. For example, in the program fragment '$S_1$ ; $S_2$ ; $S_3$', each of the $S_i$ statements typically increases $\tau$ and constructs the trace of the $\vec{p}$ variables between times $\tau_0$ and $\tau$ (*excluding* $\tau_0$, but *including* $\tau$).

We adopt an unfamiliar approach to the construction of timed variables: each statement *constrains* a segment of the original trace rather than *updating* the trace variable. This gives an elegant calculus, since we only ever have one set of trace variables to reason about for the process (rather than an initial and final set of traces for each statement). The only variable that actually gets updated destructively in the usual way is $\tau$.

These differences between our calculus and the standard refinement model make it convenient to define a new form of *trace constraining* specification statement, distinguished by the decoration '$\star$'. It abbreviates a standard specification statement [18] as follows:

$$\star\vec{v}{:}\,[A,\ E] \ \stackrel{\text{def}}{=}\ \tau{:}\,\big[A,\ E \wedge \tau_0 \leq \tau \wedge stable(\vec{p} \setminus \vec{v}, \tau_0 \,..\, \tau)\big] \ .$$

We use timed refinement calculus terminology and refer to pre and postconditions as assumptions $A$ and effects $E$, respectively. In addition to the specified effect $E$, this definition introduces implicit requirements that absolute time $\tau$ cannot go backward, and that all the variables in $\vec{p}$, except those mentioned in the frame $\vec{v}$, stay unchanged for the duration of the statement. Unchangedness of a set of timed variables $\vec{w} \subseteq \vec{p}$, for a set of absolute times $A \subseteq \mathbb{A}$, is defined as follows:

$$stable(\vec{w}, A) \quad == \quad \wedge_{x \in \vec{w}}\, \forall\, t_1, t_2 : A \bullet x(t_1) = x(t_2) \ .$$

The weakest precondition semantics of a specification statement $\star\vec{v}{:}\,[A,\ E]$, with respect to a predicate $P$ over $\vec{u}$, $\vec{v}$ and $\tau$, follows from the standard definition [18]:

$$wp(\star\vec{v}{:}\,[A,\ E], P) \ \stackrel{\text{def}}{=}\ A \wedge \big(\forall\, \tau : \mathbb{A} \bullet (E \wedge \tau_0 \leq \tau \wedge stable(\vec{p} \setminus \vec{v}, \tau_0 \,..\, \tau)) \Rightarrow P\big) \left[\frac{\tau}{\tau_0}\right] \ .$$

We now describe this new specification statement in more detail.

1. The frame $\vec{v}$ defines the timed variables whose value this statement may change. It must be a subset of $\vec{p}$, the constructed variables that this process *creates*.

2. Assumption $A$ is a predicate over external variables $\vec{u}$, constructed variables $\vec{p}$, and the time variable $\tau$. It defines the anticipated state of external and constructed variables, and the absolute time, when the statement begins.

3. Effect $E$ is a predicate over external variables $\vec{u}$, constructed variables $\vec{p}$, and initial and final values of the time variable ($\tau_0$ and $\tau$). The usual refinement calculus conventions for zero-subscripted variables apply [18, ch.6]. Intuitively, the role of $E$ is to define the values of the constructed variables $\vec{p}$ from time $\tau_0$ (exclusive) up to time $\tau$ (inclusive).

Note that there is no restriction to prevent a specification from referring to the value of variables at times outside the range $\tau_0 \,..\, \tau$. Indeed, it is often useful in high-level specifications to refer to the past, or even future, behaviour of external variables. However, such references must be removed during refinement, because executable target language constructs can access variable values only during the period the construct is executing (Section 3.5).

## 3.3 Creating traces

Since the specification statement introduced above operates only on a trace segment, we need some way of defining the trace initially when specifying a whole process. This is done using the following construct:

$$create\ \vec{p} : \vec{T} \bullet S \ .$$

This operator creates undetermined traces for the process variables $\vec{p}$ and sets the starting time to zero. It has the following weakest precondition semantics:

$$wp(create\ \vec{p} : \vec{T} \bullet S, P) \ \stackrel{\text{def}}{=}\ \Big(\forall\, \vec{p} : \mathbb{A} \rightarrow \vec{T};\ \tau : \mathbb{A} \bullet \tau = 0 \Rightarrow wp(S, P)\Big) \ .$$

Here $P$ is a predicate over $\vec{u}$, $\vec{p}$ and $\tau$, and $S$ is a program (predicate transformer) over the same variables.

## 3.4 Feasibility

A surprising feature of this approach is that most of our trace constraining statements, including our implementable target language constructs, are *infeasible*! That is, for some specification $\star\vec{v}\colon [A,\ E]$, assumption $A$ does not necessarily entail effect $E$, because this specification statement *on its own* is not free to choose values for trace variables in $\vec{v}$. (The expansion to a traditional specification statement given in Section 3.2 shows that the $\vec{v}$ variables are not even in the frame!)

This does not prevent us from applying refinement rules, but in the standard refinement calculus it would mean that we could never reach implementable code. However, in our calculus, we *can* implement some sequences of these 'infeasible' statements, due to constraints imposed by the target language. Every executable process has the form $create\ \vec{p} : \vec{T} \bullet S$, and every statement $S_i$ executed within $S$ can constrain trace values only between times $\tau_0$ (exclusive) and $\tau$ (inclusive). This means that each statement $S_i$ operates on a segment of each trace that is disjoint from the segments constrained by any other statement within $S$. It is this disjointness that makes such statements implementable. Operationally, it allows the non-deterministic choice of each time-value pair $(t \mapsto v)$ in a trace to be delayed until execution reaches the statement that has $\tau_0 < t \leq \tau$.

