

Computer-Aided Development of a Real-Time Program

Luke Wildman¹, Colin Fidge¹, David Carrington^{1,2}

¹ Software Verification Research Centre, and

² Department of Computer Science and Electrical Engineering,
The University of Queensland, Queensland 4072, Australia.
e-mail: {luke,cjf,davec}@it.uq.edu.au

Abstract. The refinement calculus is a well-established theory for formal development of imperative program code and is supported by a number of automated tools. Via a detailed case study, this article shows how refinement theory and tool support can be extended for a program with real-time constraints. The approach adapts a timed variant of the refinement calculus and makes corresponding enhancements to a theorem-prover based refinement tool.

Key words: software engineering — real-time programming — formal methods — refinement — verification — tool support

1 Introduction

The refinement calculus is a formalism for systematically deriving programs from their specifications [16,2]. To support this process, it uses a ‘wide-spectrum’ modelling language that allows specifications and executable code fragments to coexist, and defines numerous verified refinement rules that translate requirements to code. Refinement theory has now matured to the point where a number of automated

tools are available to assist in its application [4,22,25]. However, the theory and these tools are limited to satisfaction of functional requirements only, and do not consider desired timing properties.

Motivated by a (necessarily small) case study, this article illustrates the capabilities of a new tool for refinement of programs with hard real-time constraints. To support such programs, extensions to current refinement methods are needed in several areas.

1. The modelling language must be extended to support an explicit representation of the current time [23], and to use a trace-oriented semantics that allows the timing of the system’s interactions with its environment to be described [15].
2. The semantics of the target programming language must be extended to represent the timing overheads of implementing each language construct [8], and new language annotations are needed for explicitly stating timing properties [7].
3. New refinement rules that transform real-time specifications into time-annotated programs must be devised and formally verified [9,11].
4. Tool support for this new theory must be developed. In practice, this may involve extending and revising the formalism [27].

Previous publications have described the first three points in depth. Here we concentrate on the last of these requirements, a practical instantiation of new real-time refinement theory in an automated support tool.

Our starting point is the Program Refinement Tool (PRT), a theorem-prover based tool that helps the programmer apply refinement rules and discharge any associated side conditions [5]. Our goal is to extend PRT with real-time refinement concepts based on those devised by Hayes et al. [9, 11, 23].

2 Related work

As noted above, conventional methods for refining specifications to program code, and their support tools, do not handle timing requirements. There are, however, some experimental approaches with aims similar to ours.

The Temporal Agent Model (TAM) is a refinement calculus that allows real-time requirements to be translated to program-like designs [20]. It uses a trace semantics for representing the history of interactions between real-time system components, and was a significant influence on the formalism used in this article [23].

The Provably Correct Systems (ProCoS) project also devised a refinement procedure that starts from the trace-based Duration Calculus and targets occam-like programming language code [19, 18]. The ProCoS methodology is fundamentally different from ours, however, because it involves translating from a Duration Calculus specification to the regular expression-based SL language, and from there to the occam-like PL language. Our formalism seeks to undertake refinements within a single wide-spectrum model, rather than translating between languages.

Hooman has also devised a real-time specification and reasoning notation and has shown how it can be used to undertake program development [12]. Despite some similarities,

Hooman’s formalism differs in appearance from ours because it is based on Hoare triples, rather than the predicate-transformer semantics normally used in refinement calculi.

Most importantly, however, none of these methods offer significant tool support (although there is a prototype theorem-prover implementation of the Duration Calculus’ logic [21]).

In this article we confine ourselves to the development of sequential program code. Although there are real-time specification and refinement methods devoted to the development of concurrent systems using, for instance, the Temporal Logic of Actions [1], timed action systems [26], timed predicate transformers [13], or timed process algebras [6], these methods typically work at the level of overall system design, whereas our interest in this article is refinement to imperative code within a process.

3 Background

In this section we review the wide-spectrum language supported in the refinement tool and introduce the motivational case study.

3.1 Specifying real-time requirements

The starting point for program refinement is a requirement expressed in a wide-spectrum modelling language [16, 2]. Normally this language includes imperative programming statements for assignment ($:=$), sequential composition ($;$), conditional choice ($\mathbf{if} \cdot \cdot \mathbf{fi}$), iteration ($\mathbf{do} \cdot \cdot \mathbf{od}$), and scoped declarations ($\mathbf{var} \cdot \cdot \cdot$). To these are added two non-executable statements. An *assumption*, $\{A\}$, states that predicate A is expected to be true at this point. A *specification statement*, $\tilde{x}: [R]$, specifies that we must make predicate R true, by changing only those variables listed in the *frame* \tilde{x} . (Elsewhere [16], the sequence consisting of an assumption followed by a specification, ‘ $\{A\}; \tilde{x}: [R]$ ’, is

written as a single statement, ‘ $\tilde{x}: [A, R]$ ’, but we do not need this notation here.)

When interpreted in a real-time application, this modelling language is assumed to support two significant new features [23, 9].

- There is a special variable τ , of type *Time*, used to denote the current time. An assumption $\{A\}$ may use this variable in its predicate. Assumptions do not consume time. However, a specification statement $\tilde{x}: [R]$ implicitly allows time τ to advance. Following refinement calculus convention [16], a specification predicate may use τ_0 to denote the time the statement begins, and τ to denote its finishing time. (See Appendix A.)
- All system variables are represented as *timed traces*, i.e., functions over time. For instance, a real-valued variable a is modelled as a total function of type $Time \rightarrow \mathbb{R}$, and its value at some time t can be accessed through appropriate indexing as $a(t)$. Furthermore, the timed traces used by a specification are classified into three groups: input (controlled by this system’s environment), output (sent from this system to its environment), and local (used only within this system).

3.2 Case study: Divider specification

As an example, we consider a case study loosely inspired by that of Scholefield et al. [20]. The requirement is to sample two inputs a and b and produce as output c the result of dividing the first by the second. However, whereas Scholefield et al. merely introduced a single timing deadline, we make the problem more challenging by requiring that the inputs are sampled, and the output displayed, within specific windows of time, as shown in Figure 1. Both inputs a and b must be sampled between times t and u , during which they are expected not to change their values. The output c must be displayed for at least the interval from times v to w . To make a so-

lution feasible, the whole system is expected to start no later than time s .

