SOFTWARE VERIFICATION RESEARCH CENTRE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 02-38

The Variety of Variables in Automated Real-Time Refinement

L. Wildman, C. J. Fidge and D. Carrington

November 2002

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

Under consideration for publication in Formal Aspects of Computing

The Variety of Variables in Automated Real-Time Refinement

Luke Wildman[1], Colin Fidge[1], and David Carrington[2]

[1]Software Verification Research Centre
[2]School of Information Technology and Electrical Engineering
The University of Queensland, Australia

Keywords: Computer-aided programming; Program refinement theory; Real-time programming

Abstract.
The refinement calculus is a well-established theory for deriving program code from specifications. Recent research has extended the theory to handle timing requirements, as well as functional ones, and we have developed an interactive programming tool based on these extensions. Through a number of case studies completed using the tool, this paper explains how the tool helps the programmer by supporting the many forms of variables needed in the theory. These include simple state variables as in the untimed calculus, trace variables that model the evolution of properties over time, auxiliary variables that exist only to support formal reasoning, subroutine parameters, and variables shared between parallel processes.

1. Introduction

The refinement calculus is a formalism for systematically deriving programs from their specifications [Mor90, BvW98], and is supported by a number of automated programming tools [BGL$^+$97, TM95, Vic90]. Recently, the calculus was extended to handle timing requirements, as well as functional ones [Hay02, HU01, Hay00, Hay98]. However, the timing extensions introduce considerable complexity to the theory and cannot be accommodated by existing tools. We have therefore developed a new version of our Program Refinement Tool [CHN$^+$98] that supports the 'timed' refinement calculus [WFC00, WH98].

Based on our experiences with a number of real-time case studies, this paper explains how the enhancements to the tool handle a particular aspect of the real-time calculus, namely the many different roles played by 'variables'. The set of variables currently in scope define the available state space, and their modelling and maintenance is a central feature of the refinement tool, in the same way that its symbol table is a central feature of a high-level language compiler. In particular, the timed refinement calculus introduces several new forms of variables, not found in the standard calculus. In conventional refinement theory only a few

Correspondence and offprint requests to: C. J. Fidge, Software Verification Research Centre, The University of Queensland, Queensland 4072, Australia. E-mail: `cjf@svrc.uq.edu.au` forms of variable are used: *state* variables that will appear in the generated program code; *logical* variables (and *constants*) that appear temporarily to simplify formal proof steps; and named *formal parameters* to subroutines [Mor90]. However, the real-time theory complicates the picture by

- introducing a special variable 'τ' to represent the *current time* [Hay02],
- modelling state variables as *timed traces*, i.e., functions that record a history of values over time [HU01],
- introducing (non-trace) *auxiliary* variables to support reasoning about timing properties [Hay00], and
- introducing *concurrent* variables which are maintained by separate parallel processes [Mah92, MH92].

Our extended refinement tool now recognises all of these forms of variable, and provides appropriate operators and theories for their manipulation, thereby greatly simplifying the programmer's task. In this paper we give an overview of the way the tool exploits the theory by maintaining variable 'contexts'. This is illustrated by particular refinement steps from three recently completed case studies, one that generates sequential code [HU01], one that involves subroutines [HU98], and another involving parallel processes.

Section 2 reviews previous work on refinement tools. Section 3 reviews the roles of variables in the standard refinement calculus and the original version of the Program Refinement Tool. Section 4 then describes the basic extensions for supporting variables in the real-time version of the Program Refinement Tool. Section 5 explains how the tool was extended further to allow for subroutine parameters. Finally, Section 6 describes recent extensions for variables shared between parallel processes.

2. Previous Work on Refinement Tools

Semantically, applying refinement steps involves the same principles as theorem proving. Therefore, a number of refinement tools have been developed as extensions to automated theorem provers.

For instance, the CADiZ tool [TM95], which comprises formatting and typechecking components for the Z specification notation, has an associated experimental component called Zeta to perform refinement steps [JLM$^+$94]. It uses the ProofPower theorem prover to undertake proofs. A disadvantage with this arrangement is that although Zeta maintains knowledge of variable declarations and types, this information must be transferred as "an unintelligible stream of text" [Sha93] from Zeta to the prover. A similar approach appears to have been used by the earlier Refinement Editor [Vic90].

At the other extreme is the Refinement Calculator [BGL$^+$97, Lai00] which is integrated with the HOL theorem prover. It shares its basic inference mechanism with the Program Refinement Tool, but has a more elegant Graphical User Interface. Most significantly for our purposes here, however, it identifies types in the refinement language with types in the prover's logic, and supports this with a parsing and pretty-printing layer between the programmer and the proof engine. This means the refinement notation can exploit the inherent typing of HOL's higher order logic.

By contrast, our Program Refinement Tool [CHN$^+$98], including its real-time extension [WFC00], takes an intermediate approach. Like the Refinement Calculator it is built on an existing theorem prover, in this case Ergo [NTU96]. However, since Ergo uses first order logic, the Program Refinement Tool explicitly encodes type information as axioms, in a way similar to Zeta. In particular, we maintain knowledge about variables and their types in a local 'context', in the same way that Ergo proofs maintain a set of relevant hypotheses [WH98]. This approach avoids having a loose coupling between the refinement and proof tools, and the need for parsing and printing routines.

Most significantly, no refinement tools other than the Program Refinement Tool provide support for a real-time refinement calculus in the style of Morgan [Mor90] and Back and von Wright's [BvW98] well-known calculi. The closest comparable system to ours is Hooman's implementation of his real-time development laws [Hoo96] in the PVS theorem prover. Like the timed refinement calculus, Hooman's formalism includes a special variable to denote the current time, auxiliary variables which are not affected by the passage of time, and functions over the time domain to model time-dependent variables and predicates. However, since Hooman's formalism is based on temporal and Hoare logic, his development steps are presented in a 'guess and verify' style, rather than the refinement calculus' more constructive approach.

3. Background: Variables in the Program Refinement Tool

In this section we briefly review the role of variables in the conventional (untimed) refinement calculus, and relevant features of the original (untimed) Program Refinement Tool.

3.1. Variables in the Refinement Calculus

Refinement is the process of translating requirements into executable programs, in such a way that essential properties are verifiably preserved. The refinement calculus can be approached either from a theoretical [BvW98] or programming [Mor90] perspective. Our work here is largely influenced by Morgan's straightforward approach, which provides numerous laws suitable for direct application by programmers [Mor90].

To enable a smooth transition from requirements to programs, the refinement calculus operates on a 'wide-spectrum' modelling language in which specification-level constructs and programming language code may coexist. The modelling language includes familiar structured programming statements for manipulating program state, including assignment (**:=**), sequential composition (**;**), choice (**if-then-else**), and iteration (**while**). It additionally contains a *specification statement* for expressing requirements to be implemented [Mor90, §1.4.3]. This is denoted '$x\colon[P\ ,\ Q]$' and consists of a *frame* x which lists those variables that may be modified, a *precondition* predicate P which characterises those initial states in which the statement is expected to be executed, and a *postcondition* predicate Q which chararactises those final states in which the statement must finish. An important special case is an *assertion*, which is a specification with empty frame and postcondition 'true', and is denoted '$\{P\}$' [Mor90, §8.2].

The representation of programming language variables in the refinement calculus is straightforward. A *variable* declaration '$[\![\,\mathbf{var}\ v : T \bullet S\,]\!]$' declares a variable v, of type T, whose scope extends over statement S [Mor90, §3.3]. Within a specification statement $x\colon[P\ ,\ Q]$, a declared variable v may be referenced in two ways. The undecorated form 'v' denotes v's initial value in precondition P and final value in postcondition Q. The zero-subscripted form 'v_0' may appear in a postcondition Q only, and denotes v's initial value [Mor90, p. 52].

To help express requirements and support proofs, *logical constant* declarations '$[\![\,\mathbf{con}\ c : T \bullet S\,]\!]$' make a constant c, of type T, available for use in specification statement predicates [Mor90, §6.1]. Logical constants are not intended to remain in the final refined code. Programming language constants are also allowed in the calculus, and are distinguished by the keyword '**const**'—they can be modelled as a degenerate form of variable that never changes value [Mor90, p. 34].

In particular, notice that whereas executable statements, such as assignments, provide a way of updating the current system state, variable (and constant) declarations provide the means of updating the state space itself. The refinement calculus is then completed by a set of formally-verified *refinement rules* which allow specification statements and logical constant declarations to be eliminated and replaced by executable program code [Mor90, App. C].

3.2. The Program Refinement Tool

The Program Refinement Tool [CHN+98] is a research prototype for experimenting with interactive program refinement. It consists of extensions to the *Ergo* theorem prover [MNU97, NU95, UNT96]. In particular, it is based on the Program Window Inference paradigm [NH97], which allows refinement steps and proofs of side conditions to be performed in a similar manner.

At any time the tool presents the programmer with a *window* consisting of a current *focus* F, which normally consists of a specification in need of refinement, and a *context* C, which comprises known information about the state of the system at the point where the focus occurs. The programmer's task is to transform the focus F into program code, via the refinement relation '$\sqsubseteq$'. To do this, the tool offers two forms of program development step, refinement and window-opening, both of which are controlled by rules stored in the tool's theory base.

A *refinement rule* has the following form.

$$\frac{C}{F \sqsubseteq G}$$

In context C, it transforms the current focus F into a goal G, in a way that maintains the refinement relation '$\sqsubseteq$'. Refinement rules are used to introduce declarative and executable programming language statements into the program under construction.

An *opening rule* has the following form.

$$\frac{D \Rightarrow (X \sqsubseteq Y)}{C \Rightarrow (F[X] \sqsubseteq F[Y])}$$

It states that the compound focus $F[X]$ may be transformed by refining one of its components X. This is done by 'opening' a new window, with X as the new focus, which is then refined to goal Y in new context D. Window opening rules allow the programmer to navigate through the program under construction.

The tool allows the programmer to select applicable refinement or opening rules at each step. Depending on the particular rule chosen, it either automatically calculates new contexts and goals, or prompts the programmer for them in those cases where they cannot be determined automatically. Most side conditions associated with the rules are discharged automatically, although some may require theorem-proving assistance from the programmer.