For example, an assignment statement $x := E$ sets the value of $x$ at end time $\tau$ equal to the value of expression $E$ evaluated at start time $\tau_0$. If each such assignment statement consumed exactly two time units, its semantics would then be

$$\begin{array}{lcl} wp(x := E, P) & \stackrel{\text{def}}{=} & (\forall\, \tau = \tau_0 + 2 \bullet x(\tau) = E(\tau_0) \Rightarrow P)\left[\frac{\tau}{\tau_0}\right] \\ & = & x(\tau + 2) = E(\tau) \Rightarrow P\left[\frac{\tau+2}{\tau}\right] \ . \end{array}$$

So, for the sequence of assignments, '$x := 1\,;\, x := 2\,;\, x := x + 3$', starting at time zero, we have

$$\begin{array}{ll} & wp(create\ x : \mathbb{N} \bullet x := 1\,;\, x := 2\,;\, x := x + 3, P) \\ = & (\forall\, x : \mathbb{A} \rightarrow \mathbb{N};\ \tau : \mathbb{A} \bullet \\ & \quad \tau = 0 \Rightarrow \\ & \quad\quad x(2) = 1 \Rightarrow \\ & \quad\quad\quad x(4) = 2 \Rightarrow \\ & \quad\quad\quad\quad x(6) = x(4) + 3 \Rightarrow P\left[\frac{\tau+6}{\tau}\right]) \\ = & (\forall\, x : \mathbb{A} \rightarrow \mathbb{N} \mid x(2) = 1 \wedge x(4) = 2 \wedge x(6) = 5 \bullet P\left[\frac{6}{\tau}\right]) \ . \end{array}$$

Note that we can deduce that this program fragment finishes at time $\tau = 6$, and the value of $x$ at various times up until then, but cannot deduce anything about *future* values of $x$, as one would expect.

This illustrates how, for some process specification $create\ \vec{p} : \vec{T} \bullet S$, the effect of *every* statement in $S$ constrains the initial choice of traces for each $x \in \vec{p}$. The requirement that $\tau$ cannot be decreased is used to guarantee that each statement constrains a unique segment of each trace.

Interestingly, Morgan notes that some data refinement applications are simplified if 'miraculous' specifications are allowed in intermediate development steps [17]. Our timed refinement method can be viewed as an extension of this idea, where miraculous statements are not only used during development, but are also part of the executable target language.

## 3.5 Target programming language

This section defines a target programming language suitable for use with our calculus. In particular, each of the language constructs preserves the disjointness property identified above by allowing trace values to be constrained only at times between $\tau_0$ and $\tau$.

Ideally we want a target programming language which allows timing constraints on code fragments to be expressed directly. Although a handful of programming languages with first-class timing constructs have been proposed [25, 11, 10, 5], none has become widely established. Therefore, for the purposes of this paper, we target Dijkstra's guarded command language augmented with timing annotations. The definition of each executable code construct is given in Figure 1 and discussed further in this section. Note that $x \in \vec{v}$ is a constructed timed variable; $y \in \vec{u}$ is an external variable; $S, S_1, S_2, \ldots$ are statements in our language; $\mathsf{a}, \mathsf{b}, \ldots \in \mathbb{E}$ denote execution time overheads required for the implementation of target language constructs; $G, G_1, G_2, \ldots$ are 'untimed' boolean expressions in the target language; and $E$ is an 'untimed' general expression in the target language.

**D1** $x :=^{\mathsf{d}} E \stackrel{\text{def}}{=} \star x\colon [\text{true},\ \tau - \tau_0 \in \mathsf{d} \wedge x(\tau) = E(\tau_0)]$

**D2** $\mathbf{read}(x, y)^{\mathsf{d}} \stackrel{\text{def}}{=} \star x\colon [\tau \,..\, \tau + max\ \mathsf{d} \subseteq \text{dom}\ y,\ \tau - \tau_0 \in \mathsf{d} \wedge x(\tau) \in y(\!|\ \tau_0 \,..\, \tau\ |\!)]$

**D3** $\mathbf{delay\text{-}until}(E)^{\mathsf{d}} \stackrel{\text{def}}{=} \star[\tau \leqslant E(\tau),\ \tau - E(\tau_0) \in \mathsf{d}]$

**D4** $\mathbf{get\text{-}time}(x)^{\mathsf{d}} \stackrel{\text{def}}{=} \star x\colon [\text{true},\ \tau - \tau_0 \in \mathsf{d} \wedge x(\tau) \in \tau_0 \,..\, \tau]$

**D5** $\mathbf{skip} \stackrel{\text{def}}{=} delay(\{0\})$

**D6** $S_1 \,;\, S_2 \stackrel{\text{def}}{=} S_1 \circ S_2$

**D7** $\mathbf{if}\ G_i \rightarrow^{\mathsf{e}_i} S_i^{\mathsf{x}_i}\ \mathbf{fi} \stackrel{\text{def}}{=} (\sqcap i \bullet G_i(\tau) \rightarrow delay(\mathsf{e}_i) \,;\, S_i \,;\, delay(\mathsf{x}_i)) \sqcap (\neg(\vee i \bullet G_i(\tau)) \rightarrow abort)$

**D8** $\mathbf{do}\ G \rightarrow^{\mathsf{e}} S\ \mathbf{od}^{\mathsf{x}} \stackrel{\text{def}}{=} \mu F \bullet ((G(\tau) \rightarrow delay(\mathsf{e}) \,;\, S \,;\, F) \sqcap (\neg\ G(\tau) \rightarrow delay(\mathsf{x})))$

Figure 1: Target language definition.

Target language boolean guards $G$ and general expressions $E$ are represented as *untimed expressions* involving constructed variables $x \in \vec{v}$ without time dereferences [23, p.8]. (Variables $\vec{u}$ from other processes cannot be accessed.) Any such expression can be instantiated for some absolute time $t \in \mathbb{A}$ by substituting each timed variable $x$ it contains with $x(t)$ [16, p.4]. For a boolean expression $G$ in the target language define

$$G(t) \quad == \quad G\left[\frac{\vec{x}(t)}{\vec{x}}\right] ,$$

and likewise for $E(t)$.

**Assignment** is denoted $v :=^{\mathsf{d}} E$. Annotation $\mathsf{d}$ is the set of allowable execution times for the whole statement. Definition **D1** leaves all process variables $\vec{p}$ unchanged except for target variable $x$. It requires the duration between $\tau_0$ and $\tau$ to be some member of set $\mathsf{d}$, and makes the final value of $x$ at time $\tau$ equal the value of expression $E$ evaluated at time $\tau_0$. Note that the values of $x$ between $\tau_0$ and $\tau$ are not specified, to allow for implementations that update $x$ non-atomically.

**Sequential composition** of two statements $S_1$ and $S_2$ usually incurs no execution time overhead, so the conventional $S_1 \,;\, S_2$ programming notation is used (Definition **D6**). Thus sequential composition is simply function composition of predicate transformers. (However a time-annotated sequential composition operator, that allows the passage of time between its arguments, would be necessary if we were to allow, for instance, debugging code to be inserted between statements.)