To specify this requirement formally, we represent inputs a and b and output c as timed traces of type $Time \rightarrow \mathbb{R}$. (Since we are using total functions here, the programmer would be obliged to reserve a range value to denote ‘undefined’ if a partial function was desired.) The top-level specification is shown in Figure 2. Let $\ell \dots n$ be the set of all times between ℓ and n , inclusive. That is, $\ell \dots n = \{m : Time \mid \ell \leq m \leq n\}$. Let $f(\downarrow S)$ be the image of set S through function f . That is, $f(\downarrow S) = \{f(s) \mid s \in S\}$.

The specification in Figure 2 begins with assumption (1) stating two properties that the program may rely on. The first is that the time τ when the system starts executing is no later than s . The second is that inputs a and b are expected to be unchanging or ‘stable’ between times t and u . Stability of a timed-trace variable x over a set of times T is defined by stating that the trace function has only one range value over all times in T . Let $\#S$ denote the size of set S .

$$\text{stable}(x, T) \stackrel{\text{def}}{=} \#(x(\downarrow T)) = 1$$

This definition is generalised to accept a set of trace variables as its first argument in the obvious way.

The assumption is followed by specification (2) which states that output c is to be updated in such a way that its value between times v and w equals the result of dividing input a ’s value at time t by input b ’s value at time t , provided that the value of b was not zero. If b at time t equals zero, the output is left unspecified.

3.3 A real-time programming language

As usual in refinement, our target programming language is a distinguished subset of the modelling language. The definitions in Table 1 show that each programming language statement on the left is an abbreviation for a modelling language construct on the right.

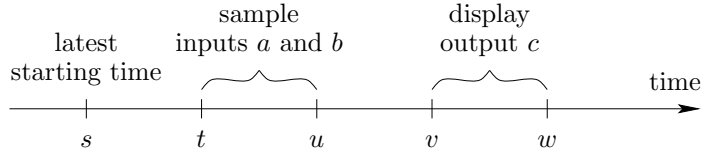


Fig. 1. Required timing for the ‘divider’ program.

$$\{\tau \leq s \wedge \text{stable}(\{a, b\}, t \dots u)\}; \quad (1)$$

$$c: [b(t) \neq 0 \Rightarrow c(v \dots w) = \{a(t)/b(t)\}] \quad (2)$$

Fig. 2. Formal specification of ‘divider’ program.

Let S be a statement, T a type, A a predicate, v a local variable, u an input variable, w an output variable, E an expression, B a boolean-valued expression, and D a time-valued expression. Expressions B , E and D may refer to local variables only. Since variables and expressions on the left-hand side appear in the final program code, they do not explicitly index timed traces, and they may not refer to the time variable τ . The only exception is assumption predicate A [7]. Where variables and expressions appear without indexing in the right-hand column of Table 1, they are implicitly indexed by time τ . See Appendix A for the formal semantics of real-time specification statements.

An `idle` statement changes no state variables but it allows an arbitrary amount of time to pass. An `assert` statement is not executable, but allows the programmer to document conditions that are expected to be true, including properties of the current time τ , and trace properties of variables. In the definition of an assignment statement, the final value of v is the value of expression E evaluated at the starting time of the statement τ_0 . Since sequential composition consumes no time, its definition is the standard (untimed) one. The semantics of an `if` statement incorporates explicit `idle` statements to model the time taken to evaluate condition B and branch to and from the appropriate alternative [8]. The same approach is used for a `while` statement,

in order to model expression evaluation and branching overheads associated with loop entry, exit and iteration. When a `declare` block is used to introduce a local variable v of type T , it is semantically interpreted to be a timed-trace variable of type $Time \rightarrow T$, and `idle` delays are introduced to model potential run-time allocation and deallocation of memory space. A `read` statement finishes with local variable v equal to some value of input variable u sampled between times τ_0 and τ . A `write` statement finishes with output variable w equal to the value of expression E . Unlike the assignment statement, expression E may be evaluated at any time during execution of the `write` because w cannot be free in E , so the expression will remain unchanged. A `delay until` statement changes no state variables and finishes when the time τ is no earlier than the specified time D . The complementary `deadline` statement requires the time τ to be no later than the specified time D . This statement is not executable (it takes no time) but serves to document timing constraints that must be verified through static analysis of the compiled code [7, 8].

3.4 Case study: Target program

Returning to our example, the code fragment in Figure 3 shows one way in which the requirement in Figure 2 could be satisfied using these language

Statement	Definition
idle	$:[\text{true}]$
assert A	$\{A\}$
$v := E$	$v: [v = E(\tau_0)]$
$S_1 ; S_2$	$S_2 ; S_2$
if B then S_1 else S_2 end if	if $B \rightarrow \text{idle} ; S_1 ; \text{idle}$ $\parallel \neg B \rightarrow \text{idle} ; S_2 ; \text{idle}$ fi
while B loop S end loop	do $B \rightarrow \text{idle} ; S ; \text{idle}$ od ; idle
declare $v : T$ begin S end	var $v : \text{Time} \rightarrow T \bullet$ $(\text{idle} ; S ; \text{idle})$
read (u, v)	$v: [v \in u(\tau_0 \dots \tau)]$
write (w, E)	$w: [w = E]$
delay until D	$:[D \leq \tau]$
deadline D	$:[\tau_0 = \tau \wedge \tau \leq D]$

Table 1. Semantic definitions for programming language statements.

constructs. We assume that inputs a and b and output c have been declared in the surrounding scope. Functionally, the program is straightforward. It declares two local variables m and n , reads values into them from inputs a and b , and sets output c equal to the result of dividing m by n . If n equals zero, the programmer has chosen to set c equal to the special value ‘ ∞ ’.

Less familiar though are the various time-specific constructs. Statements (3) and (4) are assertions that document the programmer’s belief that this code fragment will begin executing before time s and that inputs a and b will not change between times t and u . Otherwise it may be impossible to satisfy the timing requirements. Statement (6) is a **delay until** that ensures that the inputs are not read before time t , and statement (9) is a **deadline** that documents the need to complete sampling the inputs no later than time u , while they are known to be stable. The update to output c is followed by statement (13) which is another **deadline** to ensure that the output is produced by time v . Statement (14) is a **delay until**

that stops the program from progressing until time w . This ensures that c is not overwritten too soon by any subsequent code. (We assume that this program fragment ‘owns’ output variable c and that no other process may write to it.) These timing statements are sufficient to allow a static analysis tool to identify and verify the timing constraints on the final executable code generated by the compiler [8, 7].