In particular, the tool automatically maintains the current window's *context*. The context consists of a list of hypotheses, divided into three groups.

- The **pre** context maintains knowledge about the program state at the beginning of the current focus.
- The **inv** context maintains information which is invariant with respect to the current focus, especially the types of variables currently in scope.
- The **lval** context maintains information about the state space, especially the names of variables currently in scope.

A reference to some variable 'v' appearing in a modelling language statement is thus fully defined by v's definition in the surrounding context. Zero-subscripted references 'v_0' are implemented in the tool as abbreviations for logical constants that capture the corresponding variable's initial value [Mor90, Abbrev. 6.1].

Significantly, both the **pre** and **inv** contexts refer to the *values* of variables, whereas the **lval** context refers to the *names* of variables. (The context's name means 'left value', which derives from the appearance of a variable identifier on the left-hand side of an assignment to denote the variable's name, rather than its value.) This information is clearly critical to the applicability of both refinement and opening rules and its maintenance is a central aspect of the tool. (Again, an analogy can be drawn between the data stored in a compiler's symbol table and the variable typing and scoping information maintained by the **pre** and **lval** contexts, respectively.) Indeed, this is one of the primary advantages of tool support. Handcrafted refinements can rarely afford to perform this tedious bookkeeping explicitly, except in the most trivial cases.

4. Variables in Sequential Real-Time Refinement

In this section we use some extracts from a case study to introduce the way the timed refinement calculus uses variables during development of sequential real-time program code [HU01], and the corresponding extensions made to the Program Refinement Tool [WFC00].

4.1. The Receiver Case Study

The case study involves development of a 'receiver' program which must read a sequence of characters from an asynchronous data stream into a character array. The requirement is heavily time-dependent because each character is visible in a memory-mapped input register for only a limited duration. Fig. 1 shows the assumed behaviour of the incoming data stream *in*. Each character is separated from its successor by *chsep* seconds and remains defined in the input register for at least *chdef* seconds. The first character appears *chsep* seconds after a specific starting time *start*. The last character in the sequence is the special end-of-text character '`etx`'.

Hayes and Utting present a refinement of this time-dependent requirement in their timed version of the refinement calculus [HU01]. This was a non-trivial exercise—their refinement occupies several pages despite

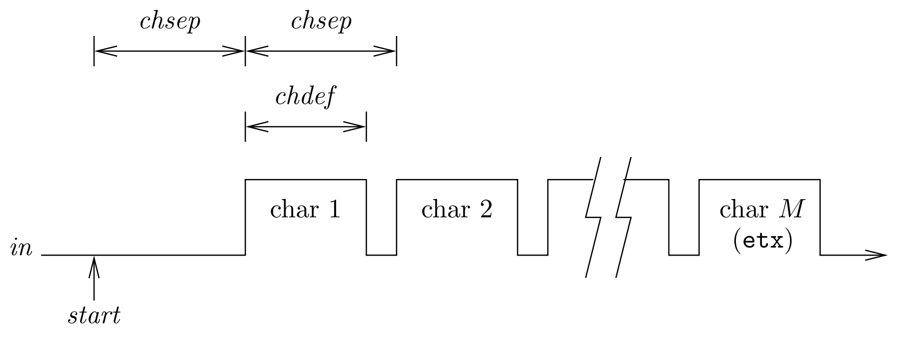

Fig. 1. Behaviour of the incoming data stream for the receiver case study.

omitting much detail. Our real-time extensions to the Program Refinement Tool allowed us to duplicate their refinement with all steps completed and all proof obligations fully discharged. Below we use some extracts from this exercise to explain how the tool's handling of variables was extended to implement significant features of the real-time calculus.

4.2. Declaring Real-Time Variables

An obvious feature of the timed calculus from the programmer's perspective is that it allows explicit references to the time domain '`Time`', both via the current-time variable τ and by the ability to index variables with times. Predicates in the timed calculus can thus express not only simple state transition behaviour, but also the history of interactions between the program and its environment. To model the way that program variables evolve over time, they are represented in the real-time calculus not as simple values, but as *timed traces*. These are functions from times to values, and can thus capture the variable's entire history [UF96]. Consequently, the timed calculus distinguishes several classes of variable, each of which satisfies a particular role and has different characteristics [Hay98].

- An *input* variable, declared with keyword '**input**', is a timed trace that can be accessed by the program, but not changed by it.
- An *output* variable, declared with keyword '**output**', is a timed trace that is constructed by the program and is visible externally.
- A *local* variable, declared with keyword '**var**', is a timed trace that is constructed by the program but cannot be seen by the environment.
- The *current time* variable 'τ' is a non-trace variable of type `Time` [UF96]. It may not appear in programming language code.
- An *auxiliary* variable, declared with keyword '**aux**', is a non-trace variable used to support formal proof. Like logical constants, such variables may not appear in programming language code [Hay00].

The timed calculus also supports *logical constants*, declared with keyword '**con**', like those in the standard calculus [HU01]. Unlike auxiliary variables, they cannot change value and therefore never appear in decorated form. They are treated as simple values (rather than interpreting them as trace functions with a single-valued range). Most importantly, they may be of absolute-time type `Time`, so that they can be used to record significant times. The timed calculus also allows a **const** declaration for constants that appear in the executable program [HU98].

The Program Refinement Tool implements all of these different variable declarations through extensions to its context mechanism [WH98]. For instance, the programmer begins the 'receiver' case study by declaring variables and constants that are global to the code fragment to be developed. These include the various time-valued constants, such as the separation between characters, *chsep*. The programmer declares '**con** *chsep*: `Time`' to the tool, which causes it to add the following hypotheses to the context.

lval $\mathit{chsep} \in \mathrm{Constants}$
inv $\mathit{chsep} : \mathtt{Time}$

More interestingly, when the programmer declares the incoming stream of characters as '**input** *in* : `Char`', the tool adds the following hypotheses.

lval $in \in$ Inputs
inv $in : \mathtt{Time} \to \mathtt{Char}$

The invariant reminds us that, in the timed calculus, trace variables are represented as total functions from the `Time` domain. They have a value at every point in time, rather than just an initial and a final value with respect to each statement. In effect, a timed-trace variable is a constant function which models all past and future values [UF96]. This also applies to output (**output**) and local (**var**) variables. For instance, the receiver program is required to store the message received in a local array, modelled here as a function over the natural numbers. The programmer's declaration of '**var** *msg* : `Nat` → `Char`' causes the tool to add the following information to the context.

lval $msg \in$ Locals
inv $msg : \mathtt{Time} \to (\mathtt{Nat} \to \mathtt{Char})$

The special 'now' variable, τ, is implicitly declared by the tool in every context, in both undecorated and zero-subscripted forms.

lval $\tau \in$ Variables
lval $\tau_0 \in$ Constants
inv $\tau : \mathtt{Time}$
inv $\tau_0 : \mathtt{Time}$

In this case, τ appears to the tool as a conventional (non-trace) variable which has an initial and final value in the usual way. Auxiliary variables are represented similarly. For instance, mid-way through the receiver's refinement, it becomes convenient to temporarily introduce a loop counter variable n, to make it easier to express the timing constraints defining the earliest and latest times at which the n^{th} character may be sampled ($start + n * chsep$ and $start + n * chsep + chdef$, respectively). This is declared as '**aux** n : `Nat`' and adds the following information to the context.

lval $n \in$ Variables
inv $n : \mathtt{Nat}$

The tool does not declare zero-subscripted versions of auxiliary variables by default. If needed, they may be introduced as explicitly-declared constants. (In fact, initial value 'n_0' *is* needed later during the receiver's refinement, and is introduced in this way.)

4.3. Expressing Real-Time Requirements

Once declared, variables may appear within the program code under construction. Input, output and local variables may be used in programming language statements, while specification statements may additionally refer to auxiliary variables and the special 'now' variable. For instance, the top-level requirements specification of the receiver is preceded by the following assertion.

$\{\tau \leqslant start\}$

This states that the current time, when the receiver program begins, is assumed to be no later than absolute time *start*. This is essential because if the receiver program is activated too late, it will not be able to read the whole message.

More generally, 'timed' variables may be used in several different ways by predicates within timed refinement calculus specification statements [Hay98]. The non-trace variables, i.e., τ and the auxiliary variables, can appear in undecorated (final value) and zero-subscripted (initial value) forms as in the standard calculus. However, the timed-trace input, output and local variables are fundamentally different. Although complex mechanisms have been suggested for 'lifting' predicates so that the correct usage of such variables can be determined automatically [MF01], the Program Refinement Tool instead avoids potential ambiguities by having the programmer make the intended usage of each variable clear. Thus, a timed-trace variable v in a predicate may

- appear undecorated as ‘v’, in which case it is treated as a function,
- be indexed by an absolute time t as ‘$v(t)$’ to refer to its value at that specific time,
- be decorated as ‘$\grave{}v$’, to denote its value when the enclosing statement began, i.e., $v(\tau_0)$, and
- be decorated as ‘v'’, to denote its value when the enclosing statement ends, i.e., $v(\tau)$.

These decorations are recognised by the tool as syntactic abbreviations.

For example, after the assertion above, (part of) the receiver program’s basic requirement is expressed as the following specification statement. (The precondition is merely ‘true’ and is omitted.) Let constant M be the number of characters in the message. (In the full case study this value is derived from the location of the ‘`etx`’ character in the data stream [HU01].)

$$\mathit{msg}\colon [\mathit{msg}' = \{i : 1 \mathrel{..} M \bullet (i \mapsto \mathit{in}(\mathit{start} + i * \mathit{chsep}))\}]$$

The frame tells us that the requirement is to update *msg* in such a way that the predicate inside the square brackets is satisfied. The predicate itself shows two different uses of timed variables. On the left, ‘msg'’ denotes the required final value of function (array) *msg* when the statement terminates. The set comprehension on the right constructs this required final value using an index i which ranges over the number of characters in the message. For each index value i, $\mathit{msg}(i)$’s final value is required to map to the specific character appearing in the incoming data stream *in* at absolute time $\mathit{start} + i * \mathit{chsep}$. These times denote the earliest moments at which each character is known to be available. (Fortunately, the program is not required to sample the values at *exactly* these times—Fig. 1 reminds us that the program has up to *chdef* seconds to read each character from data stream *in*.)