**Choice** is denoted $\mathbf{if}\ G_i \rightarrow^{\mathsf{e}_i} S_i^{\mathsf{x}_i}\ \mathbf{fi}$. For the $i^{\text{th}}$ alternative, $\mathsf{e}_i$ is the overhead of evaluating the appropriate guard(s) and branching to the body $S_i$ of that alternative, and $\mathsf{x}_i$ is the overhead of exiting the **if**...**fi** construct once $S_i$ has finished. Definition **D7** demonically chooses a guarded alternative and introduces the implementation overheads as delays. If all guards are false the choice statement 'aborts'.

The $delay$ statement in Figure 1 is not considered to be part of the executable target language, but is defined as:

$$delay(\mathsf{d}) \quad \stackrel{\text{def}}{=} \quad \star[\text{true},\ \tau - \tau_0 \in \mathsf{d}] \ .$$

This is an idealised relative delay statement that allows no possibility of 'overrun'. No process variables are changed by delaying, so the frame is empty.

As usual a statement that aborts is considered to behave unpredictably. In a reactive context, however, it can be argued that this should be modelled differently than the usual chaotic **abort** statement [18, p.12] because we cannot revoke outputs already produced by the process before it failed. We therefore define

$$abort \quad \stackrel{\text{def}}{=} \quad \star\vec{p}\colon [\text{true},\ \tau_0 \leqslant \tau]$$

which may take *any* amount of time, and leaves all process variables $\vec{p}$ unconstrained after time $\tau_0$.

**Iteration** is denoted $\mathbf{do}\ G \rightarrow^{\mathsf{e}} S\ \mathbf{od}^{\mathsf{x}}$. Annotation $\mathsf{e}$ is the overhead of evaluating the (true) guard and reaching the body $S$ at each iteration, and $\mathsf{x}$ is the overhead of evaluating the (false) guard and exiting the loop. Definition **D8** uses the conventional fixed-point method to define iteration [19], but adds the timing impact of its implementation overheads.

**Skip** is defined trivially by definition **D5** as a delay of duration zero.

For the example in Section 5 we also need some embedded programming constructs.

**Suspending** the process is denoted $\mathbf{delay\text{-}until}(E)^{\mathsf{d}}$. This is an important capability for a real-time programming language, allowing activity to be delayed until after a particular absolute time $E$. Annotation $\mathsf{d}$ is the acceptable overrun of the delay implementation beyond the specified time. (Again recall that absolute-time-valued expression $E$ may refer to variables from $\vec{p}$ only, so is stable while being evaluated.) Definition **D3** does not specify the behaviour of the statement if time $E$ has already passed.

**Reading** a value from the environment is denoted $\mathbf{read}(x, y)^{\mathsf{d}}$. Many embedded applications allow the ability to read from a memory-mapped i/o register. Here $x \in \vec{p}$ is the target variable, $y \in \vec{u}$ is the source register, and $\mathsf{d}$ is the set of acceptable execution times for the whole operation. The assumption in definition **D2** requires there to be a defined value for $y$ at any time it may be accessed. The effect is much the same as that of assignment except that the value placed in $x$ may be *any* value of $y$ between the starting and finishing times.

**Real-time clock access** is denoted $\mathbf{get\text{-}time}(x)^{\mathsf{d}}$, where $x \in \vec{p}$ is a variable that will be assigned the 'current' absolute time and $\mathsf{d}$ is the set of acceptable durations for the operation. Definition **D4** is again similar to assignment except that the value assigned to $x$ is some absolute time between $\tau_0$ and $\tau$. (When the statement finishes the value held in $x$ may thus differ from the 'true' time by up to $max\ \mathsf{d}$ time units.)

Note that the programmer is not obliged to supply numbers for each superscripted timing annotation during code development. Indeed, to do so may unnecessarily inhibit later object code generation [3]. In practice it is desirable to leave these annotations as symbolic constants that will be instantiated by a suitably instrumented analysis tool such as a real-time compiler [6] or timing analyser [20][21, p.39]. Of course the programmer may choose to instantiate the time sets with anticipated values in order to check timing feasibility of the emerging system at any stage of development.

# 4 Refinement rules

The definitions in Figure 1 allow the assignment, read, suspend, clock access and skip constructs to be introduced directly. Also, we inherit those standard refinement calculus laws that do not rely on the target language such as 'weaken precondition (assumption)' [18, p.8], 'strengthen postcondition (effect)' [18, p.54], 'introduce logical constant' [18, p.53] and 'remove logical constant' [18, p.53]. However the compound timed language constructs above, sequential composition, choice and iteration, require derived rules if they are to be introduced easily.

Let $\star\vec{v}\colon [A,\ E]$ be a well-typed specification statement as defined in Section 3.2. Sequential composition can be introduced readily. The general rule is the standard refinement calculus one [18, p.56]. (Provisos appear above the line and the refinement rule below.)

**Law S1a:** Introduce sequential composition

$$\frac{\text{constant } T \text{ is fresh}}{\begin{array}{ll} & \star\vec{v}\colon [A,\ E] \\ \sqsubseteq & \left\| \left[ \mathbf{con}\ T = \tau \bullet \star\vec{v}\colon [A,\ M] \right] ; \star\vec{v}\colon \left[ M\left[\frac{T}{\tau_0}\right],\ E\left[\frac{T}{\tau_0}\right] \right] \right\| \end{array}}$$

$M$ is a predicate on $\vec{u}$, $\vec{v}$, $\tau_0$ and $\tau$. Effectively, only time variable $\tau$ changes. Logical constant $T$ captures the starting time and, as usual for such constants, must later be 'refined away' [18, p.52].

Often, however, we are concerned only with the overall duration of a specification, not its place in absolute time. In such situations the following derived rule is useful.

**Law S1b:** Introduce sequential composition

$$\frac{\begin{array}{l}\mathsf{d}_1 \dot{+} \mathsf{d}_2 \subseteq \mathsf{d} \\ \tau_0 \text{ does not appear free in } M \text{ or } E\end{array}}{\begin{array}{ll} & \star\vec{v}\colon [A,\ E \wedge \tau - \tau_0 \in \mathsf{d}] \\ \sqsubseteq & \star\vec{v}\colon [A,\ M \wedge \tau - \tau_0 \in \mathsf{d}_1]\ ;\star\vec{v}\colon [M,\ E \wedge \tau - \tau_0 \in \mathsf{d}_2]\end{array}}$$

Here $\mathsf{d}$ represents the required duration(s) of the original specification and $\mathsf{d}_1$ and $\mathsf{d}_2$ are the execution times of the sequential components. The first proviso ensures that the execution time of the two statements in sequence will satisfy the original requirement. The second proviso ensures that $M$ and $E$ do not refer to their starting time because it differs for their different appearances in the rule [18, p.56].