4 The real-time refinement tool

The challenge now is to translate the requirement in Figure 2 to the program in Figure 3 using the theory of real-time refinement [11, 9]. In the general case, achieving such a translation accurately and efficiently requires appropriate tool support. In this section we show how this particular refinement is performed and use it to explain the capabilities of our real-time refinement tool.

As noted above, we chose to extend the Program Refinement Tool (PRT) [5] to incorporate the new concepts and rules associated with real-time refine-

```

assert  $\tau \leq s$  ; -- assumed starting time (3)
assert  $\text{stable}(\{a, b\}, t \dots u)$  ; -- assumed input behaviour (4)
declare (5)
   $m, n : \mathbb{R}$ 
begin
  delay until  $t$  ; -- wait until inputs are stable (6)
  read( $a, m$ ) ; (7)
  read( $b, n$ ) ; (8)
  deadline  $u$  ; -- finish reading inputs while stable (9)
  if  $n \neq 0$  then (10)
    write( $c, m/n$ ) (11)
  else
    write( $c, \infty$ ) (12)
  end if ;
  deadline  $v$  ; -- produce output early enough (13)
  delay until  $w$  -- leave output unchanged (14)
end

```

Fig. 3. The final ‘divider’ program.

ment. PRT supports the goal-directed *program window inference* [17] style of refinement. It provides a multi-window user interface, programmer-definable tactics, and the ability to record and replay proof scripts. PRT is implemented in a theorem-proving environment so that the same conceptual model is used for both refinement steps and proof of side conditions. As explained below, making PRT suitable for real-time applications required extensions to its support for ‘contexts’ and ‘opening rules’ as well as implementing the real-time refinement rules themselves.

4.1 Real-time contexts

PRT maintains a *context* which consists of three types of hypotheses that can be used when discharging any proof obligations introduced by refinement steps.

- The **lval** context maintains knowledge about the declaration and scope of program variables and constants.

- The **pre** context maintains knowledge about the initial state of the current program step or transition.
- The **inv** context maintains knowledge known to hold in every program state, e.g., types of variables.

To extend PRT for real-time refinements, additional contextual information was introduced to distinguish the three classes of timed-trace variables (**input**, **local** and **output**), any ‘normal’ (non timed-trace) variables (**var**), and time-invariant constants (**con**). Each of these new notations abbreviates a number of basic hypotheses. For instance, stating ‘**output** $c : \mathbb{R}$ ’ in the context is a shorthand for ‘**lval** $c : \text{output}$ ’, which tells us that c is an output variable, and ‘**inv** $c : \text{Time} \rightarrow \mathbb{R}$ ’ which defines the underlying type of c [27]. Similarly for **input** and **local**.

To apply the real-time refinement tool to our case study, the programmer begins by stating the following initial context of known facts, which represent the implicit environment in which the specification in Figure 2 is meant to be interpreted. They include the declarations of external inputs and outputs and

the relationship between the absolute-time constants.

input $a, b : \mathbb{R}$ (i)

output $c : \mathbb{R}$

con $s, t, u, v, w : Time$ (ii)

inv $s \leq t < u \leq v < w$ (iii)

Type *Time* is known to the tool and is used in hypothesis (ii) to declare the significant time constants. The assumed relationship between these constants is introduced as invariant hypothesis (iii).

The tool has also been extended to allow time variable τ to appear in real-time specifications, in both plain and zero-subscripted form. In this case study it is the only variable that is *not* modelled as a timed trace [23].

con $\tau_0 : Time$ (iv)

var $\tau : Time$ (v)

These two facts apply to every statement, and define how the statement may refer to the time at which it starts, τ_0 , and finishes, τ . (The finishing time τ of one statement is the same as the starting time τ_0 of its successor.) The current statement is free to choose how long it takes, so hypothesis (v) declares finishing time τ to be a variable. However, the time at which the current statement is invoked is not under its control, so hypothesis (iv) treats starting time τ_0 as a given constant.

Normally the programmer does not want to be distracted by explicitly indexing all occurrences of timed-trace variables, and such indexing never occurs in the final programming language code. To make indexing implicit, the tool therefore supports a form of the ‘@’ substitution operator [14, 11]. For some expression E containing references to a timed-trace variable v , let $E@ \tau$ denote expression E with each unindexed occurrence of ‘ v ’ replaced by ‘ $v(\tau)$ ’. The standard refinement calculus usually introduces zero-subscripted variables to denote initial values [16]. Therefore, also let $E@(\tau_0, \tau)$ additionally replace any unindexed, zero-subscripted timed-trace variable ‘ v_0 ’ with ‘ $v(\tau_0)$ ’. Supporting these helpful abbreviations in the

real-time refinement tool makes its user interface much closer to traditional refinement calculus notation.

4.2 Real-time window opening rules

In program window inference, *opening rules* are used to change the current *focus* of interest by initiating separate development of a particular specification component [17]. In doing so, these rules also make corresponding modifications to the current context. Figure 4 shows some of the opening rules implemented by the real-time refinement tool. Let a context be represented by three sets of hypotheses, **pre** P , **lval** L and **inv** I . Let ‘ \sqsubseteq ’ be the refinement relation, and a boxed statement be the focus of interest. Let $\text{sp}(S, P)$ be the strongest postcondition derivable by execution of statement S starting with precondition P [27]. Let $\text{hide}(x, H)$ be an operation which deletes any hypotheses in list H that contain free occurrences of identifier x . Let Γ be the set of all local and output variables currently in scope.

Each opening rule in Figure 4 is of the following form.

$$\frac{C_1 \models T_1}{C_2 \models T_2}$$

Such a rule states that if a ‘minor’ transformation T_1 is possible in context C_1 , then the corresponding ‘major’ transformation T_2 is possible in context C_2 . Notice that context C_1 is always some extension of context C_2 and that transformation T_1 is applied to the ‘focus’ component of transformation T_2 . Such a rule is used when the programmer wishes to undertake overall transformation T_2 by performing component transformation T_1 on the current focus. The tool automatically establishes the proof environment in which to perform the minor transformation T_1 and, when this has been successfully completed, instantiates the result to achieve the major step T_2 .