However, this *timed* specification statement is defined semantically in quite a different way from specification statements in the standard refinement calculus [HU01]. To account for the fact that (most) variables in the timed calculus actually denote traces, and to support the implicit ‘now’ variable, the underlying meaning of the above statement is actually as follows, expressed as a standard refinement calculus specification statement.

$$\begin{aligned}\mathit{msg}\colon [&\mathit{msg}(\tau) = \{i : 1 \mathrel{..} M \bullet (i \mapsto \mathit{in}(\mathit{start} + i * \mathit{chsep}))\} \land \\ &\tau_0 \leqslant \tau \land \\ &\mathrm{stable}((\mathrm{Outputs} \cup \mathrm{Locals}) \setminus \{\mathit{msg}\}, \tau_0 \mathrel{..} \tau)]\end{aligned}$$

Here the reference to the final value of trace variable *msg* has been replaced by the expression ‘$\mathit{msg}(\tau)$’ it denotes. However, no such ‘lifting’ was required for the reference to trace variable *in* because it was already explicitly indexed. Also two additional conjuncts have been added to the predicate. The first tells us that the finishing time τ of the statement can be no earlier than its starting time τ_0. This is an implicit property of any statement. The second says that all output and local trace variables, less the variable *msg* in the frame, remain ‘stable’ for the interval from times τ_0 to τ, inclusive. This ensures that all trace variables under control of this program fragment are well-defined for the interval during which this statement executes, even those that are not explicitly changed by the statement. Stability of a trace variable means that it has the same value at every moment in the interval of interest [UF96].

The Program Refinement Tool supports all these additional features of timed specification statements through appropriate abbreviations, thus making the timed calculus appear similar to the standard one from the programmer’s perspective. The tool also includes appropriate theories for reasoning about the ‘stability’ of variables over specific time intervals.

As well as changing the meaning of specification statements, the timed calculus also extends the semantics of executable programming language constructs [HU01]. For instance, a typical conditional statement, ‘**if** B **then** S_1 **else** S_2’, is defined to allow an ‘idle delay’ before and after the execution of statements S_1 and S_2. These delays model the run-time overheads of evaluating expression B, jumping to the appropriate choice, and exiting the entire compound statement. It is important to account for these delays because, although output and local variables remain stable during these intervals, input variables may change value while, for instance, expression B is being evaluated. Again, the Program Refinement Tool hides these complexities from the programmer by providing appropriate definitions and abbreviations, thus allowing the timed calculus to be used in a manner as close as possible to that of the familiar untimed one.

4.4. Real-Time Refinement and Opening Rules

Most importantly, the timed calculus provides numerous refinement rules for translating specifications such as that above into executable program code [HU01]. The Program Refinement Tool therefore implements corresponding timed refinement and opening rules, and automatically maintains and applies appropriate contexts [WFC00].

For instance, a timed refinement rule with significant consequences for the maintenance of contexts in the tool is the following [HU01].

Refinement Rule 1 (Introduce variable declaration).

$$\frac{\begin{array}{l}\textbf{lval}\ v \notin w\\ \textbf{lval}\ v \text{ not-free-in } P\\ \textbf{lval}\ v \text{ not-free-in } Q\\ \textbf{inv}\ \text{non-empty}(T)\\ \textbf{inv}\ \text{idle-invariant}(P)\\ \textbf{inv}\ \text{pre-idle-invariant}(Q)\\ \textbf{inv}\ \text{post-idle-invariant}(Q)\end{array}}{w{:}\,[P\ ,Q] \sqsubseteq [\![\, \textbf{var}\ v : T \bullet v, w{:}\,[P\ ,Q]\,]\!]}$$

Given a specification statement $w{:}\,[P\ ,Q]$, this rule allows the programmer to declare a new variable v, of type T, which is then made available to the specification by adding it to the frame. This refinement step is allowed provided the list of conditions above the line can be satisfied. The first four provisos are conventional and are found in the corresponding rule in the standard refinement calculus [Mor90, §3.3]. They tell us that the new variable v must not already be listed in the specification's frame w, that v may not occur free in predicates P and Q, and that type T must contain at least one value.

However, the final three provisos in Refinement Rule 1 are peculiar to the timed calculus. They all stem from the possibility that the precondition P and postcondition Q predicates in the specification statement may be time-sensitive, i.e., their truth may be altered by the passage of time. If this were so, it would not be safe to introduce the declaration, because this may involve run-time overheads associated with allocating and deallocating memory space. The three additional provisos therefore ensure that the predicates in the specification are not affected by such overheads. Recall that a precondition predicate P refers to the initial state only, whereas a postcondition predicate Q may refer to both initial and final states [Mor90, Ch. 6]. The 'idle-invariant' property therefore checks that the truth of a simple state predicate is unaffected by an arbitrary idle delay which does not change the values of output or local variables [HU01]. The 'pre-idle-invariant' and 'post-idle-invariant' properties check that the truth of a relation between initial and final values is unaffected by a preceding or succeeding delay, respectively [Hay98].

In general, proving idle-invariance of a predicate can be a significant theorem proving problem. Fortunately, there are syntactic checks that can be applied in most cases [HU01], and the Program Refinement Tool discharges these automatically, in much the same way that it automatically discharges most type obligations [WFC00].

In the receiver case study, for instance, the programmer applies this rule to the timed specification statement from Section 4.3 above to introduce a local variable *ch* that will be used to hold the character most recently sampled from *in*.

$$\begin{array}{l} \quad \mathit{msg}{:}\,[\mathit{msg}' = \{i : 1 \mathinner{..} M \bullet (i \mapsto \mathit{in}(\mathit{start} + i * \mathit{chsep}))\}] \\ \sqsubseteq \text{`by Refinement Rule 1'} \\ \quad [\![\, \textbf{var}\ \mathit{ch} : \texttt{Char} \bullet \\ \qquad \mathit{ch}, \mathit{msg}{:}\,[\mathit{msg}' = \{i : 1 \mathinner{..} M \bullet (i \mapsto \mathit{in}(\mathit{start} + i * \mathit{chsep}))\}] \\ \quad]\!] \end{array}$$

The Program Refinement Tool proves the rule's first four provisos trivially in this case and the fifth is not relevant since the specification statement has no explicit precondition. That the complex postcondition above is both pre and post-idle-invariant is less obvious, however. Fortunately, the tool takes advantage of the context and relevant syntactic checks to prove this. In particular, a predicate is pre-idle-invariant if it contains no free references to the starting time τ_0 [HU01, Law 45], which is true of the predicate above. To show post-idle-invariance, the tool exploits the context in which this refinement was performed. Thanks to the '**lval**' properties $start \in$ Constants and $chsep \in$ Constants, we know that expression '$start + i * chsep$' above is also constant and denotes a specific *absolute* time, which is enough to tell us that the reference to *in* above is unaffected by the passage of time. Also, since the context tells us that $msg \in$ Locals, we know that the (implicit) reference to $msg(\tau)$ is post-idle-invariant because local variables are under the control of the current statement and do not change value merely due to the passage of time.

Having performed this refinement step, the programmer would naturally like to focus on the specification statement within the declarative block, in order to make use of the new variable. To support this, the Program Refinement Tool implements opening rules such as the following [WH98]. Let S be a statement in our modelling language, P be a set of precondition hypotheses, I be a set of invariant hypotheses, and L be a set of l-value hypotheses. Let '**idle**' be a statement which takes an arbitrary amount of time but does not change output, local or auxiliary variables [HU01]. Let 'hide(x, H)' be an operator that returns hypothesis list H less any hypotheses that refer to term x. Finally, let 'sp.S.P' be the *strongest postcondition* resulting from starting statement S in a state satisfying precondition predicate P [NH97].

Opening Rule 1 (Enter variable block).

$$\frac{\begin{array}{l}(\mathbf{inv}\ \mathrm{hide}(v, I)\\ \ \mathbf{lval}\ \mathrm{hide}(v, L)\\ \ \mathbf{pre}\ \mathrm{sp}.\mathbf{idle}.(\mathrm{hide}(v, P))\\ \ \mathbf{lval}\ v \in \mathrm{Locals}\\ \ \mathbf{inv}\ v : \mathtt{Time} \to T) \Rightarrow S_1 \sqsubseteq S_2\end{array}}{(\mathbf{pre}\ P,\ \mathbf{inv}\ I,\ \mathbf{lval}\ L) \Rightarrow [\![\, \mathbf{var}\ v : T \bullet S_1 \,]\!] \sqsubseteq [\![\, \mathbf{var}\ v : T \bullet S_2 \,]\!]}$$

This rule allows the programmer to focus on statement S_1 (by opening a window), refine it to statement S_2 (using some refinement rules), and then substitute S_2 for S_1 in the variable block (by closing the window). The overall effect of these steps is shown by the contextualised refinement below the line. The context above the line defines the hypotheses under which the refinement from S_1 to S_2 may be performed. It tells us that focusing on statement S_1 maintains invariant hypotheses I and l-value hypotheses L. However, it also says that the precondition within the declarative block is the result of following the original precondition P with an idle delay. This accounts for the possible run-time overheads of allocating memory space for variable v. A time-sensitive property that held when the variable block was reached may no longer be true when the statement inside the block begins. Furthermore, in each of these new contexts, already-existing occurrences of variable v are removed by the 'hide' function. This avoids conflicts between a previously-declared variable v, if any, and the one introduced by this declaration. Finally, the new context also contains the additional information that 'v' is now visible as a local timed-trace variable of range type T.

For instance, the next step in the receiver example is to focus on the specification statement within the declaration of variable *ch* above. When the programmer tells the Program Refinement Tool to do this, using Opening Rule 1, the tool automatically adds the following data to the context (as well as calculating the new precondition, which is merely 'true' in this case).

$$\begin{array}{l}\mathbf{lval}\ ch \in \mathrm{Locals}\\ \mathbf{inv}\ ch : \mathtt{Time} \to \mathtt{Char}\end{array}$$

The programmer can then continue applying refinement rules to transform the specification statement to executable code in this updated context. More significantly, at each step the tool makes use of the accumulated information in the context to automatically discharge proof obligations associated with the typing of variables and expressions, the stability of real-time variables, and the invariance of time-sensitive predicates.