Choice statements are complicated by the initial implementation overhead required to reach the chosen alternative and possibly a final overhead in exiting the construct.

**Law S2a:** Introduce alternatives

$$\frac{\begin{array}{l}\text{constant } T \text{ is fresh} \\ A \Rrightarrow \vee i \bullet G_i(\tau)\end{array}}{\begin{array}{ll} & \star\vec{v}\colon [A,\ E] \\ \sqsubseteq & \mathbf{con}\ T = \tau \bullet \\ & \mathbf{if} \\ & \quad G_i \rightarrow^{\mathsf{e}_i} \star\vec{v}\colon \left[\begin{array}{l} A\left[\frac{T}{\tau}\right] \wedge G_i(T) \\ \wedge\ \tau - T \in \mathsf{e}_i \\ \wedge\ stable(\vec{p}, T\,..\,\tau)\end{array}\right., \forall U \mid (U - \tau \in \mathsf{x}_i \wedge stable(\vec{p}, \tau\,..\,U)) \bullet E\left[\frac{T}{\tau_0}, \frac{U}{\tau}\right]\Bigg]^{\mathsf{x}_i} \\ & \mathbf{fi}\end{array}}$$

As usual the main proviso is that at least one of the guards $G_i$ is true when the statement begins [18, p.48]. (Recall that assumption $A$ includes $\tau$, and the '$delay$' in definition **D7** ensures the variables in the guard are stable while the expression is evaluated.) Constant $T$ serves to capture the starting time of the whole construct because, due to entry overheads, this is different from the starting time of each alternative. The assumption and effect are then modified to account for the entry and exit overheads, $\mathsf{e}_i$ and $\mathsf{x}_i$, of each alternative. The assumption notes that some time in $\mathsf{e}_i$ has passed since $T$ and that all process variables have been stable during this interval. The effect requires us to establish $E$ at some absolute time $U$ after a subsequent delay of *any* time in $\mathsf{x}_i$. (This effect is thus $wp(delay(\mathsf{x}_i), E)$.)

Again a convenient derived rule covers the common situation where only the overall duration is of interest. To give freedom to adjust timing endpoints we need to prevent the specifier from constraining their position in absolute time. This can be done by requiring that $\tau_0$ and $\tau$ are used in a predicate *only* to dereference timed variables. This is expressed in the following rule as the 'only dereference' proviso.

**Law S2b:** Introduce alternatives

$$\frac{\begin{array}{l}\wedge i \bullet \mathsf{e}_i \dot{+} \mathsf{d}_i \dot{+} \mathsf{x}_i \subseteq \mathsf{d} \\ \tau_0 \text{ and } \tau \text{ only dereference in } A \text{ and } E \\ A \Rrightarrow \vee i \bullet G_i(\tau)\end{array}}{\begin{array}{ll} & \star\vec{v}\colon [A,\ E \wedge \tau - \tau_0 \in \mathsf{d}] \\ \sqsubseteq & \mathbf{if}\ G_i \rightarrow^{\mathsf{e}_i} \star\vec{v}\colon [A \wedge G_i(\tau),\ E \wedge \tau - \tau_0 \in \mathsf{d}_i]^{\mathsf{x}_i}\ \mathbf{fi}\end{array}}$$

Since $\tau$ is not fixed in absolute time by assumption $A$, the third proviso implicitly requires that at least one of the guards $G_i$ must be true at *any* time the statement may be executed!

Introducing iteration is especially complex in a timed scenario. A time-dependent loop invariant does not necessarily hold everywhere outside the loop body because the mere act of evaluating the loop guard may make a true invariant false due to the passage of time. Therefore we adopt the idea that a 'timed' loop invariant is expected to be true only at the moment the guard is *about to be* evaluated [2].

As in rules **S1a** and **S2a** we wish to use a logical constant to capture the starting time of each iteration. However since this must be done *before* the loop guard is evaluated we are forced to introduce a special notation for this:

$$docon\ C \bullet G \rightarrow^{\mathsf{e}} S\ od^{\mathsf{x}}\ .$$

The formal definition of this temporary construct is the same as definition **D8** with a '$\mathbf{con}\ C$' declaration surrounding the 'true guard' alternative. A $docon$ construct can be refined to a target language $\mathbf{do}$ statement using the usual 'remove logical constant' law [18, p.53], when the constant does not appear in $G$ or $S$.

**Law S3:** Introduce iteration

$$\frac{\begin{array}{l}\text{constant } T \text{ is fresh}\\ \tau_0 \text{ does not appear free in } E\\ A \Rrightarrow inv\\ \star[inv \wedge \neg\ G(\tau),\ E] \sqsubseteq delay(\mathsf{x})\end{array}}{\star\vec{v}\colon [A,\ E] \quad \sqsubseteq \quad docon\ T = \tau \bullet\ G \rightarrow^{\mathsf{e}} \star\vec{v}\colon \left[\begin{array}{l} inv\left[\frac{T}{\tau}\right] \\ G(T) \\ \tau - T \in \mathsf{e} \\ stable(\vec{p},\ T\,..\,\tau)\end{array}\ ,\ \begin{array}{l} inv \\ \\ 0 \le F_\tau < F_T \\ \\ \end{array}\right]\ od^{\mathsf{x}}}$$

Invariant $inv$ is a predicate on $\vec{u}$, $\vec{v}$ and $\tau$, and is expected to be true when the loop guard is about to be evaluated. The variant function $F$ is used to ensure termination after some finite number of iterations.[1] The last proviso allows for the final overhead of exiting the loop. It requires that the invariant is true and the guard false only *before* exit overhead $\mathsf{x}$ is encountered and that this extra delay will establish the desired effect. Similarly, in the loop body the assumption is that the invariant and guard were true at time $T$, before entry overhead $\mathsf{e}$ was encountered. (Here the range of times in $\mathsf{e}$ is assumed to be broad enough to cover the possibility that it takes longer to reach the body on the first iteration due to loop initialisation code.)