Returning to our example, consider the specification in Figure 2. By the def-

Opening Rule A (Focus after assumption)

$$\frac{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L; \text{ pre } A @ \tau \\ \models \boxed{S} \sqsubseteq S' \end{array}}{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L \\ \models \{A\}; \boxed{S} \sqsubseteq \{A\}; S' \end{array}}$$

Opening Rule B (Focus on var block body) Let v' be a fresh variable.

$$\frac{\begin{array}{l} \text{pre } \text{sp}(\text{idle}, P[v'/v]); \text{ inv } I[v'/v]; \text{ lval } L[v'/v]; \text{ local } v : T \\ \models \boxed{S} \sqsubseteq S' \end{array}}{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L \\ \models \text{declare } v : T \text{ begin } \boxed{S} \text{ end } \sqsubseteq \text{declare } v : T \text{ begin } S' \text{ end} \end{array}}$$

Opening Rule C (Focus on second component)

$$\frac{\begin{array}{l} \text{pre } \text{sp}(S_1, P); \text{ inv } I; \text{ lval } L \\ \models \boxed{S_2} \sqsubseteq S'_2 \end{array}}{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L \\ \models S_1; \boxed{S_2} \sqsubseteq S_1; S'_2 \end{array}}$$

Opening Rule D (Strengthen effect)

$$\frac{\begin{array}{l} \text{con } \tau_0 : \text{Time}; \text{ pre } \text{hide}(\tau_0, P)[\tau_0/\tau]; \text{ pre } \tau_0 \leq \tau; \\ \text{pre } \text{stable}(F \setminus \tilde{x}, \tau_0 \dots \tau); \text{ inv } \text{hide}(\tau_0, I); \text{ lval } \text{hide}(\tau_0, L) \\ \models \boxed{R @ (\tau_0, \tau)} \Leftarrow R' @ (\tau_0, \tau) \end{array}}{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L \\ \models \tilde{x}: \boxed{R} \sqsubseteq \tilde{x}: R' \end{array}}$$

Opening Rule E (Focus on first alternative)

$$\frac{\begin{array}{l} \text{pre } B @ \tau; \text{ pre } \text{sp}(\text{idle}, P); \text{ inv } I; \text{ lval } L \\ \models \boxed{S_1} \sqsubseteq S'_1 \end{array}}{\begin{array}{l} \text{pre } P; \text{ inv } I; \text{ lval } L \\ \models \text{if } B \text{ then } \boxed{S_1} \text{ else } S_2 \text{ end if } \sqsubseteq \text{if } B \text{ then } \boxed{S'_1} \text{ else } S_2 \text{ end if} \end{array}}$$

Fig. 4. Real-time window opening rules.

initiation in Table 1, assumption (1) does not require further development since it is already equivalent to the two **assert** statements (3) and (4) in Figure 3. However, to transform specification (2) we use the refinement tool's point-and-click interface to highlight and select this statement. The tool automatically determines that Opening Rule A from Figure 4 matches this situation and applies it to the specification.

When focussing on a statement S following an assumption $\{A\}$, Opening Rule A tells us that the new focus in-

herits the fact that predicate A is initially true as part of its context. This knowledge can then be used to help refine statement S to S' . The top line in Opening Rule A requires that it is possible to refine S to S' in a context consisting of previous hypotheses P , I and L , plus a new **pre** hypothesis $A @ \tau$. The $@$ operator is used when predicate A appears outside of the assumption braces to allow for the possibility that it included unindexed references to timed-trace variables when it appeared in assumption $\{A\}$.

Refinement Rule 1 (Introduce var block)

$$\begin{array}{l}
 \mathbf{lval} \ v \notin \tilde{x} \\
 \mathbf{lval} \ v \text{ nfi } R \\
 \mathbf{inv} \ \text{non-empty}(T) \\
 \mathbf{inv} \ \text{pre-idle-invariant}(R) \\
 \mathbf{inv} \ \text{post-idle-invariant}(R) \\
 \hline
 \tilde{x}: [R] \sqsubseteq \mathbf{declare} \ v : T \mathbf{begin} \ v, \tilde{x}: [R] \mathbf{end}
 \end{array}$$
Refinement Rule 2 (Introduce simple sequence) Let Q be a predicate.

$$\begin{array}{l}
 \mathbf{lval} \ \text{no-subscripts}(Q) \\
 \mathbf{lval} \ \text{no-subscripts}(R) \\
 \hline
 \tilde{x}: [R] \sqsubseteq \tilde{x}: [Q] ; \tilde{x}: [R]
 \end{array}$$
Refinement Rule 3 (Separate deadline)

$$\begin{array}{l}
 \mathbf{inv} \ D @ \tau \in \text{Time} \\
 \mathbf{lval} \ \text{no-subscripts}(D) \\
 \hline
 \tilde{x}: [R \wedge \tau \leq D] \sqsubseteq \tilde{x}: [R] ; \mathbf{deadline} \ D
 \end{array}$$
Refinement Rule 4 (Introduce conditional statement)

$$\begin{array}{l}
 \mathbf{inv} \ \text{idle-stable}(B) \\
 \mathbf{inv} \ \text{pre-idle-invariant}(R) \\
 \mathbf{inv} \ \text{post-idle-invariant}(R) \\
 \hline
 \tilde{x}: [R] \sqsubseteq \mathbf{if} \ B \ \mathbf{then} \ \tilde{x}: [R] \ \mathbf{else} \ \tilde{x}: [R] \ \mathbf{end} \ \mathbf{if}
 \end{array}$$
Fig. 5. Real-time refinement rules.

When we focus on specification (2) and apply this rule, the real-time refinement tool extends the context in Section 4.1 to include the following new hypotheses, derived from the two conjuncts in assumption (1). Subsequent refinement of specification (2) may now make use of this knowledge.

- (vi) $\mathbf{pre} \ \tau \leq s$
- (vii) $\mathbf{pre} \ \text{stable}(\{a, b\}, t \dots u)$

In this case, neither conjunct is changed by the @ operator appearing in the opening rule. In the first, neither special variable τ nor constant s is a timed trace, and in the second the timed traces a and b are both explicitly indexed by absolute times, as shown by the definition of ‘stable’ in Section 3.2.

The real-time refinement tool implements many such opening rules [27]. Most importantly, the tool automatically maintains the context as each rule

is applied, relieving the programmer of this tedious and error-prone task.

4.3 Real-time refinement rules

A formal *refinement rule* transforms the current focus to a new focus provided that any premises (side conditions) can be proven in the current context. Figure 5 shows some of the real-time refinement rules implemented by the tool. Predicate ‘ $v \text{ nfi } E$ ’ means that variable v does not occur free in expression E , and predicate ‘non-empty(T)’ means that type T has at least one element. Predicates ‘pre-idle-invariant’ and ‘post-idle-invariant’ check that their argument remains true even when it is preceded by, or followed by, respectively, an arbitrary idle delay [9,27]. Similarly, an ‘idle-stable’ expression is unaffected by preceding or succeeding idle delays. Predicate

‘no-subscripts’ checks that its argument expression contains no occurrences of zero-subscripted variables.