By continuing in this way, the Program Refinement Tool was used to successfully duplicate Hayes and Utting's receiver case study [HU01] all the way down to executable program code. Unlike their handcrafted version, however, every refinement step was completed in detail and all proof obligations were fully discharged, thanks to the way the tool automatically maintains and exploits variable contexts.

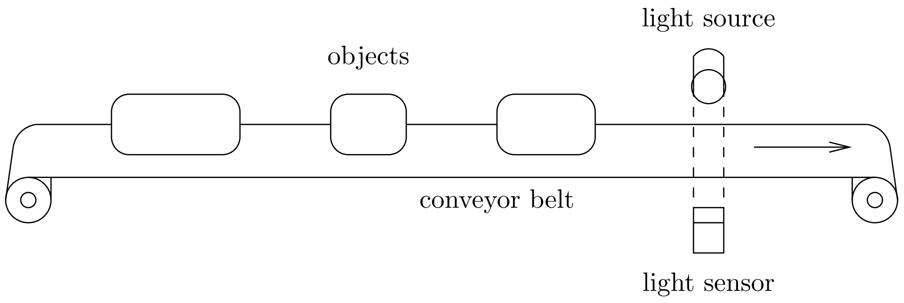

Fig. 2. Assumed operating environment for the object-sizer program.

5. Parameter Passing in Refinement of Real-Time Subroutines

Section 4 explained how real-time variables (and constants) are implemented in the Program Refinement Tool. In this section we explain how the tool supports subroutine parameter passing. This is illustrated by extracts from another significant case study [HU98].

5.1. The Object-Sizer Case Study

The 'object-sizer' case study is another example of real-time program refinement from the literature [HU98]. As shown in Fig. 2, it assumes that objects move along a conveyor belt and pass through a light beam. The program is required to use the signal produced by a light sensor to determine the size of each object. This is done by polling the signal's value and using the observed duration for which the light beam is blocked, plus the known speed of the conveyor belt, to calculate the object's size. Clearly, the problem is heavily time-dependent—correct results rely on sampling the sensor frequently and accurately enough.

Hayes and Utting's refinement of this problem is notable for its introduction of a parameterised subroutine to avoid duplicated code. For each object, the program needs to recognise changes in the sensor signal that mark the passing of an object. Their refinement introduces a subroutine *detect_chng* which recognises a change in the signal's value, and is parameterised by whether the change of interest is a rising or falling edge. However, this introduces the issue of how to represent procedure declarations and calls, and parameter passing, in the timed refinement calculus. The description of these mechanisms in the published object-sizer case study was sketchy, and insufficient for implementation in the Program Refinement Tool. Indeed, as explained in the next section, the handling of subroutines in the refinement calculus generally is rather unclear.

5.2. Previous Work on Procedures in the Refinement Calculus

There have been a number of approaches to adding procedures to the (untimed) refinement calculus. We briefly survey these approaches here, to set the scene for the Program Refinement Tool's solution.

Morgan introduced procedures using a substitution-based semantics [Mor90, Ch. 12]. He allowed scoped procedure declarations to associate a name with a parameterised segment of modelling language code. Within this scope, procedure calls are effectively replaced in-line by the corresponding procedure body. Parameter passing was also achieved by syntactic substitution of actual parameters for formal ones.

Groves subsequently criticised this substitution-based approach by noting that the order in which the procedure body and the calling code is refined can be significant [Gro96]. Cavalcanti *et al.* also noted that the substitution-based approach creates problems due to the way variables local to the procedure body are semantically 'free' [CSW99]. This causes, for instance, subtle differences between performing the substitutions at the point of call versus the point of definition. Cavalcanti *et al.* also observed that Back's earlier, more complicated approach, which allowed parameterised higher-order 'statements' that are not executable until instantiated, does not suffer from the same problems [CSW99]. This approach uses separate forms of parameterisation to model calls by value, result and reference.

In a different setting, Back and von Wright aimed to avoid any distinction between the programming language and its semantics [BvW98, p. 229]. They used higher-order logic lambda expressions to represent procedures, with the lambda arguments providing a call-by-reference parameter passing mechanism [BvW98, §5.5]. They also gave a semantics-level model of recursive procedures using least fixed points [BvW98, §20.3].

The complexities of handling variable scoping in the presence of recursive procedures inspired Hesselink to take a more fundamental approach [Hes99]. To accommodate procedures, he chose to extend the predicate transformer semantics underlying the refinement calculus. Once again the issue of how to appropriately bind variables local to the procedure body was a major concern. Rather than 'stacking' local variables at each call, his solution was to introduce pairs of variable frames to the semantics. These explicitly distinguish those variables that are merely 'accessible' from those that are 'modifiable' in the current scope, and serve to bind local declarations to the corresponding predicate transformer.

Sampaio investigated the use of the refinement calculus to model the way procedures are compiled to assembler code [Sam97]. His weakest-precondition semantics for parameter passing uses explicit assignments to local variables before and/or after the call [Sam97, §5.4]. He takes a simple approach and declares fresh variables for each procedure call, although this introduces a subtle need for 'dynamic' declarations [Sam97, p. 102].

The most recent work in the area is that of Naumann [Nau01]. To define a predicate-transformer semantics for procedures he takes a particularly elegant view by defining a 'higher order' modelling language in which language statements are part of the state space. Thus there is a higher-order type for declared procedures, and procedure-valued variables can be assigned constants of procedure type. This allows declared procedures to be freely passed around the program code to the point at which they are called. For instance, a conventional '**let**' abbreviation can be used to bind a procedure-valued constant to a name which can appear in a procedure call [Nau01, §2]. Furthermore, recursive procedures can be created explicitly by using the existing '**rec**' statement.

Indeed, an approach similar to this was used by Laibinis in extending the Higher Order Logic-based Refinement Calculator for procedures [Lai00]. In parallel with our work, his research also used Staples' technique of separating a procedure's specification from its body [Sta99]. However, we could not directly use a higher-order model in our first-order Program Refinement Tool.

5.3. Declaring Procedures and Parameters

In the light of the various approaches described in Section 5.2, we developed our own solution to implementing 'timed' procedures and parameters in the Program Refinement Tool. In doing so, we devised several new refinement and opening rules.

A procedure declaration in a programming language associates the procedure's name p with a statement S that implements its body. During program development by refinement, however, statement S will initially be a specification of the procedure's functionality, which is subsequently refined to executable statements. In practice, it is helpful to retain the original specification of the procedure because this tells us whether calling this procedure will satisfy some requirement encountered later during the development [Sta99]. Therefore, the Program Refinement Tool uses a multi-part procedure block.

$$[\![\, \mathbf{proc}\ p(F)\ \mathbf{is}\ v\colon [P\ , Q] \sqsubseteq S_1 \bullet S_2 \,]\!]$$

This statement declares a procedure named p which is visible to statement S_2. Here F is an optional formal parameter profile, $v\colon [P,Q]$ is the specification statement defining the procedure's behaviour, and statement S_1 is the procedure body, which must be a refinement of the specification. The parameter profile is a list of parameter names and their associated modes. Available parameter passing modes include the usual '**val**' (value), '**res**' (result), and '**valres**' (value-result). An additional mode, peculiar to the timed calculus, is '**con**' [HU98]. This allows logical constants, especially those of type `Time`, to be passed to procedures—these parameters do not appear in the final, executable program.

For example, mid-way through the object-sizer case study [HU98], the Program Refinement Tool's context includes the following information.

$$\begin{array}{l}
\mathbf{lval}\ \mathit{size} \in \text{Outputs} \\
\mathbf{inv}\ \mathit{size} : \mathtt{Nat} \\
\mathbf{lval}\ \{\mathit{falls}, \mathit{rises}, \mathit{error}\} \subseteq \text{Constants} \\
\mathbf{inv}\ \mathit{falls}, \mathit{rises}, \mathit{error} : \mathtt{Time} \\
\mathbf{lval}\ \{\mathit{speed}, \mathit{delta}\} \subseteq \text{ProgConsts} \\
\mathbf{inv}\ \mathit{speed}, \mathit{delta} : \mathtt{Nat} \\
\mathbf{pre}\ \mathit{error} * \mathit{speed} \leqslant \mathit{delta} \\
\mathbf{pre}\ \mathit{signal\_changes}
\end{array}$$

Output variable *size* is intended to receive the computed size of the next object. Logical constants *rises* and *falls* model the times at which the passage of this particular object causes the sensor signal to rise and fall, respectively—the signal is high while an object is blocking the light beam. (In practice, no program could determine these times with absolute precision, so these specification-level constants cannot appear in the final program.) Logical constant *error* is an acceptable timing error in the detection of changes to the sensor signal. Programming language constant *speed* defines the assumed speed of the conveyor belt (in units of distance over time) and *delta* defines an acceptable tolerance on the measured object size (in units of length). Tolerance *delta* is related by the '**pre**' condition to constant *error*. Finally, the *signal_changes* condition (not shown here) is an assumed property of the signal produced by the light sensor. It states that at any point in time there is a (rising or falling) edge in the future trace of the signal, capable of being detected by the program [HU98]. If this were not true a requirement to detect a change may not be satisfiable. This condition is invariant with respect to idle delays. It holds in practice if there is no limit on the number of objects which may appear on the conveyor belt.

Within this context, the requirement to be satisfied is expressed by the following specification statement.

$$\mathit{size}\colon [\mathit{size}' \in (\mathit{speed} * (\mathit{falls} - \mathit{rises})) \pm \mathit{delta}]$$

In other words, the final value of *size* must equal the object's actual measured size, calculated as $\mathit{speed} * (\mathit{falls} - \mathit{rises})$, but may vary from this by up to *delta* to allow for practical implementation limitations. Recognising that this requirement relies on identifying two changes in the sensor signal, the programmer then decides to introduce a reusable subroutine for this purpose.

The following refinement rule allows parameterised procedure declarations to be introduced to a program. Let S be a statement in our modelling language; p be a procedure name; *vf* be a formal value-parameter list of type T_1; *rf* be a formal result-parameter list of type T_2; *vrf* be a formal value-result-parameter list of type T_3; *cf* be a formal logical-constant-parameter list of type T_4; and $v\colon [P\,, Q]$ be a specification statement. The rule can be modified easily for different parameter profiles.