Finally we note a very useful consequence of our modelling method. Since all timed variables are defined over the whole time domain, and only $\tau$ changes, our calculus enjoys the following equivalence for any assertion predicate $P$ and statement $S$.

**Law S4:** Carry assertion forward

$$\frac{\tau \text{ is not free in } P}{\{P\}\,;S = \{P\}\,;S\,;\{P\}}$$

Predicate $P$ can be any assertion involving variables $\vec{u}$ and $\vec{v}$, as long as it is not dependent on the *current* time.

# 5 Example: message receiver

As a practical example consider a system that receives messages from a memory-mapped i/o location. We use Z [26] schema notation for expressing types and predicates. Let given set

$$[\mathbb{C}]$$

be the set of incoming characters, with a subset of printable, i.e., non-control, characters, and two distinguished control characters for delimiting messages.

[1] In our calculus, it would also be possible to define an infinite loop construct. However, the standard fixed point definition of loops used in **D8** needs to be modified to handle infinite loops and this is a topic of ongoing research.

$$
\begin{array}{|l}
STX, ETX : \mathbb{C} \\
PrintChars : \mathbb{P}\,\mathbb{C} \\
\hline
STX \neq ETX \\
STX \notin PrintChars \\
ETX \notin PrintChars
\end{array}
$$

We define well-formed incoming messages $\mathbb{M}$ as any sequence of characters $s$ (excluding $ETX$) delimited by $STX$ and $ETX$,

$$\mathbb{M} \quad == \quad \{s : \mathrm{seq}(\mathbb{C} \setminus \{ETX\}) \bullet \langle STX \rangle \frown s \frown \langle ETX \rangle\} \ .$$

Such a message appears at the input location starting at a particular absolute time.

$$
\begin{array}{|l}
start : \mathbb{A}
\end{array}
$$

Characters appear regularly with period $charsep$ but each character only remains reliably readable for $chardef$ time units.

$$
\begin{array}{|l}
charsep, chardef : \mathbb{D} \\
\hline
chardef \leqslant charsep
\end{array}
$$

The input stream is of type $\mathbb{A} \pfun \mathbb{C}$. Some syntactic conveniences for manipulating this function, given the above constants, are defined below. Let $c$ be of type $\mathbb{A} \pfun \mathbb{C}$, with at least one $ETX$ character appearing after $start$, and $A$ of type $\mathbb{P}\,\mathbb{A}$.

$$
\begin{array}{rcl}
samples(c) & == & \{n : \mathbb{N} \bullet start + n * charsep\} \lhd c \\
etxtime(c) & == & min\,\mathrm{dom}(samples(c) \rhd \{ETX\}) \\
edges(c) & == & (\mathrm{dom}\, samples(c)) \cap (start \,..\, etxtime(c)) \\
partmsg(c, A) & == & squash(A \lhd samples(c)) \\
fullmsg(c) & == & partmsg(c, edges(c))
\end{array}
$$

Notation $samples$ restricts $c$ to those times at which characters are expected to appear, starting at time $start$. However, to recognise the end of a well-formed message, $etxtime$ gives the earliest time at which the $ETX$ character appears in this unbounded set of samples. The $edges$ abbreviation then yields the starting time of each character in the current well-formed message up to and including the $ETX$ character. Given some set of absolute times $A$, $partmsg$ returns the sequence of characters from the message sampled at those times. This is used for extracting prefixes of the message up to an absolute time. Lastly, $fullmsg$ returns the complete message.

The input variable is then defined as follows.

*In*

$$
\begin{array}{|l}
in : \mathbb{A} \pfun \mathbb{C} \\
\hline
fullmsg(in) \in \mathbb{M} \\
\forall\, t : edges(in) \bullet t \,..\, t + chardef \subseteq \mathrm{dom}\, in \land stable(in, t \,..\, t + chardef)
\end{array}
$$

In other words, from time $start$ there is a well-formed message where each character is defined for at least $chardef$ time units and is stable for this period.

There are three variables in our target system.

*Out*

$$
\begin{array}{|l}
msg : \mathbb{A} \fun \mathrm{seq}\,\mathbb{C} \\
char : \mathbb{A} \fun \mathbb{C} \\
next : \mathbb{A} \fun \mathbb{A}
\end{array}
$$

The principal output is $msg$ which will contain the received message. Variables $char$ and $next$ are used to hold the most recently read character, and the time at which the next character is expected to arrive, respectively.

The top-level specification is then as follows.

$$\star Out\colon [GoodMsg,\ GetMsg] \tag{1}$$

In the following refinement let $\mathsf{a}, \mathsf{b}, \ldots \in \mathbb{E}$ denote sets of acceptable execution times, typically denoting worst-case requirements as contiguous sets starting at zero. (In the general case these time sets may be defined by expressions dependent on the absolute time, system state or both. Such precision is rarely needed, however, and constant sets usually suffice.) For ease of expression in the text we take the liberty of referring to these sets as if they are singletons.

Overall assumption $GoodMsg$ states that $msg$ is initially the empty sequence and the current time is some, hopefully short, time $\mathsf{a}$ after $start$. Delay $\mathsf{a}$ represents the time required to initiate the sporadic $GetMsg$ activity. (For instance, $\mathsf{a}$ may be the time required to initiate an external interrupt handler; this delay is outside the scope of the refinement undertaken here, but is included for realism.)

*GoodMsg*

$$\tau : \mathbb{A};\ In;\ Out$$

$$msg(\tau) = \langle\,\rangle$$
$$\tau \in \{start\} \dot{+} \mathsf{a}$$

The desired effect $GetMsg$ is to place the message, less all control characters it contains, in $msg$, and to do so in a time proportional to the length of the message.

*GetMsg*

$$\tau : \mathbb{A};\ In;\ Out$$

$$msg(\tau) = fullmsg(in) \upharpoonright PrintChars$$
$$\tau \leqslant etxtime(in) + charsep$$

(The timing requirement here is rather tight, but allows for the possibility that the following message appears immediately after the final character of the current message.)

Recognising that an iterative solution will be needed, refinement begins by introducing an invariant.

(1) $\sqsubseteq$ "by **S1a**; remove constant"

$$\star Out\colon [GoodMsg,\ Inv]\ \mathbf{;} \tag{2}$$
$$\star Out\colon [Inv,\ GetMsg] \tag{3}$$

The invariant requires that we be within $\mathsf{x}$ time units of the leading edge of a character in the incoming message, that the 'current' character can be found in variable $char$, that the $msg$ accumulated to date contains all characters up to but excluding the current one, and that the $next$ variable holds a time within $\mathsf{z}$ units of the 'edge' time.