Each rule in Figure 5 is of the following form.

$$\frac{C}{S_1 \sqsubseteq S_2}$$

Such a rule states that statement S_1 can be transformed into statement S_2 , in a context satisfying the hypothesis list C .

Returning to our case study, once we have opened on specification (2), the tool offers the programmer a list of applicable refinement rules. By selecting Refinement Rule 1 (Figure 5) we can transform the specification to introduce a new local variable. Provided that we can prove the conditions above the line in this rule are all true, we may perform the refinement step shown below the line. The first three conditions to be checked are standard. The new variable must not already appear in the frame or the specification predicate, and its type must not be empty. The final two conditions are specific to the real-time refinement calculus. They are necessary because placing a specification statement inside a variable block may introduce time delays due to the implementation overheads of allocating and deallocating memory space for the local variable, so we must check that the specification predicate is still meaningful when preceded and succeeded by delays. The new variable is also added to the frame of the original specification statement, so that this statement may modify its value.

Applying Refinement Rule 1 (twice) to our current focus introduces the `declare` block appearing as statement (5) in Figure 3.

$$\begin{aligned} & \text{Specification (2)} \\ & \sqsubseteq \text{declare} \\ & \quad m, n : \mathbb{R} \\ & \text{begin} \\ & \quad m, n, c : [b(t) \neq 0 \Rightarrow \quad (15) \\ & \quad \quad c(v \dots w) = \\ & \quad \quad \quad \{a(t)/b(t)\}] \end{aligned}$$

`end`

In this case the refinement tool uses syntactic checks to automatically discharge the proof obligations associated with this step. Variables m and n do not appear anywhere in the original specification. Also, because all appearances of timed-trace variables in the specification predicate are explicitly indexed with absolute times, the predicate is inherently invariant with respect to delays.

The real-time refinement tool implements such refinement rules for each target language construct. To assist the programmer, the tool automatically determines what proof obligations must be satisfied for the rule to be applied, and the context in which this proof must be done. In many cases, such as the example above, these obligations are discharged automatically by in-built tactics. Otherwise, an undischarged proof obligation is recorded. This must be satisfied before the refinement step can be accepted. When the programmer decides to discharge the obligation, the tool creates an appropriate theorem-proving environment in which to undertake the proof.

In general, the annotations on the premises in such refinement rules indicate which classes of contextual information may be used by the tool to discharge them: **lval**, **pre** and **inv** hypotheses may be used to discharge a **pre** obligation; **lval** and **inv** hypotheses may be used to discharge an **inv** obligation; and **lval** hypotheses only may be used to discharge an **lval** obligation. Premises annotated by **lval** can be proven automatically.

4.4 Partitioning the program

In this section we continue the case study started above, illustrating the use of further real-time opening and refinement rules. Our overall strategy is to divide the requirement into two parts, the first to read the inputs, and the second to write the output.

The next step is to focus on the specification statement within the variable block. To do so, we use Opening Rule B from Figure 4 [27]. This rule says that focus S can be refined to statement S' within a variable block, provided that S can be refined to S' in a context where v is declared to be a local variable of type T , and the prestate is that obtained from P after an arbitrary idle delay. This delay accounts for the fact that, although we know P is true at the beginning of the variable block, some time may elapse before the statement S within the block is reached. An arbitrary name v' is substituted for v throughout the context to override any existing occurrences of v inherited from the surrounding scope.

When the tool is used to focus on specification (15) using this rule, the new context it creates includes the following new hypotheses. (Recall from Section 4.1 that ‘**local**’ abbreviates both **inv** and **lval** predicates.)

$$\mathbf{local} \ m, n : \mathbb{R} \quad (\text{viii})$$

$$\mathbf{pre} \ (\exists r : \mathit{Time} \bullet \quad (\text{ix})$$

$$r \leq s \wedge$$

$$r \leq \tau \wedge$$

$$\mathbf{stable}(c, r \dots \tau))$$

Hypothesis (ix) replaces hypothesis (vi), and is the result of calculating postcondition $\mathbf{sp}(\mathbf{idle}, \tau \leq s)$ [27]. It tells us that output variable c will remain stable from some point no later than time s up to the time when specification statement (15) is reached. However, in the absence of more information about how long memory allocation actually takes, this condition (correctly) tells us little about the absolute time at which statement (15) starts executing. (Remember that ‘ τ ’ in precondition hypothesis (ix) is actually the starting time τ_0 of statement (15), whereas ‘ r ’ is the time the **declare** block was reached.)

Interestingly, hypothesis (vii) remains part of the context following this step. This is because the expression ‘ $\mathbf{stable}(\{a, b\}, t \dots u)$ ’ does not rely on τ , and is thus unaffected by the idle

delay used in calculating the strongest postcondition. The expression refers to a property between *absolute* times t and u , and is thus unaffected by changes to the current time τ . This illustrates an interesting aspect of specifying and reasoning with timed traces: we can refer to variable values at *any* time, even in the past or the future, and properties tied to particular moments are unaffected by the passage of time. Indeed, hypothesis (vii) remains valid in all the steps below.

In this new context, we can select and apply Refinement Rule 2 from Figure 5 to start partitioning the specification into sequential components [27]. Refinement Rule 2 tells us that we can introduce a new sequential component before the current specification statement, provided that special variable τ_0 does not appear free in either predicate. In effect, this ensures that the predicate does not refer to the starting time τ_0 . Where predicates appear in a refinement rule outside their surrounding specification statements, they may not refer to the starting time of the statement.

The tool allows us to apply Refinement Rule 2 to divide the current focus, i.e., specification (15), into two parts. The first is expected to read inputs a and b into local variables m and n , and to finish no later than time u . The second must then produce the desired output c .

Specification (15)

$$\sqsubseteq \ m, n, c : [m = a(t) \wedge \quad (\text{16}) \\ n = b(t) \wedge \tau \leq u] ;$$

$$m, n, c : [b(t) \neq 0 \Rightarrow \quad (\text{17}) \\ c(v \dots w) = \{a(t)/b(t)\}]$$

Of course, the creative aspect of such a refinement step is to devise the new predicate for the first component. The tool prompts the programmer to enter the predicate, and automatically checks it for both syntactic correctness, and to ensure that it uses only variables known in the current context. (The tool does not check type correctness, although type consistency is enforced by the re-

finement rules.) In this case, variables a and b are known thanks to hypothesis (i), and variables m and n were declared in hypothesis (viii).