Refinement Rule 2 (Introduce procedure declaration).

$$\begin{array}{l}
\mathbf{lval}\ p \text{ not-free-in } S \\
\mathbf{lval}\ \mathit{vf} \cap v = \varnothing \\
\mathbf{lval}\ \mathit{cf} \cap v = \varnothing \\
\mathbf{lval}\ \{\mathit{rf}, \mathit{vrf}\} \subseteq v \\
\mathbf{lval}\ \mathit{rf} \text{ not-free-in } P \\
\hline
S \sqsubseteq [\![\, \mathbf{proc}\ p(\mathbf{val}\ \mathit{vf} : T_1, \mathbf{res}\ \mathit{rf} : T_2, \mathbf{valres}\ \mathit{vrf} : T_3, \mathbf{con}\ \mathit{cf} : T_4)\ \mathbf{is} \\
\qquad v\colon [P\,, Q] \sqsubseteq (v\colon [P\,, Q] \text{ in-context } C) \bullet \\
\qquad S\,]\!]
\end{array}$$

$$\text{where } C \overset{\text{def}}{=} \begin{array}[t]{l}
(\mathbf{lval}\ \{\mathit{vf}, \mathit{rf}, \mathit{vrf}\} \subseteq \text{Locals} \\
\ \mathbf{lval}\ \mathit{cf} \subseteq \text{Constants} \\
\ \mathbf{inv}\ \mathit{vf} : \mathtt{Time} \to T_1 \\
\ \mathbf{inv}\ \mathit{rf} : \mathtt{Time} \to T_2 \\
\ \mathbf{inv}\ \mathit{vrf} : \mathtt{Time} \to T_3 \\
\ \mathbf{inv}\ \mathit{cf} : T_4)
\end{array}$$

The refinement rule below the line allows statement S to be enclosed within a procedure declaration. On the right of the refinement relation is the procedure block with its name p and formal parameter list. This is followed by a programmer-supplied specification statement $v\colon [P\,, Q]$ which defines the required behaviour of the procedure's body. The default procedure body is then the same specification statement, but this time with its own context C. The Program Refinement Tool's 'S in-context C' construct allows a statement S to be bound to its own particular context C (which extends the context in which the overall construct resides). In this case it is used to associate the procedure body with the particular context of formal parameters it may access. Abbreviation C tells us that formal parameters vf, rf and vrf appear as local timed-trace variables to the procedure body, and cf as a logical constant. The contextualised specification statement will be refined later to produce the executable procedure body.

Above the line in Refinement Rule 2, the first '**lval**' proviso tells us that new procedure name p may not already appear free in statement S. The next two provisos ensure that the value vf and constant cf parameters do not appear in the procedure body's frame v, because these parameters cannot be changed by the procedure. The fourth proviso states that the result rf and value-result vrf parameters are included in the specification statement's frame, because the procedure body may update these variables. Any other variables listed in v denote global variables updated by the procedure (also see Section 5.5). The final proviso checks that the formal result parameters rf are not used in precondition P. This would be unhelpful because they have not been initialised at that point.

This rule can be used to introduce procedure *detect_chng* to the object-sizer program [HU98]. When the rule is invoked, the Program Refinement Tool prompts the programmer to enter the formal parameter list and the specification of the procedure body, and then checks the necessary provisos. For clarity we omit the contextualised procedure body below.

$$
\begin{array}{l}
\quad \mathit{size}\colon [\mathit{size}' \in (\mathit{speed} * (\mathit{falls} - \mathit{rises})) \pm \mathit{delta}] \\
\sqsubseteq \text{ `by Refinement Rule 2'} \\
\quad [\![\, \mathbf{proc}\ \mathit{detect\_chng}(\mathbf{con}\ \mathit{chngs} : \mathtt{Time}, \mathbf{val}\ \mathit{srqd} : \mathtt{Bool}, \mathbf{res}\ \mathit{chngt} : \mathtt{Time})\ \mathbf{is} \\
\qquad \mathit{chngt}\colon [\mathit{signal\_changes}\ , \mathit{chngs} \leqslant \mathit{chngt}' \leqslant (\mathit{chngs} + \mathit{error}) \wedge \mathit{chngs} < \tau] \sqsubseteq \cdots \bullet \\
\qquad \mathit{size}\colon [\mathit{size}' \in (\mathit{speed} * (\mathit{falls} - \mathit{rises})) \pm \mathit{delta}] \\
\quad]\!]
\end{array}
$$

Here the programmer has introduced formal parameter *chngs* which represents the time at which the sensor signal change to be detected actually occurs. Since the final program cannot directly access this information, it is a logical constant parameter which may not appear in the executable code. A Boolean value parameter *srqd* defines the 'signal required'—it is true if a rising edge is to be detected, otherwise a falling one is sought. Result parameter *chngt* returns the time at which the signal's change was actually detected.

The specification statement defining procedure *detect_chng*'s behaviour has a frame which tells us that it updates formal parameter *chngt*. Its precondition, *signal_changes*, requires that there is indeed a detectable change in the signal's trace after the time at which the procedure is called [HU98]. The postcondition requirement contains two conjuncts. The first says the final value of parameter *chngt* should be within *error* seconds of the time *chngs*, when the change actually occurred. The second says that the finishing time τ of the procedure body must exceed the time at which the signal changed. This ensures that the same change to the signal cannot be read twice—the executable code could never do this anyway, but the abstract specification still needs to exclude the possibility explicitly.

5.4. Refining Real-Time Procedures and Calls

Having declared such a procedure, we can now refine its body to executable code. To do this the tool provides two simple opening rules that allow the programmer to focus on the procedure body within the declarative block. The first rule is trivial and merely opens on the 'in-context' statement within the block. The second is the general rule for opening a 'S in-context C' construct—it changes the focus to statement S and adds the local context C to the current one (also see Opening Rule 3 in Section 6.3 below). This makes the formal parameters part of the context while the programmer is refining the procedure body. From this point, refinement of the body to executable code proceeds by application of (timed) refinement rules [HU98].

More interesting are the rules for focussing on the statement within the scope of a procedure declaration and for introducing procedure calls. The following opening rule allows the programmer to focus on and refine a statement S_2 which is within the scope of a procedure p with formal parameter profile F.

Opening Rule 2 (Enter procedure block).

$$\frac{\begin{array}{l}(\textbf{inv}\ \text{hide}(p, I) \\ \ \textbf{lval}\ \text{hide}(p, L) \\ \ \textbf{pre}\ \text{hide}(p, P) \\ \ \textbf{lval}\ p(F) \in \text{Procedures} \\ \ \textbf{inv}\ p(F) \overset{\text{def}}{=} v\colon [P\ , Q]) \Rightarrow S_2 \sqsubseteq S_3\end{array}}{(\textbf{pre}\ P,\ \textbf{inv}\ I,\ \textbf{lval}\ L) \Rightarrow [\![\, \textbf{proc}\ p(F)\ \textbf{is}\ v\colon [P\ , Q] \sqsubseteq S_1 \bullet S_2 \,]\!] \sqsubseteq [\![\, \textbf{proc}\ p(F)\ \textbf{is}\ v\colon [P\ , Q] \sqsubseteq S_1 \bullet S_3 \,]\!]}$$

The rule is similar to Opening Rule 1. It adds the procedure's signature $p(F)$ to the context, and hides any global occurrences of name 'p'. Entering the procedure block is assumed not to involve any run-time overheads—these are incurred when the procedure is called. Importantly, the final '**inv**' hypothesis associates procedure p with its specification $v\colon [P\ , Q]$. This specification provides a 'contract' between the procedure and its caller, regardless of any subsequent refinements performed on the procedure body S_1 [Sta99]. Also note that variables free in the specification statement will typically include formal parameter names from F, so it is important to associate procedure name p with its parameter passing profile F in the new '**lval**' and '**inv**' hypotheses.

In the object-sizer case study, Opening Rule 2 is used to focus on the specification statement from Section 5.3. Doing so makes the definition of procedure *detect_chng* available in the context. To prepare for introduction of calls to this procedure, a series of refinement steps then introduces two local `Time`-valued variables, *riset* and *fallt*, which approximate the actual time at which the sensor signal rises and falls, respectively. The specification is then broken up into three statements, the first two of which are requirements to find these times, and the final statement is an assignment which performs the necessary arithmetic [HU98].

$$\begin{array}{l} \quad \mathit{size}\colon [\mathit{size}' \in (\mathit{speed} * (\mathit{falls} - \mathit{rises})) \pm \mathit{delta}] \\ \sqsubseteq [\![\, \textbf{var}\ \mathit{riset}, \mathit{fallt} : \texttt{Time} \bullet \\ \qquad \mathit{riset}\colon [\mathit{rises} \leqslant \mathit{riset}' \leqslant (\mathit{rises} + \mathit{error})]\ ; \\ \qquad \mathit{fallt}\colon [\mathit{falls} \leqslant \mathit{fallt}' \leqslant (\mathit{falls} + \mathit{error})]\ ; \\ \qquad \mathit{size} := \mathit{speed} * (\mathit{fallt} - \mathit{riset}) \\ \quad]\!] \end{array}$$

Clearly the two specification statements above have almost the same form as the specification of procedure *detect_chng*. The procedure's postcondition is stronger because it has an additional constraint on its finishing time τ, but this is an acceptable strengthening of the predicates above. Also the procedure requires precondition *signal_changes*, but this property is in the global '**pre**' context, so it is available to both of the specification statements above. Therefore, it just remains to replace the specification statements with calls to the procedure itself.

This is supported by a refinement rule for replacing a specification with a procedure call. The general form of the rule is given below, assuming the same parameter profile as used for Refinement Rule 2 above. Let *va*, *ra*, *vra* and *ca* be actual value, result, value-result and logical-constant parameter lists, respectively.