*Inv*

$$\tau : \mathbb{A};\ In;\ Out$$

$$\exists\, e : edges(in) \mid \tau \in \{e\} \dot{+} \mathsf{x} \bullet$$
$$char(\tau) = in(e)$$
$$\wedge\ \ msg(\tau) = partmsg(in, start\,..\,e-1) \upharpoonright PrintChars$$
$$\wedge\ \ next(\tau) \in \{e\} \dot{+} \mathsf{z}$$

The initial value of $msg$ already satisfies $Inv$, so we divide specification (2) into two parts which initialise $next$ and $char$, respectively.

(2) $\sqsubseteq$ "by strengthen effect"

$$\star Out\colon \big[GoodMsg,\ [\,Inv \mid \tau - \tau_0 \in \mathsf{b} \dot{+} \mathsf{c}\,]\big]$$

$\sqsubseteq$ "by **S1b**"

$$\star Out\colon \big[GoodMsg,\ [\,InitNext \mid \tau - \tau_0 \in \mathsf{b}\,]\big]\ \mathbf{;} \tag{4}$$
$$\star Out\colon \big[InitNext,\ [\,Inv \mid \tau - \tau_0 \in \mathsf{c}\,]\big] \tag{5}$$

*InitNext*

$\tau : \mathbb{A};\ In;\ Out$

---

$stable(\{msg\}, start\,..\,\tau)$
$next(\tau) \in start\,..\,\tau$
$\tau \in \{start\} \dot{+} \mathsf{a} \dot{+} \mathsf{b}$

These two sequential components can then be refined to code using the definitions in Figure 1.

(4) $\sqsubseteq$ "by weaken assumption; strengthen effect; **D4**"

$\mathbf{get\text{-}time}(next)^{\mathsf{b}}$

Strengthening the effect to match **D4** introduces a timing proof obligation

$\mathsf{a} \dot{+} \mathsf{b} \subseteq \mathsf{z}$

to ensure the time is read within $\mathsf{z}$ time units.

Similarly, a **read** can be immediately introduced.

(5) $\sqsubseteq$ "by weaken assumption; strengthen effect; **D2**"

$\mathbf{read}(char, in)^{\mathsf{c}}$

Strengthening the effect to allow **D2** to be applied generates timing obligation

$\mathsf{a} \dot{+} \mathsf{b} \dot{+} \mathsf{c} \quad \subseteq \quad \mathsf{x}\ .$

Furthermore, weakening the assumption also requires

$\mathsf{a} \dot{+} \mathsf{b} \dot{+} \mathsf{c} \quad \subseteq \quad 0\,..\,chardef$

in order to know the position of the statement in absolute time so that a 'good' character is read. The third predicate in schema $InitNext$ makes this possible.

The next step is to introduce iteration.

(3) $\sqsubseteq$ "by **S3**"

$$docon\ T = \tau \bullet char \neq ETX \rightarrow^{\mathsf{d}} \star Out\colon [StartLoop,\ InvTerm]\ od^{\mathsf{e}} \tag{6}$$

At the beginning of the each iteration we know that $\mathsf{d}$ time units has passed since the invariant and guard were true.

*StartLoop*

$Inv\left[\frac{T}{\tau}\right]$

---

$char(T) \neq ETX$
$\tau - T \in \mathsf{d}$
$stable(Out, T\,..\,\tau)$

*InvTerm*

$Inv$

---

$0 \leq etxtime(in) + charsep - \tau$
$\quad < etxtime(in) + charsep - T$

The proviso in rule **S3** requires us to show that this loop will establish $GetMsg$ when exit overhead $\mathsf{e}$ is encountered, so we need to know that

$\mathsf{x} \dot{+} \mathsf{e} \quad \subseteq \quad 0\,..\,charsep\ .$

The inequality conditions in $InvTerm$ can be refined away immediately, because $\tau \leq etxtime(in) + charsep$ follows from $Inv$, and we can ensure that time advances on each iteration ($T < \tau$) by adding the timing obligation $0 \notin \mathsf{d}$ (this is a constraint that all target language loop implementations satisfy).

(6) $\sqsubseteq$ "by strengthen effect"

$$\star Out\colon [StartLoop,\ Inv] \tag{7}$$

The loop body can be divided into four parts, one to update each of the three constructed variables, and one to keep the loop synchronised with the incoming data stream.

$$
\begin{array}{lll}
(7) & \sqsubseteq & \text{“by (\textbf{S1a}; remove constant; \textbf{S4}) } \times 3\text{”} \\
 & & \star Out\colon [StartLoop,\ UpdateMsg]\ \textbf{;} \qquad (8) \\
 & & \star Out\colon [UpdateMsg,\ UpdateNext]\ \textbf{;} \qquad (9) \\
 & & \star Out\colon [UpdateNext,\ Wait]\ \textbf{;} \qquad (10) \\
 & & \star Out\colon [Wait,\ Inv] \qquad (11)
\end{array}
$$

As discussed in Section 4, law **S4** allows us to carry any assertion that does not involve $\tau$ forward through a program. We achieve this effect here by adding the loop invariant at time $T$ to the new effects in the above sequence.

*UpdateMsg*

$$
\begin{array}{|l}
\hline
Inv\left[\frac{T}{\tau}\right] \\
\hline
\tau - T \in \mathsf{d} \dot{+} \mathsf{y} \\
stable(\{next\},\ T\,..\,\tau) \\
msg(\tau) = partmsg(in, start\,..\,T) \upharpoonright PrintChars \\
\hline
\end{array}
$$

*UpdateNext*

$$
\begin{array}{|l}
\hline
Inv\left[\frac{T}{\tau}\right] \\
\hline
\tau - T \in \mathsf{d} \dot{+} \mathsf{y} \dot{+} \mathsf{k} \\
next(\tau) = next(T) + charsep \\
msg(\tau) = partmsg(in, start\,..\,T) \upharpoonright PrintChars \\
\hline
\end{array}
$$

*Wait*

$$
\begin{array}{|l}
\hline
Inv\left[\frac{T}{\tau}\right] \\
\hline
\tau \in \{next(\tau)\} \dot{+} \mathsf{l} \\
next(\tau) = next(T) + charsep \\
msg(\tau) = partmsg(in, start\,..\,T) \upharpoonright PrintChars \\
\hline
\end{array}
$$