4.5 Reading the inputs

In this section we continue the case study by developing the requirement for reading the two input variables. This illustrates the introduction of real-time `delay until` and `deadline` statements.

The tool allows us to focus on specification (16) and apply Refinement Rule 2 again. Focussing on the first component of a sequential composition does not change the context, since the first component cannot gain knowledge from the second [27].

Specification (16)

$$\sqsubseteq : [t \leq \tau] ; \quad (18)$$

$$m, n : [m = a(t) \wedge n = b(t) \wedge \tau \leq u] \quad (19)$$

(For brevity, we have omitted trivial steps that remove variables from the frames of these two statements [11, Law 5].) By the definition in Table 1, specification (18) is equivalent to `delay until` statement (6) in Figure 3.

We now want to focus on the second component of this sequential composition. To support this, the tool implements Opening Rule C shown in Figure 4 [27]. This rule tells us that to refine the second component S_2 of a sequential composition, the refinement step must be valid in a context where the prestate is defined as the strongest postcondition derivable from executing the first component S_1 in the prestate P of the whole composition.

Focussing on specification (19) causes the tool to apply Opening Rule C and extend the current context with the following new hypothesis, among others.

$$\mathbf{pre} \ t \leq \tau \quad (\text{x})$$

It tells us that, by virtue of the preceding statement, we know that the current time τ is no earlier than t when

specification statement (19) is reached. (Recall that ‘ τ ’ in a **pre** context represents the *starting* time of the current statement—the ‘ τ_0 ’ notation appears only within specification statement predicates.)

We now want to use this knowledge to transform specification (19) so that it is closer to our target programming language definitions. This involves focussing not on the specification statement itself, but on the predicate within the statement. To support this, the tool implements Opening Rule D. This powerful rule tells us that we can focus on the predicate R inside a specification statement and transform it to predicate R' , provided that R' implies R in the extended context shown above the line. Both predicates R and R' must be explicitly annotated with the @ operator when they appear outside of a specification statement in case they make unindexed use of trace variables. Similarly, calculation of the new context is complicated because predicate R may contain references to τ_0 when it is within a specification statement, but this identifier is not meaningful when the predicate appears outside such a statement. Therefore, the context above the line explicitly introduces τ_0 as a *Time*-valued constant and hides existing references to this identifier, if any. Also, to allow meaningful interpretation of R outside of its surrounding statement, the new context introduces the expected relationship between τ_0 and τ , and the stability property for those local and output variables I that are not in the frame \tilde{x} . (See Appendix A.) Although this rule appears intimidating, the refinement tool applies all of these updates automatically. The programmer does not need to be aware of the details above to use the rule successfully.

Most importantly, Opening Rule D illustrates how the refinement tool is smoothly integrated into its theorem-proving environment. The relationship to be proven above the line is not refinement, but implication. The programmer thus works with the same concepts

when applying refinement transformations and verifying properties.

When the tool applies Opening Rule D to specification (19), it offers the opportunity to transform the predicate to some stronger requirement. Anticipating the need to transform the predicate so that **read** statements can be introduced in our case study, we choose ‘ $m \in a(\tau_0 \dots \tau) \wedge n \in b(\tau_0 \dots \tau) \wedge \tau \leq u$ ’, which introduces the need to prove the following property.

$$\begin{aligned} m &= a(t) \wedge n = b(t) \wedge \tau \leq u \\ \Leftarrow & (m \in a(\tau_0 \dots \tau) \wedge \\ & n \in b(\tau_0 \dots \tau) \wedge \tau \leq u) \end{aligned}$$

As per Opening Rule D, this is to be done in an expanded context that includes the following new hypothesis, among others.

$$\mathbf{pre} \ t \leq \tau_0 \quad (\text{xi})$$

Hypothesis (xi) was derived by the tool from the preceding hypothesis (x). Opening Rule D substitutes τ_0 for τ in any **pre** hypothesis P .

To prove this property, the programmer can use any of the hypotheses that the tool has maintained so far. In particular, note that hypothesis (vii) tells us that inputs a and b are unchanged from times t to u , that hypothesis (xi) tells us that the starting time τ_0 is no earlier than time t , and that the condition to be proven explicitly states that the finishing time τ will be no greater than u . With this knowledge, it is clear that if a and b are sampled at any times between τ_0 and τ , then the values read will equal $a(t)$ and $b(t)$, respectively. This is proven interactively, using conventional theorem proving techniques, by making use of one of the tool’s stability theorems which tells us that a sample drawn from any time within a stable interval yields the same value that the stable variable had at the beginning of the interval. Once we have done this, the tool allows us to complete the following refinement step.

$$\begin{aligned} & \text{Specification (19)} \\ \sqsubseteq & m, n: [m \in a(\tau_0 \dots \tau) \wedge \quad (20) \\ & n \in b(\tau_0 \dots \tau) \wedge \\ & \tau \leq u] \end{aligned}$$

The final conjunct in specification (20) is of a familiar form—it matches the definition of a **deadline** statement in Table 1. The refinement tool implements Refinement Rule 3 in Figure 5 for just such a situation [27]. This rule tells us that given a specification that must be completed by time D , the timing constraint can be separated out as a distinct **deadline** statement. However this can be done only if we can prove that expression D is of type *Time*, and that D does not refer to special variable τ_0 .

The tool thus allows us to perform the following refinement step that introduces **deadline** statement (9) to our target program in Figure 3. The trivial premises are both discharged automatically.

Specification (20)

$$\sqsubseteq m, n: [m \in a(\tau_0 \dots \tau) \wedge \quad (21) \\ n \in b(\tau_0 \dots \tau)];$$

deadline u

The two conjuncts remaining in specification (21) both match the definition of **read** statements in Table 1. It is straightforward from this point to apply a generalised ‘introduce sequence’ refinement step [27], similar to Refinement Rule 2 above, to partition specification (21) into two sequential specifications, perform some simple frame-contraction steps, and thus yield **read** statements (7) and (8) in Figure 3.

4.6 Producing the output

In this section we complete the case study by refining the requirement for writing the output variable. It illustrates introduction of a conditional statement, and further time-specific statements.