Refinement Rule 3 (Introduce procedure call).

$$\frac{\begin{array}{l}\textbf{lval}\ p(\textbf{val}\ \mathit{vf} : T_1, \textbf{res}\ \mathit{rf} : T_2, \textbf{valres}\ \mathit{vrf} : T_3, \textbf{con}\ \mathit{cf} : T_4) \in \text{Procedures} \\ \textbf{inv}\ p(\textbf{val}\ \mathit{vf} : T_1, \textbf{res}\ \mathit{rf} : T_2, \textbf{valres}\ \mathit{vrf} : T_3, \textbf{con}\ \mathit{cf} : T_4) \overset{\text{def}}{=} v\colon [P\ , Q] \\ \textbf{lval}\ \text{distinct-names}(\mathit{ra}, \mathit{vra}) \\ \textbf{inv}\ \text{idle-invariant}(U) \\ \textbf{inv}\ \text{pre-idle-invariant}(V) \\ \textbf{inv}\ \text{post-idle-invariant}(V) \\ \textbf{lval}\ \{\mathit{ra}, \mathit{vra}\} \subseteq w \\ \textbf{lval}\ v[\mathit{ra}, \mathit{vra}/\mathit{rf}, \mathit{vrf}] \subseteq w \\ \textbf{inv}\ \text{rtproc-pre-ob}(U, P) \\ \textbf{inv}\ \text{rtproc-call-ob}(U, Q, (w \setminus \{\mathit{ra}, \mathit{vra}\}), V)\end{array}}{w\colon [U\ , V] \sqsubseteq \textbf{call}\ p(\mathit{va}, \mathit{ra}, \mathit{vra}, \mathit{ca})}$$

The refinement below the line simply replaces a specification statement $w\colon [U\ , V]$ with a call to procedure p.

The numerous provisos above the line ensure that this is done in an appropriate environment, especially with respect to parameter passing. The first two provisos require that a procedure p with an appropriate formal parameter profile is in the context. Procedure p's behaviour is defined by specification statement $v\colon [P\ ,Q]$. The third proviso requires that the actual result ra and value-result vra parameter lists contain distinct identifiers, so that there is no aliasing. The next three provisos require idle invariance of the precondition U and postcondition V predicates in the specification statement being refined. This ensures that they allow for any run-time overheads associated with calling procedures, such as allocating and deallocating stack space and copying parameters. The seventh proviso ensures that the actual result ra and value-result vra parameter lists are in the frame w of the statement being refined, because the variables in these lists may be changed by the procedure.

Ideally, this rule would be completed by simply introducing the usual three provisos needed for a specification statement $w\colon [U\ ,V]$ to be refined by a specification statement $v\colon [P\ ,Q]$. These are that frame v is contained within frame w, that precondition U implies precondition P, and that postcondition Q implies postcondition V. However, we must take account of the fact that the procedure's specification $v\colon [P\ ,Q]$ is expressed in terms of the formal parameters, whereas the specification $w\colon [U\ ,V]$ representing the procedure call is expressed in terms of the actual parameters. Therefore, the remainder of the provisos in Refinement Rule 3 must be stated with respect to appropriate substitutions of actual parameters for formal ones.

The eighth proviso in Refinement Rule 3 therefore states that the frame v of the procedure's specification, with appropriate substitutions of result and value-result parameters, is a subset of the variables in the frame w of the statement being refined. This checks that the procedure will not change any variables not changed by the original requirement.

The final two provisos relate to the behavioural correctness of the refinement. Predicate 'rtproc-pre-ob' checks that the precondition of the procedure's body is not stronger than that of the requirement being refined. In essence, the condition to be proven is $U \Rightarrow P$. However, this must be done with appropriate substitutions of actual for formal parameters. The full definition of this predicate therefore includes declarations to make the values of actual value va, value-result vra and constant ca parameters visible, but makes no assumptions about actual result parameters ra since these are undefined in the procedure body's precondition P. The 'rtproc-pre-ob' predicate is also careful to ensure that 'τ' is defined to be the time at which the procedure body begins execution, rather than the time at which the call is made.

The 'rtproc-call-ob' predicate is similar but both initial and final values of variables and parameters must be considered. It aims to check that the procedure body's postcondition Q achieves the desired effect V, in situations where precondition U held initially. In essence, it proves condition $(U \wedge Q) \Rightarrow V$. Again, however, appropriate parameter substitutions and assumptions must be introduced. In particular, 'stable' conditions must be included for those variables $w \setminus \{ra, vra\}$ that cannot be changed by the procedure, but which were listed in the frame w of the calling statement.

Using this rule, in the context of procedure *detect_chng*'s declaration, we can now replace the requirements to detect changes in the light sensor's signal with calls to the procedure.

$$\begin{array}{l} \quad riset\colon [rises \leqslant riset' \leqslant (rises + error)] \\ \sqsubseteq \text{ `by Refinement Rule 3'} \\ \quad \textbf{call } detect\_chng(rises, \text{true}, riset) \end{array}$$

Similarly for the requirement to update *fallt*.

Thus, with the definition and implementation of these refinement and opening rules for procedures, we were able to successfully complete the object-sizer case study [HU98] using the Program Refinement Tool. Although our rules for manipulating procedures appear intimidating, due to their long lists of provisos, most of the conditions are trivial and the tool discharges them automatically, so the programmer can still apply the rules with ease. Also, our discovery of several minor errors in the published version of the object-sizer case study [HU98] bears out the benefit of tool support.

5.5. Discussion on Procedure Scoping

Introducing procedures to the Program Refinement Tool raised a number of interesting issues with respect to its handling of variables. Consider the following program fragment [Mac83, p. 114] which could occur during refinement of a specification '$v\colon [Q]$' to a procedure call '**call** p'.

```
⟦ var $v : T$ •                                  -- global declaration of variable '$v$'
  ⟦ proc $p$ is $v\colon [Q] \sqsubseteq \cdots v := E \cdots$ •   -- procedure body which updates variable $v$
    ⟦ var $v : T$ •                              -- local declaration of variable '$v$'
      call $p$                                   -- procedure call (which is a refinement of specification $v\colon [Q]$)
        ⋮
```

Variable v occurs 'free' in the body of procedure p, i.e., the variable is global to the procedure. Traditionally, this raises the issue of *which* variable v is modified by the call to procedure p. Good programming language design favours binding of free identifiers in procedure bodies at the point of declaration (static binding), rather than at the point of call (dynamic binding), as this is easier to understand, and means that the effect of calling a procedure is independent of the point at which it is called [Mac83, §3.3]. Unfortunately, the scoping rules of the standard refinement calculus' modelling language mean that procedure calls are assessed in the scope in which they occur, rather than the scope in which the procedure was declared, thus resulting in dynamic binding [Sam97]. Programmers using some refinement formalisms are therefore warned that they must apply appropriate renaming to achieve the right effect [Nau01].

The Program Refinement Tool avoids this problem altogether, thanks to its maintenance of unambiguous variable contexts. Refinement of procedure p's body occurs in the context of the global declaration of v, thus supporting static binding, as desired. Furthermore, to develop the above code fragment, the programmer needs to focus the tool on specification '$v\colon [Q]$', which occurs in the context of the local declaration of v, with the intention of refining it to '**call** p'. This is done using Opening Rule 1 from Section 4.4. In the situation shown above, the 'hide' operations in that rule remove all hypotheses that refer to the global variable v, including the '**lval**' hypothesis that defines 'p' as a procedure whose body contains a free reference to 'v'. In effect, this makes the procedure declaration inaccessible, and prevents the programmer from performing the ambiguous refinement step. (More generally, this also avoids the danger of procedures having unanticipated side effects on global variables—a refinement step to introduce a call to a procedure with side effects is precluded unless the global variables are listed in the frame of the specification statement being refined.)

As well as ambiguous updates to multiply-declared variables, a similar problem for block-structured programming languages is ambiguous calls to multiply-declared procedures.

```
⟦ proc $p$ is $\cdots$ •                          -- global declaration of procedure '$p$'
  ⟦ proc $q$ is $\cdots \sqsubseteq \cdots$ call $p \cdots$ •   -- declaration of procedure $q$ which calls procedure $p$
    ⟦ proc $p$ is $\cdots$ •                      -- local declaration of procedure '$p$'
      call $q$
        ⋮
```

When the call is made to procedure q, it is not obvious which procedure p is subsequently invoked, the one visible at the point where procedure q was declared, or the one visible where q was called. Once again, conventional programming wisdom says that static binding is the best alternative, so the call should be to the 'global' version of p, and the Program Refinement Tool's opening rule for procedure blocks achieves this through manipulation of procedure declarations in contexts in the same way as for variables above.

Recursive procedures are rarely used in embedded, real-time programs due to the difficulties in establishing their timing characteristics and their unpredictable memory requirements. We have therefore ignored recursively defined procedures so far. However, we note that the scoping mechanism presented above is adequate for the definition of recursive procedures even in the face of nested redeclaration. Since the '**lval**' context stores procedure specifications, introduction of a recursive call can be achieved by the development of a matching specification within the procedure body.

Finally, we note that procedure specifications are stored as constants (they are referentially transparent) in the Program Refinement Tool's context. However, in contrast to Naumann's model of procedures [Nau01], neither assignment to procedure-valued variables nor passing of procedures as parameters is supported by the tool. The main benefit of storing procedure specifications is their use as semantic targets for the introduction of procedure calls. In particular, the frame of the procedure's specification is useful in determining the validity of the call with respect to the scope of the effect of the call. Thus our approach achieves the goal of static binding without the additional complexity of being truly higher-order.

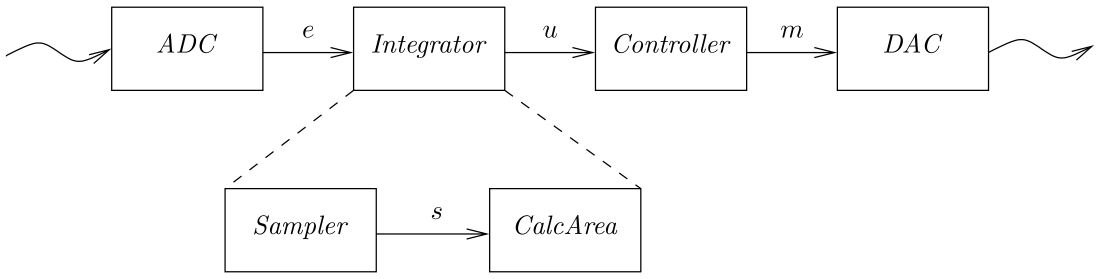

Fig. 3. Assumed environment for the integrator process's refinement.