Each of these four components can then be readily refined to code. The second component becomes an assignment with duration $\mathsf{k}$.

$$
\begin{array}{lll}
(9) & \sqsubseteq & \text{“by strengthen effect; weaken assumption; \textbf{D1}”} \\
 & & next :=^{\mathsf{k}} next + charsep
\end{array}
$$

The third component requires us to wait until a particular absolute time, with an overrun of up to $\mathsf{l}$ time units.

$$
\begin{array}{lll}
(10) & \sqsubseteq & \text{“by strengthen effect; weaken assumption; \textbf{D3}”} \\
 & & \mathbf{delay\text{-}until}(next)^{\mathsf{l}}
\end{array}
$$

Weakening the assumption introduces the following proof obligation

$$\mathsf{x} \dot{+} \mathsf{d} \dot{+} \mathsf{y} \dot{+} \mathsf{k} \quad \subseteq \quad 0\,..\,charsep\ ,$$

in order to guarantee that the requested delay time has not already passed. In other words, we must complete each iteration in under $charsep$ time units.

The fourth component generates a $\mathbf{read}$ statement.

$$
\begin{array}{lll}
(11) & \sqsubseteq & \text{“by strengthen effect; weaken assumption; \textbf{D2}”} \\
 & & \mathbf{read}(in, char)^{\mathsf{m}}
\end{array}
$$

Weakening the assumption to get the specification in the right form to apply definition **D2** introduces the following proof obligation

$$\mathsf{z} \dot{+} \mathsf{l} \dot{+} \mathsf{m} \quad \subseteq \quad 0 \mathinner{..} chardef \ ,$$

in order to guarantee that the incoming character stream is accessed at the right time. Recall the $\mathsf{z}$ represents how much the value of $next$ may be beyond the leading edge of each character, $\mathsf{l}$ is the overrun of the preceding **delay-until** statement, and $\mathsf{m}$ is the execution time of this statement. Also strengthening the effect appropriately requires

$$\mathsf{z} \dot{+} \mathsf{l} \dot{+} \mathsf{m} \quad \subseteq \quad \mathsf{z} \ ,$$

so that the invariant can be re-established in time.

The final requirement, $UpdateMsg$, need not be dependent on the absolute time, given the information provided by $StartLoop$, so we re-express it as a 'relative' time specification.

(8) $\sqsubseteq$ "by strengthen effect; weaken assumption"

$$\star Out\colon \left[\left[\, \tau : \mathbb{A};\ In;\ Out \,\right],\ \left[\, AddChar \mid \tau - \tau_0 \in \mathsf{y} \,\right]\right] \tag{12}$$

The assumption has predicate 'true' and the effect is defined as follows.

*AddChar*

$\tau_0, \tau : \mathbb{A};\ In;\ Out$

---

$stable(\{next\}, \tau_0 \mathinner{..} \tau)$

$char(\tau_0) \in PrintChars \Rightarrow msg(\tau) = msg(\tau_0) \frown \langle char(\tau_0) \rangle$

$char(\tau_0) \notin PrintChars \Rightarrow msg(\tau) = msg(\tau_0)$

This can then be easily refined to a choice, the second alternative of which has no work to do and can be immediately replaced by **skip**.

(12) $\sqsubseteq$ "by weaken assumption; **S2b**; **D5**"

**if**

$$char \in PrintChars \rightarrow^{\mathsf{f}} \star Out\colon \left[\left[\, \tau : \mathbb{A};\ In;\ Out \mid char(\tau) \in PrintChars \,\right],\ \left[\, AddChar \mid \tau - \tau_0 \in \mathsf{g} \,\right]\right]^{\mathsf{h}} \tag{13}$$

[]

$$char \notin PrintChars \rightarrow^{\mathsf{i}} \mathbf{skip}^{\ \mathsf{j}}$$

**fi**

The use of law **S2b** introduces two new timing obligations,

$$\mathsf{f} \dot{+} \mathsf{g} \dot{+} \mathsf{h} \quad \subseteq \quad \mathsf{y}$$
$$\mathsf{i} \dot{+} \mathsf{j} \quad \subseteq \quad \mathsf{y} \ .$$

Finally the first alternative can be immediately refined to an assignment.

(13) $\sqsubseteq$ "by strengthen effect; weaken assumption; **D1**"

$$msg :=^{\mathsf{g}} msg \frown \langle char \rangle$$

We can now remove redundant constant $T$, turning the $docon$ loop into a standard **do** loop. Putting all the above

steps together then yields the following time-annotated program,

$$
\begin{array}{l}
\text{-- `a' time units allowed to reach this point from } start \\
\mathbf{get\text{-}time}(next)^{\mathsf{b}}\mathbf{;} \\
\text{-- `z' time units allowed to reach this point from } start \\
\mathbf{read}(in, char)^{\mathsf{c}}\mathbf{;} \\
\mathbf{do} \text{ -- `x' time units allowed to reach guard from each `edge'} \\
\quad char \neq ETX \rightarrow^{\mathsf{d}} \\
\qquad \mathbf{if} \text{ -- this compound statement takes `y' time units} \\
\qquad\quad char \in PrintChars \rightarrow^{\mathsf{f}} msg :=^{\mathsf{g}} msg \frown \langle char \rangle^{\ \mathsf{h}} \\
\qquad [] \\
\qquad\quad char \notin PrintChars \rightarrow^{\mathsf{i}} \mathbf{skip}^{\ \mathsf{j}} \\
\qquad \mathbf{fi;} \\
\qquad next :=^{\mathsf{k}} next + charsep\mathbf{;} \\
\qquad \mathbf{delay\text{-}until}(next)^{\mathsf{l}}\mathbf{;} \\
\qquad \mathbf{read}(in, char)^{\mathsf{m}} \\
\mathbf{od}^{\mathsf{e}}
\end{array}
$$

with the following timing constraints to be discharged. Firstly **read** statements must occur at the right absolute time:

$$
\begin{array}{rcl}
\mathsf{a} \dot{+} \mathsf{b} \dot{+} \mathsf{c} & \subseteq & \mathsf{x} \\
\mathsf{a} \dot{+} \mathsf{b} \dot{+} \mathsf{l} \dot{+} \mathsf{m} & \subseteq & \mathsf{x} \\
\mathsf{x} & \subseteq & 0 \mathinner{..} chardef\ .
\end{array}
$$