Firstly, we focus on specification (17), causing the tool to again apply Opening Rule C. To do this we must ‘back out’ of the tree of refinement and proof steps completed in Section 4.5. The tool allows the programmer to do this by ‘closing’ each completed step and automatically keeps track of the

context as this is done, effectively reverting it to the state represented by hypotheses (i) to (ix) above. However, by then focussing on specification (17), and stepping through the strongest postcondition of the code developed in Section 4.5, we cause the tool to introduce several new hypotheses to the context.

$$\begin{aligned} & \mathbf{pre} \tau \leq u \\ & \mathbf{pre} m(\tau) = a(t) \quad (\text{xii}) \\ & \mathbf{pre} n(\tau) = b(t) \quad (\text{xiii}) \end{aligned}$$

Refinement Rule 2 can be applied to divide the requirement even further. The new first component aims to complete the update to output c no later than time v . (Again, we omit trivial ‘contract frame’ steps.)

Specification (17)

$$\sqsubseteq c: [b(t) \neq 0 \Rightarrow \quad (22) \\ c = a(t)/b(t) \wedge \\ \tau \leq v] ;$$

$$c: [b(t) \neq 0 \Rightarrow \quad (23) \\ c(v \dots w) = \{a(t)/b(t)\}]$$

Focussing on specification (22) then allows us to apply Refinement Rule 3 as follows to produce **deadline** statement (13) in Figure 3.

Specification (22)

$$\sqsubseteq c: [b(t) \neq 0 \Rightarrow \quad (24) \\ c = a(t)/b(t)] ; \\ \mathbf{deadline} \ v$$

Focussing on the predicate in specification (24) causes the tool to apply Opening Rule D, and allows the programmer to strengthen the predicate in two ways. By taking advantage of hypotheses (xii) and (xiii), plus the fact that m and n are not in the frame of this statement and are thus stable, we can replace occurrences of ‘ $a(t)$ ’ with ‘ m ’ and ‘ $b(t)$ ’ with ‘ n ’. The required proof in this case is straightforward.

Specification (24)

$$\sqsubseteq c: [n \neq 0 \Rightarrow c = m/n] \quad (25)$$

We introduce Refinement Rule 4 in Figure 5 to allow refinement of any specification to a conditional statement.

This rule tells us that, regardless of the value of boolean condition B , the choice will satisfy the original requirement. The specification predicate R must be invariant with respect to preceding and succeeding idle delays, to allow for the time taken to enter and exit the **if** construct. Similarly, expression B must not be dependent on the time at which it is evaluated. (This is trivially true because we require that B refers to local variables only.)

Using the tool to apply Refinement Rule 4 to specification (25) yields **if** statement (10) in Figure 3.

Specification (25)

$$\begin{aligned} & \sqsubseteq \mathbf{if} \ n \neq 0 \ \mathbf{then} \\ & \quad c: [n \neq 0 \Rightarrow c = m/n] \quad (26) \\ & \quad \mathbf{else} \\ & \quad c: [n \neq 0 \Rightarrow c = m/n] \quad (27) \\ & \quad \mathbf{end} \ \mathbf{if} \end{aligned}$$

The premises are satisfied in this case because the boolean expression and specification predicate both refer to local variables only, and are not dependent on the absolute time at which they are evaluated. Thus their values will not change during any arbitrary idle delay. Again, the tool automatically discharges these obligations.

Opening Rule E in Figure 4 allows us to focus on the first component in a conditional statement. The context in which statement S_1 is to be refined allows for a potential idle delay due to evaluating the boolean expression. It is also extended with a new hypothesis telling us that expression B must have been true for this statement to have been selected. A symmetric rule allows us to focus on the second alternative with ‘ $\neg B$ ’ added to the context instead.

Thus, focussing on specification (26) using this rule introduces the following hypothesis to the context.

$$\mathbf{pre} \ n(\tau) \neq 0 \quad (\text{xiv})$$

Hypothesis (xiv) can be used to discharge the proof obligation associated

with applying Opening Rule D to undertake the following refinement.

$$\begin{array}{l} \text{Specification (26)} \\ \sqsubseteq c: [c = m/n] \end{array}$$

By definition, this gives us **write** statement (11) in Figure 3.

Backing out of this part of the refinement tree, and focussing on specification (27), replaces hypothesis (xiv) in the context with the following hypothesis.

$$\mathbf{pre} \ n(\tau) = 0 \quad (\text{xv})$$

Combining hypothesis (xv) with the predicate in specification (27) yields ‘true’. In effect, the programmer is given complete freedom in deciding how to update output c when n equals zero. We use Opening Rule D to perform the following refinement, which yields **write** statement (12) in Figure 3.

$$\begin{array}{l} \text{Specification (27)} \\ \sqsubseteq c: [c = \infty] \end{array}$$

In this case, the proof obligation is ‘ $(c(\tau) = \infty) \Rightarrow \text{true}$ ’ and is thus immediately satisfied.

To complete the refinement, we again back up the refinement tree and use Opening Rule C to focus on specification (23). Doing so introduces the following hypotheses to the context.

$$\begin{array}{l} \mathbf{pre} \ \tau \leq v \quad (\text{xvi}) \\ \mathbf{pre} \ (b(t) \neq 0 \Rightarrow \quad (\text{xvii}) \\ \quad \quad c(\tau) = a(t)/b(t)) \\ \mathbf{pre} \ b(t) = 0 \Rightarrow c(\tau) = \infty \end{array}$$

Adding this knowledge to the predicate in specification (23) allows us to apply Opening Rule D to enable the following refinement step.

$$\begin{array}{l} \text{Specification (23)} \\ \sqsubseteq c: [\text{stable}(c, \tau_0 \dots \tau) \wedge \quad (\text{28}) \\ \quad \quad w \leq \tau] \end{array}$$

The required proof is straightforward. Hypothesis (xvii) tells us that output c already has a satisfactory value at the beginning, and hypothesis (xvi) tells us

that this has been achieved no later than time v . Therefore, to achieve the requirement that c has the desired value from time v to time w , we merely need to ensure that the value of c remains unchanged until at least time w . (It does not matter if the program executes beyond time w because the original specification in Figure 2 did not put a bound on the final finishing time.) A trivial step to remove c from the frame [11, 10], which implicitly makes it unchanging, allows us to omit the stability predicate.

$$\begin{array}{l} \text{Specification (28)} \\ \sqsubseteq : [w \leq \tau] \end{array}$$

This is equivalent to **delay until** statement (14) in Figure 3 and completes the formal derivation of the target program.