6. Concurrent Variables in Parallel Real-Time Refinement

So far we have concentrated on refinement to sequential and modular program code. Recent research on the Program Refinement Tool has also explored its extension to handle parallel programs. In this section we use another case study to explain how this is handled.

6.1. The Integrator Case Study

To experiment with parallel processes with timing constraints, we chose a case study from the domain of discrete real-time control theory [FP80, p. 14]. As shown in Fig. 3, the aim is to refine one component of a closed-loop (feedback-driven) control system. On the left, an analogue-to-digital converter, *ADC*, produces a signal e that represents the measured error between the desired and actual system output. The *Integrator* component calculates a trapezoidal approximation u to the integral of this error. Control engineers use approximations to the integral to eliminate steady-state error [Bis94, p. 185]. This information is then used by the *Controller* to calculate the manipulated control signal m. This exits on the right through a digital-to-analogue converter, *DAC*, and into the plant to be controlled, from which sensors will return the new measured error via a feedback loop (not shown).

Each of the components is assumed to be iterative, with a common period of duration D. The signals are modelled as vectors, one element for each period, where v_k denotes vector v's k^{th} value. The *Integrator* component is required to produce a signal that satisfies the following control equation [FP80, p. 14].

$$u_k = u_{k-1} + \frac{D}{2}(e_k + e_{k-1})$$

The specific refinement to be performed is to split this requirement into two parallel processes. The *Sampler* process reads the error input e and stores the sample in a vector s. The *CalcArea* process uses this to calculate the area under function s, and puts the result in output vector u. A shared-memory multi-processor architecture is assumed—while the *Sampler* process is reading and storing the k^{th} sample, the *CalcArea* process is simultaneously accessing the $(k-1)^{\text{th}}$ sample. Mutual exclusion is enforced by correct timing behaviour only.

6.2. Refining Parallel Components

Each of the components in Fig. 3 represents a parallel process, with its own thread of control and its own state space. Introducing parallel processes to the timed refinement calculus is still a research topic [PH02]. It has proven difficult to find a definition of parallel composition, in the predicate-transformer semantics used by refinement calculi, that is well behaved with respect to extreme processes such as those that perform 'aborting' computations [Smi01]. Nevertheless, definitions that are satisfactory for the majority of process specifications have been available for some time [Mah92, Mah95, Mah94], so we used these to implement corresponding rules in the Program Refinement Tool.

At the beginning of the integrator case study, the following context for the *Integrator* component is created.

lval $e \in$ Inputs
inv $e : \texttt{Time} \rightarrow \texttt{Int}$
lval $u \in$ Outputs
inv $u : \texttt{Time} \rightarrow (\texttt{Nat} \rightarrow \texttt{Real})$
pre $\tau \leqslant -D$

Here error input e is modelled as a timed trace of integers. The calculated integral is modelled as a timed-trace consisting of a vector of real values. Using a vector allows one value to be written while its predecessor is being read. (In the final implementation a small circular buffer would be sufficient for this purpose, but using the unbounded vector above simplifies the top-level specifications.) The '**pre**' predicate tells us that the whole system starts a full period before time 0, which gives enough time for the pipeline of processes to be properly 'primed'.

Within this context, the integrator's requirement is given by the following specification statement.

$$u\colon [\mathit{ADC}\ , \mathit{Integrator}]$$

The ADC predicate defines the assumed behaviour of the error input e. For some number r, let $\lfloor r \rfloor$ be the largest integer no greater than r, i.e., r rounded down to the nearest integer.

$$\mathit{ADC} \stackrel{\text{def}}{=} \forall t : \texttt{Time} \bullet e(t) = e(\lfloor t/D \rfloor * D)$$

In other words, the output from the analogue-to-digital converter is modelled by an arbitrary signal which is stable throughout each period of duration D, i.e., it is a step function [FP80, p. 3]. The integrator's requirement is then to produce the area under this function. To allow for the time needed to sample values and perform the necessary arithmetic, the requirement states that the n^{th} value of vector u does not need to be made available to the *Controller* process until time $(n + 2) * D$.

$$\begin{aligned} \mathit{Integrator} \stackrel{\text{def}}{=} \forall n : \texttt{Nat}, t : \texttt{Time} \bullet \\ &(n + 2) * D \leqslant t \Rightarrow \\ &\quad (u(t))(n) = \textstyle\sum_{1 \leqslant i \leqslant n} \frac{D}{2} * (e(i * D) + e((i - 1) * D)) \end{aligned}$$

For brevity, we have expressed the *Integrator* predicate over all natural numbers n above. In fact, the actual case study performed in the Program Refinement Tool was slightly more complicated and placed an upper bound on the number of periods. (Technically, the predicate above prevents the specification statement from ever terminating, which is handled in the timed refinement calculus by a distinguished specification statement [Hay02], but we ignore this complication here.) As is typical for high-level specifications of whole process behaviours, all references to timed-trace variables such as u and e in both of the above predicates are explicitly indexed with absolute times—there are no free references to starting time τ_0 or finishing time τ.

To partition such a specification into parallel components, a refinement rule for introducing parallel composition is needed. The situation illustrated in Fig. 3 is a special case where all information flow is from left to right. Therefore, the rule needed is one which introduces an asymmetric *piping* operator, denoted '$\gg$', which is formally defined as functional composition of predicate transformers [Mah92, §4.2].

Refinement Rule 4 (Introduce pipe).

$$\begin{array}{l}
\textbf{lval } v_1 \subseteq \text{Locals} \\
\textbf{inv } v_1 : \texttt{Time} \rightarrow T \\
\textbf{lval } v_2 \subseteq (\text{Locals} \cup \text{Outputs}) \\
\textbf{lval } v_1 \cap v_2 = \varnothing \\
\textbf{lval } v_2 \text{ not-free-in } Q_1 \\
\textbf{lval } \tau \text{ not-free-in } Q_1 \\
\textbf{lval } \tau_0 \text{ not-free-in } Q_1 \\
\hline
v_1, v_2 \colon [P \mathbin{,} Q_1 \wedge Q_2] \sqsubseteq ((v_1 \colon [P \mathbin{,} Q_1] \text{ in-context } C_1) \gg (v_2 \colon [Q_1 \mathbin{,} Q_2] \text{ in-context } C_2))
\end{array}$$

where $C_1 \overset{\text{def}}{=} (\textbf{lval } v_1 \subseteq \text{Outputs}$
$\quad \textbf{inv } v_1 : \texttt{Time} \rightarrow T)$
and $C_2 \overset{\text{def}}{=} (\textbf{lval } v_1 \subseteq \text{Inputs}$
$\quad \textbf{inv } v_1 : \texttt{Time} \rightarrow T)$

The refinement below the line allows a specification statement to be split into two statements connected by the piping operator. The list of variables 'v_1, v_2' to be updated is divided among the parallel components, as are the two conjuncts Q_1 and Q_2 that define the overall desired effect. The precondition P of the whole system is assumed to define the environment of the left-hand component, whereas the effect Q_1 of the left-hand component becomes the precondition of the right-hand one. Most importantly, the two parallel processes each carry their own context. The left-hand component constructs variables in list v_1, so these are outputs in its context C_1, whereas the right-hand component views these variables as inputs.

The first two provisos say that the variables in list v_1 are expected to be local timed-traces of type T. The third proviso requires the variables v_2 updated by the second parallel component to be either local or output variables. The fourth proviso merely requires lists v_1 and v_2 to be distinct. The fifth proviso requires that postcondition predicate Q_1, belonging to the left-hand component, does not rely on variables v_2, constructed by the right-hand component, in keeping with the asymmetry of the piping operator. The final provisos ensure that predicate Q_1, which must be suitable for use as both a precondition and a postcondition, does not rely on the current time variable τ, which has a different meaning in these different situations.

This rule is used in the integrator case study as part of a sequence of refinement steps that first introduces the new variable s that connects the two parallel components, and then splits the requirement in half. Firstly, let *Sampler* be a predicate defining the intended behaviour of the process that samples and stores values from the analogue-to-digital converter.

$$\begin{array}{l}
\mathit{Sampler} \overset{\text{def}}{=} \forall n : \texttt{Nat}, t : \texttt{Time} \bullet \\
\qquad ((n+1) * D \leqslant t) \Rightarrow (s(t))(n) = e(n * D)
\end{array}$$

In other words, the sampling component is required to have stored the nth sample by time $(n+1) * D$. The overall refinement then proceeds as follows.

$$\begin{array}{l}
\quad u \colon [\mathit{ADC} \mathbin{,} \mathit{Integrator}] \\
\sqsubseteq \text{ 'By Refinement Rule 1'} \\
\quad [\![\, \textbf{var } s : \texttt{Nat} \rightarrow \texttt{Int} \bullet \\
\qquad s, u \colon [\mathit{ADC} \mathbin{,} \mathit{Integrator}] \\
\quad]\!] \\
\sqsubseteq \text{ 'By strengthening the postcondition [HU01, Law 7]'} \\
\quad [\![\, \textbf{var } s : \texttt{Nat} \rightarrow \texttt{Int} \bullet \\
\qquad s, u \colon [\mathit{ADC} \mathbin{,} \mathit{Sampler} \wedge \mathit{Integrator}] \\
\quad]\!] \\
\sqsubseteq \text{ 'By Refinement Rule 4'} \\
\quad [\![\, \textbf{var } s : \texttt{Nat} \rightarrow \texttt{Int} \bullet \\
\qquad (s \colon [\mathit{ADC} \mathbin{,} \mathit{Sampler}] \text{ in-context } C_1) \gg (u \colon [\mathit{Sampler} \mathbin{,} \mathit{Integrator}] \text{ in-context } C_2) \\
\quad]\!]
\end{array}$$

$$\begin{aligned}\text{where } C_1 &\overset{\text{def}}{=} (\textbf{lval}\ s \in \text{Outputs}\\ &\qquad \textbf{inv}\ s : \texttt{Time} \to (\texttt{Nat} \to \texttt{Int}))\end{aligned}$$

$$\begin{aligned}\text{and } C_2 &\overset{\text{def}}{=} (\textbf{lval}\ s \in \text{Inputs}\\ &\qquad \textbf{inv}\ s : \texttt{Time} \to (\texttt{Nat} \to \texttt{Int}))\end{aligned}$$

Each of the two parallel processes thus created carries its own view of local timed-trace variable s. The left-hand one considers it to be an output, while the right-hand one views it as an input.