Also, regardless of the path taken by the **if** statement,

$$
\begin{array}{rcl}
\mathsf{f} \dot{+} \mathsf{g} \dot{+} \mathsf{h} & \subseteq & \mathsf{y} \\
\mathsf{i} \dot{+} \mathsf{j} & \subseteq & \mathsf{y}\ ,
\end{array}
$$

each iteration must occur quickly enough to finish before the next character arrives (including the first),

$$\mathsf{x} \dot{+} \mathsf{d} \dot{+} \mathsf{y} \dot{+} \mathsf{k} \subseteq 0 \mathinner{..} charsep\ ,$$

and each iteration must advance time

$$0 \notin \mathsf{d}\ .$$

Lastly, we must be able to exit the loop quickly enough to satisfy the overall $GetMsg$ requirement when the $ETX$ character is encountered:

$$\mathsf{x} \dot{+} \mathsf{e} \subseteq 0 \mathinner{..} charsep\ .$$

# 6 Discussion

In the Quartz methodology [3] real-time system development is considered complete only when it has been shown that the actual execution time overheads of each language construct satisfy the timing annotations. Observe that all of the timing overheads appearing in the final code in Section 5 above, denoted b to m inclusive, could be determined by a suitably instrumented analysis tool. The outstanding timing obligations, which merely involve summing these times, can then be discharged easily. We are pursuing this goal in two ways.

1. For non-safety-critical applications timing constraints could be translated into assertions to be checked by a real-time simulator such as the *Nana* [15] toolkit.

2. Where timing correctness must be absolutely guaranteed a verified compilation strategy can be used to formally discharge the timing constraints against the generated object code and a formal model of the target execution environment [1, ch.7][3].

# 7 Related work

As explained in Section 2, the Quartz project aims to improve the timed refinement calculus by extending it into the realm of sequential code refinement. Also, although strongly inspired by TAM, we believe the Quartz approach has a greater chance of industrial acceptance than TAM because it is closer to the already-familiar refinement calculus.

A number of other major real-time projects have goals almost identical to Quartz, especially the safemos [1] and *ProCoS* [9] projects, and Hooman's development method [8].

In particular the safemos project independently developed a time-annotated sequential programming language SAFE whose syntax and definitions are very similar to those in Section 3.5 above [1, ch.6]. It too has sequential programming language constructs augmented with allowable implementation overheads [1, p.116]. Its semantics goes further than ours by also considering expression evaluation in detail [1, p.124]. Automated support for the language using the HOL theorem prover is available [1, p.130].

However SAFE does not have a **delay-until** statement, a necessary feature for implementing periodic real-time tasks. More significantly, though, the safemos project is limited to sequential programs only [1, p.37]. By starting from a refinement model that already includes parallelism, Quartz offers a more powerful method.

Furthermore, Hooman and the safemos and *ProCoS* projects target occam-based programming language code. Although elegant, occam has not been embraced by industry. To gain user acceptance Quartz aims to adopt already-popular methods where feasible. Quartz uses Z-based notations for specification, adheres to the standard refinement calculus as far as possible, and targets the Ada 95 programming language.

# 8 Conclusion

We have shown how the refinement calculus can be applied to development of real-time programs. This was done by adding the notion of timed variable traces from the 'timed refinement calculus' and a special time variable inspired by the 'Temporal Agent Model'. Unusually, the resulting calculus models code execution by successively constraining the trace variables, rather than updating them; only the time variable is destructively updated. Moreover, the specifications for individual sequential components are infeasible until their surrounding context is considered.

## Acknowledgements

We wish to thank Ian Hayes and Brendan Mahony for suggesting several improvements to this work, and Peter Kearney and Phil Maker for their help with the example. The Quartz project is funded by the Information Technology Division of the Australian Defence Science and Technology Organisation.

# References

[1] J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.

[2] C. Fidge. Adding real time to formal program development. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 618–638. Springer-Verlag, 1994.

[3] C. Fidge, P. Kearney, and M. Utting. Quartz: An integrated formal development method for real-time software. *IEEE Software*, 1996. To appear (also available as SVRC TR 94-26).

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

[5] N. Gehani and K. Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *The Journal of Real-Time Systems*, 3:377–405, 1991.

[6] P. Gopinath, T. Bihari, and R. Gupta. Compiler support for object-oriented real-time software. *IEEE Software*, 9(5):45–50, September 1992.

[7] E.C.R. Hehner. Termination is timing. In J.L.A. van de Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, June 1989.

[8] J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.

[9] H. Jifeng. *Provably Correct Systems*. McGraw-Hill, 1995.

[10] K.B. Kenny and K-J. Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[11] E. Kligerman and A.D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[12] B. Mahony. Networks of predicate transformers. Technical Report TR 95-5, Software Verification Research Centre, February 1995.

[13] B. Mahony and I. Hayes. A case study in timed refinement: A central heater. In J.M. Morris and R.C. Shaw, editors, *Fourth Refinement Workshop*, pages 138–149. Springer-Verlag, 1991.

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

[15] P.J. Maker. Nana—improved support for assertions and logging in C and C++. Technical Report TR-7-95, Northern Territory University, School of Information Technology, September 1995.

[16] C. Millerchip, B. Mahony, and I. Hayes. The generic problem competition: A whole system specification of the boiler system. In D. Del Bel Belluz and H.C. Ratz, editors, *Software Safety: Everybody's Business—Proc. 1993 International Workshop on Design and Review of Software Controlled Safety-Related Systems*, 1994.

[17] C. Morgan. Data refinement by miracles. *Information Processing Letters*, 25(5), January 1988.

[18] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[19] C. Morgan and T. Vickers. *On the Refinement Calculus*. Springer-Verlag, 1994.

[20] C.Y. Park and A.C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proc. IEEE Real-Time Systems Symposium*, pages 72–81, Florida, December 1990.

[21] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.

[22] D. Scholefield and H. Zedan. TAM: A formal framework for the development of distributed real-time systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 411–428. Springer-Verlag, 1992.

[23] D. Scholefield and H. Zedan. A standard for finite TAM. Technical Report YCS 206, Department of Computer Science, University of York, 1993.

[24] D. Scholefield, H. Zedan, and H. Jifeng. A specification-oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 131:219–241, 1994.

[25] A.D. Stoyenko and W.A. Halang. Extending Pearl for industrial real-time applications. *IEEE Software*, 10(4):65–74, July 1993.

[26] J.B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.