5 Conclusion

Via a detailed example, we have illustrated the capabilities of a new tool for formal development of real-time programs. It combines real-time refinement concepts with the program window inference paradigm to provide a practical basis for computer-aided development of verified real-time software. Of course, as with any refinement or theorem-proving tool, finding the overall development or proof strategy is the programmer’s responsibility. Nevertheless, by automatically maintaining the myriad trivial details associated with rigorously applying each refinement step, the tool eases the burden on the programmer, and thus reduces the possibility of error.

Other, larger case studies have also been undertaken using the tool, involving the introduction of iterative code, subroutines and ‘auxiliary’ variables. Admittedly these examples are still far from industrial scale. Nevertheless, the success in industry of other refinement aids such as the B tool [3] gives us confidence that our tool will be applicable in such domains as it matures.

Future enhancements to the tool will see the introduction of concise notations for representing time intervals, and additional tactics for further automating discharge of time-dependent proof obligations.

Acknowledgements. We wish to thank Ian Hayes, Axel Wabenhurst and an anonymous reviewer for their comments on an earlier version of this article. This research was funded by ARC Large Grant A49702415: *Efficient development of verified concurrent real-time programs through tool support.*

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Language and Systems*, 16(5):1543–1571, September 1994.
2. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
3. J. C. Bicarregui et al. Formal methods in practice: Case studies in the application of the B method. *IEE Proceedings — Software Engineering*, 144(2):119–133, April 1997.
4. M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 40–61. Springer, 1997.
5. D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.
6. J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, February 1995.
7. C. J. Fidge, I. J. Hayes, and G. Watson. The deadline command. *IEE Proceedings—Software*, 146(2):104–111, April 1999. Special Issue on Real-Time Systems.
8. S. Grundon, I. J. Hayes, and C. J. Fidge. Timing constraint analysis. In C. McDonald, editor, *Computer Science '98: Proc. 21st Australasian Computer Science Conference*, pages 575–586. Springer-Verlag, 1998.
9. I. J. Hayes. Separating timing and calculation in real-time refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific '98*, Discrete Mathematics and Theoretical Computer Science, pages 1–16. Springer-Verlag, 1998.
10. 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, 1996*, Electronic Workshops in Computing. Springer-Verlag, 1997. <http://www.ewic.org.uk/ewic/>.
11. I. J. Hayes and M. Utting. Deadlines are termination. In D. Gries and W.-P. de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 186–204. Chapman and Hall, 1998.
12. J. Hooman. Assertional specification and verification. In M. Joseph, editor, *Real-Time Systems: Specification, Verification and Analysis*, chapter 5, pages 97–146. Prentice-Hall, 1996.
13. B. P. Mahony. The refinement calculus and data-flow processes. In *Proc. Second Australasian Refinement Workshop*, pages 1–28, Brisbane, September 1992.
14. B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, Department of Computer Science, University of Queensland, 1992.
15. 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.
16. C. Morgan. *Programming from Specifications*. Prentice-Hall, second edition, 1994.
17. R. Nickson and I. J. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.
18. M. Schenke. Transformational design of real-time systems. part II: From program specifications to programs. *Acta Informatica*, 36:67–96, January 1999.
19. M. Schenke and E.-R. Olderog. Transformational design of real-time systems. part I: From requirements to program specifications. *Acta Informatica*, 36:1–65, January 1999.

20. D. Scholefield, H. Zedan, and He Jifeng. A specification-oriented semantics for the refinement of real-time systems. *Theoretical Computer Science*, 131:219–241, 1994.
21. J. U. Skakkebaek. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Department of Computer Science, Technical University of Denmark, November 1994. ID-TR 1994-150.
22. I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software—Practice & Experience*, 25(3):305–330, March 1995.
23. M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS Seventh Refinement Workshop*, Electronic Workshops in Computing, Springer-Verlag, 1996. <http://www.ewic.org.uk/ewic/>.
24. M. Utting and C. J. Fidge. Refinement of infeasible real-time programs. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 243–262. Springer-Verlag, 1997.
25. T. Vickers. An overview of a refinement editor. In *Proceedings of the Fifth Australian Software Engineering Conference (ASWEC'90)*, pages 39–44, 1990.
26. A. Wabenhurst. A model of real-time distributed systems. In D. Gries and W.-P. de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PRO-COMET'98)*, pages 462–481. Chapman and Hall, 1998.
27. L. Wildman and I. Hayes. Supporting contexts in the sequential real-time refinement calculus. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop & Formal Methods Pacific '98*, Discrete Mathematics and Theoretical Computer Science, pages 352–369. Springer-Verlag, 1998.

A Semantics of real-time specification statements

As noted in Section 3.1, it is necessary to adopt a special interpretation of assertions and specification statements in

real-time applications. Let $\text{wp}(S, P)$ be the weakest precondition for statement S to achieve postcondition P . To allow for the possibility that the predicate A in a real-time assertion contains unindexed occurrences of timed-trace variables, we make use of the ‘@’ operator (Section 4.1).

$$\text{wp}(\{A\}, P) = A @ \tau \wedge P$$

That is, an assertion only guarantees to achieve P if P was already true (assertions do not change the program state), and the assertion predicate A is true at the current time τ .

The semantics for a real-time specification statement also uses @ to allow for the possibility that its predicate R contains unindexed and zero-subscripted timed-trace variables. Local and output variables that do not appear in the frame \tilde{x} are treated as unchanging [23]. Let Γ be the set of all local and output variables currently in scope.

$$\begin{aligned} & \text{wp}(\tilde{x}: [R], Q) \\ &= (\forall \tau : \text{Time} \bullet \\ & \quad (\tau_0 \leq \tau \wedge \\ & \quad \text{stable}(\Gamma \setminus \tilde{x}, \tau_0 \dots \tau) \wedge \\ & \quad R @ (\tau_0, \tau)) \Rightarrow Q)[\tau/\tau_0] \end{aligned}$$

This tells us that the only variable that may actually change value is the special time variable τ . In doing so it may not go backwards. Trace ‘variables’ do not actually change in the usual sense, but instead each statement constrains their ranges over a particular time interval [23]. Expression $\Gamma \setminus \tilde{x}$ gives the set of local and output trace variables *not* influenced by this statement. It is required that each such variable remains stable (Section 4.1) in the interval between times τ_0 and τ . Predicate R can be used to specify some requirement on the variables in \tilde{x} between these two times (although there is no restriction against referring to values outside this time interval). When specification statements of this form are composed in sequence, the requirements on each consecutive time interval act in concert to fully define the overall behaviour of each trace variable [24].