6.3. Opening on Parallel Processes

The case study continues by refining the two individual processes. In particular, the right-hand process has yet to make any profitable use of variable s. Its postcondition predicate, *Integrator*, does not even mention s—it defines output u directly in terms of input e. However, given that the *Sampler* precondition tells us that local variable s is defined in terms of e, it is now possible to strengthen the *Integrator* predicate to a '*CalcArea*' one that redefines u in terms of s, and does not mention e at all.

To do this, the programmer needs to focus on the right-hand process. Firstly, Opening Rule 1 from Section 4.4 allows the programmer to open a window within the declaration of s. A trivial opening rule then allows us to focus on the second component in a '$\gg$' construct—the rule makes no changes to the context because each parallel processes is assumed to carry its own context. Next, the following simple rule is used to focus within the specific 'in-context' construct of interest.

Opening Rule 3 (Enter input context).

$$\frac{\begin{array}{l}(\textbf{inv}\ I\\ \ \textbf{lval}\ \text{hide}(v, L)\\ \ \textbf{pre}\ P\\ \ \textbf{lval}\ v \in \text{Inputs}) \Rightarrow S_1 \sqsubseteq S_2\end{array}}{\begin{array}{l}(\textbf{pre}\ P,\ \textbf{inv}\ I,\ \textbf{lval}\ L) \Rightarrow ((S_1 \text{ in-context } (\textbf{lval}\ v \in \text{Inputs},\ \ \textbf{inv}\ v : \texttt{Time} \to T)) \sqsubseteq\\ \qquad\qquad\qquad\qquad\qquad (S_2 \text{ in-context } (\textbf{lval}\ v \in \text{Inputs},\ \ \textbf{inv}\ v : \texttt{Time} \to T)))\end{array}}$$

In effect, the context above the line changes any existing classification of variable v to 'input', by hiding its occurrences in hypothesis list L and adding a new '**lval**' hypothesis. However, existing '**inv**' and '**pre**' hypotheses concerning v are left unchanged because these may contain useful information about the variable's behaviour.

Via this rule it is thus possible to focus on the right-hand process in the integrator case study, and perform the necessary strengthening of the *Integrator* predicate to a *CalcArea* one that uses vector s. The remainder of the case study then consists of (timed) refinement steps to derive executable statements from the *Sampler* and *CalcArea* components. While intricate, due to the need to keep track of which values must be accessible in particular periods, these steps are relatively straightforward applications of the timed refinement calculus rules [HU01].

7. Conclusion

We have described recent extensions to the way an automated programming tool manages and manipulates a wide variety of 'variables', including constants, procedure parameters, and input/output streams, during formal development of real-time programs. By automatically maintaining a context of which variables are in scope, and their types, the tool removes much of the tedium of rigorously applying refinement steps. Its automatic tactics allow proof obligations concerning type compatibility of real-time variables and expressions, stability of real-time variables, and invariance of real-time predicates, to be discharged without user intervention. The tool thus allows each refinement step to be fully verified, whereas the equivalent 'paper' proofs must inevitably skip much of this fine detail, and therefore risk overlooking errors.

The computer-aided refinements mentioned above differ in some technical details from the handcrafted versions that inspired them. An obvious difference is that the tool uses a machine-readable ASCII representation of specifications, rather than the usual mathematical symbols. This makes the tool's refinements appear much clumsier. (For readability, we have used mathematical notation throughout this paper.) Also, some refinement laws are implemented differently in the tool. Most notable is the way that zero-subscripted constants must be introduced by an explicit step, whereas the conventional refinement calculus [Mor90] assumes that this is done implicitly.

Ongoing enhancements to the real-time refinement calculus have addressed reasoning about non-terminating loops [Hay02] and improved the modelling of parallel processes [PH02], and it is fully expected that corresponding extensions can also be made to our real-time Program Refinement Tool.

Finally, the Program Refinement Tool, like all theorem prover-based software, remains complex and intimidating to use. Even with tool support, rigorously performing refinements is currently practical for small fragments of highly critical code only. However, future research aims to simplify the tool further, by improving the user interface and increasing the degree of automation through additional theories and tactics.

Acknowledgements

We wish to thank Geoffrey Watson for correcting errors in a draft of this article. An earlier version of Sections 2 to 4 of this paper was presented at the REFINE 2002 workshop in Copenhagen [WF02]. Preparation of this paper was funded in part by Australian Research Council Large Grant A00104650, *Verified Compilation Strategies for Critical Computer Systems.*

References

[BGL+97] 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.

[Bis94] C. C. Bissell. *Control Engineering*, volume 15 of *Tutorial Guides in Electronic Enginering.* Chapman and Hall, 1994. Second edition.

[BvW98] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[CHN+98] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A program refinement tool. *Formal Aspects of Computing*, 10(2):97–124, 1998.

[CSW99] A. Cavalcanti, A. Sampio, and J. Woodcock. An inconsistency in procedures, parameters, and substitution in the refinement calculus. *Science of Computer Programming*, 33(1):87–96, January 1999.

[FP80] G. F. Franklin and J. D. Powell. *Digital Control of Dynamic Systems.* Addison-Wesley, 1980.

[Gro96] L. Groves. Procedures in the refinement calculus: A new approach? 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/`.

[Hay98] 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. Invited paper.

[Hay00] I. J. Hayes. Real-time program refinement using auxiliary variables. In M. Joseph, editor, *Sixth International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2000)*, volume 1926 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, 2000.

[Hay02] I. J. Hayes. Reasoning about real-time repetitions: Terminating and nonterminating. *Science of Computer Programming*, 43(2–3):161–192, May/June 2002.

[Hes99] W. H. Hesselink. Predicate transformers for recursive procedures with local variables. *Formal Aspects of Computing*, 11(6):616–636, 1999.

[Hoo96] 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.

[HU98] 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.

[HU01] I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.

[JLM+94] D. T. Jordan, C. J. Locke, J. A. McDermid, C. E. Parker, B. A. P. Sharp, and I. Toyn. Literate formal development of Ada from Z for safety critical applications. In *Proc. SafeComp'94*. ISA, 1994.

[Lai00] L. Laibinis. *Mechanised Formal Reasoning About Modular Programs.* PhD thesis, Department of Computer Science, Åbo Akademi University, 2000.

[Mac83] B. J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* Holt, Rinehart and Winston, 1983.

[Mah92] B. P. Mahony. *The Specification and Refinement of Timed Processes.* PhD thesis, Department of Computer Science, University of Queensland, 1992.

[Mah94] B. Mahony. Using the refinement calculus for dataflow processes. Technical Report 94-32, Software Verification Research Centre, October 1994.

[Mah95] B. Mahony. Networks of predicate transformers. Technical Report 95-5, Software Verification Research Centre, February 1995.

[MF01] A. P. Martin and C. J. Fidge. Lifting in Z. In C. J. Fidge, editor, *Computing: The Australasian Theory Symposium 2001*, volume 42 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001. `http://www.elsevier.nl/locate/entcs`.

[MH92] 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.

[MNU97] A. Martin, R. Nickson, and M. Utting. A tactic language for Ergo. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, pages 186–207. Springer, 1997.

[Mor90] C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[Nau01] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Science of Computer Programming*, 41(1):1–51, September 2001.

[NH97] R. Nickson and I. J. Hayes. Supporting contexts in program refinement. *Science of Computer Programming*, 29(3):279–302, 1997.

[NTU96] R. Nickson, O. Traynor, and M. Utting. Cogito Ergo Sum: Providing structured theorem prover support for specification formalisms. *Australian Computer Science Communications*, 18(1):149–158, February 1996.

[NU95] R. Nickson and M. Utting. A new face for Ergo: Adding a user interface to a programmable theorem prover. In H. Hasan and C. Nicastri, editors, *HCI, A Light into the Future: Proceedings of OZCHI'95*, pages 204–209. Ergonomics Society of Australia Inc., November 1995.

[PH02] S. Peuker and I. Hayes. Towards a refinement calculus for concurrent real-time programs. In C. George and H. Miao, editors, *Formal Methods and Software Engineering (ICFEM'02)*, volume 2495 of *Lecture Notes in Computer Science*, pages 335–346. Springer-Verlag, 2002.

[Sam97] A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing.* World Scientific, 1997.

[Sha93] B. Sharp. A basic guide to Zeta. Draft, York Software Engineering Ltd, December 1993.

[Smi01] G. Smith. Introducing parallel composition to the timed refinement calculus. In H. El Gindy and C. J. Fidge, editors, *PART 2000: Proceedings of the 7th Australasian Conference on Parallel and Real-Time Systems*, pages 139–148. Springer-Verlag, 2001.

[Sta99] M. Staples. *A Mechanised Theory of Refinement.* PhD thesis, Cambridge University, 1999.

[TM95] I. Toyn and J. A. McDermid. CADiZ: An architecture for Z tools and its implementation. *Software—Practice & Experience*, 25(3):305–330, March 1995.

[UF96] 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/`.

[UNT96] M. Utting, R. Nickson, and O. Traynor. Theory structuring in Ergo 4.1. In M. E. Houle and P. Eades, editors, *Proc. Computing: The Australasian Theory Symposium (CATS'96)*, pages 137–146, Melbourne, January 1996.

[Vic90] T. Vickers. An overview of a refinement editor. In *Proceedings of the Fifth Australian Software Engineering Conference (ASWEC'90)*, pages 39–44. IREE, 1990.

[WF02] L. Wildman and C. J. Fidge. The variety of variables in computer-aided real-time programming. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *Refine 2002: Proceedings of the BCS/FACS Refinement Workshop*, volume 70(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. Invited paper.

[WFC00] L. Wildman, C. J. Fidge, and D. A. Carrington. Computer-aided development of a real-time program. *Software—Concepts & Tools*, 19(4):190–202, August 2000.

[WH98] 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.