SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-12

Adding Real Time to Formal Program Development

Colin Fidge

June 1994

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Adding Real Time to Formal Program Development

Colin Fidge
Software Verification Research Centre,
Department of Computer Science,
The University of Queensland,
Queensland 4072, Australia

Abstract

Rules are presented for efficiently adding real-time requirements to existing functional program refinements.

1 Introduction

Existing refinement rules allow us to formally derive a program from its functional specification. However non-functional requirements, such as timeliness, reliability, security, etc., cannot be handled.

A number of recent proposals have suggested ways of adding "hard" real-time requirements to the refinement process. Unfortunately the inherent complexity of analysing timing behaviour either made the proof obligations very complex, or led to unrealistic simplifying assumptions.

Here we improve the ease and efficiency with which "real-time refinements" can be performed, while retaining a realistic timing model. To do this we define rules that directly build on existing functional refinements by adding time in a separate step. This allows us to divorce, as far as possible, the discharge of functional and timing obligations. Nevertheless, situations in which the timing behaviour is intimately linked to functional behaviour can still be accommodated.

Some knowledge of program refinement and the specification language Z is assumed.

To appear in *Proc. Formal Methods Europe '94*, Barcelona, Spain, 24–28 October 1994.

2 Real-time Refinement Rules

2.1 Definitions and Notation

Based on the works of Potter et al. and Wordsworth, we express refinements using Z as the requirements language and the guarded command language as the target programming language [16, 18].

For functional specifications let schema $PSpace$ contain declarations of the variables in the "program space" [18].

For timing specifications assume a discrete linear model for absolute time, i.e.,

$$AbsTime == \mathbb{N} \ ,$$

and let an auxiliary variable,

$$Time \mathrel{\widehat=} [\, now : AbsTime \,] \ ,$$

denote the current absolute time. This represents the moment at which the state-changes defined by each operation become observable. In proper use time may not go backwards, i.e.,

$$\Delta Time \mathrel{\widehat=} [\, Time; \ Time' \mid now' \geqslant now \,]$$

and

$$\Xi Time \mathrel{\widehat=} [\, \Delta Time \mid now' = now \,] \ .$$

(Such a simple model for time is adequate as long as all timing expressions measure time from the same source.)

In the following rules let S, S_1, S_2,... represent "functional" operation specifications of the form $[\, \Delta PSpace \mid \cdots]$, and let s, s_1, s_2,... denote predicate schemata of the form $[\, PSpace \mid \cdots]$. We also use G, G_1, G_2,... to denote predicate schemata when the predicates represent boolean guards that are directly expressible in the target language.

Similarly, let T, T_1, T_2,... represent "timed" operation specifications of the form $[\, \Delta PSpace; \ \Delta Time \mid \cdots]$, and let t, t_1, t_2,... denote timed predicates of the form $[\, PSpace; \ Time \mid \cdots]$.

To reason about the timing behaviour of high-level language code the primitive timing properties of the target architecture and environment are needed; see Appendix A for the particular values used herein. For instance, the time required to load the value of an integer variable is denoted lvar, and the time to take a conditional branch is cbry. We also use the same font to represent the time required to evaluate expressions appearing in the target language. For example, E denotes

the time required to evaluate a target code expression equivalent to Z expression E.

Since such timing information often contains a degree of uncertainty these primitives actually denote non-empty *sets* of such times, representing all *possible* execution times. (Although worst-case execution times are normally the main concern we have found that recording all possible times can provide hints for potential optimisations.) Define such sets as

$$ExTimes == \mathbb{P}_1 \mathbb{N}$$

and a "set addition" operator, denoted $\dot{+}$ or $\dot{\sum}$, to add two such sets by adding every pair of elements, i.e.,

$$_ \dot{+} _ : ExTimes \times ExTimes \rightarrow ExTimes$$

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

For example,

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

Notice that adding two "uncertain" timings results in an even broader range of possible execution times, as we would expect.

2.2 Rules for Adding Real-time Requirements

This section presents rules for adding real-time requirements to conventional refinement steps.

Timing specifications can refer to variables in *PSpace*, so that data-dependent timing behaviour can be expressed. In the following rules, however, care must be taken when adding a real-time requirement T to a functional specification S so that T does not inadvertantly contradict or in some other way constrain S. Formally, hiding the time variables in the conjoined schemata must return the original functional specification, i.e.,

$$(S \wedge T) \setminus (now, now') = S \ .$$

This obligation is required whenever any timing schema is added to a functional one below.

To account for the overheads associated with compound program statements the rules frequently need to state that some timing predicate t was expected to have held *before* the occurrence of some "time-changing" action T. Let

$$t \ before \ T \stackrel{\text{def}}{=} (\text{post}(t \wedge T))[_/_']$$

$$\text{if} \quad S \sqsubseteq x := E$$

$$\text{then} \quad \begin{array}{|l} \hline T \\ S;\ \Delta Time \\ \hline t \\ now' - now \in \mathsf{gadd} \dotplus \mathsf{E} \dotplus \mathsf{stor} \\ \hline \end{array} \sqsubseteq x := E$$

Figure 1: Adding timing constraints to assignment-introduction.

be a predicate that defines possible states resulting from performing T in an initial state satisfying t. This is used in defining timing pre-conditions, so notation $[_/_']$ represents a systematic renaming to remove primes [18].

Similarly, let

$$t' \ \mathit{after} \ T \stackrel{\text{def}}{=} (\text{pre}(T \wedge t'))'$$

be a predicate that defines possible states from which predicate t' can be reached *after* the passage of time specified by T. This is used in defining timing post-conditions.

2.2.1 Assignment.

Our "unit" rule is that for assignment (Fig. 1). It states that if some functional specification S has already been shown to be refinable into an assignment statement $x := E$, then a real-time specification T can be refined into the same assignment, as long as the specified duration of T is compatible with the calculated timing behaviour of the assignment code. Predicate t may express a timing pre-condition (and thus, via the duration, an implicit post-condition).

In a timing schema $now' - now$ represents the possible durations of the operation. Expression $\mathsf{gadd} \dotplus \mathsf{E} \dotplus \mathsf{stor}$ denotes the possible execution times of this *particular* target language assignment statement, i.e., the time required to get the address of the destination variable x, plus the time required to evaluate the expression E, plus the time to store the result in x.

2.2.2 Sequence.

When defining rules for structured statements we must account for the overheads of their implementation. We therefore define timing schemata to represent such actions. In the case of sequential composition there is usually no timing overhead associated with the ';' operator so we define

$$seq \mathrel{\widehat=} [\Xi PSpace;\ \Xi Time]$$

$$\text{if} \quad S \sqsubseteq S_1 \,;\, S_2$$

$$\text{then} \quad \begin{array}{|l} \hline T \\ S;\ \Delta Time \\ \hline t_1 \\ t_2' \\ \hline \end{array} \sqsubseteq \begin{array}{|l} \hline T_1 \\ S_1;\ \Delta Time \\ \hline t_1 \\ t_3' \\ \hline \end{array} \;;\; \begin{array}{|l} \hline T_2 \\ S_2;\ \Delta Time \\ \hline t_3 \textit{ before seq} \\ t_2' \\ \hline \end{array}$$

Figure 2: Adding timing constraints to sequence-introduction.

to denote this. (This may not be true in a geographically distributed system!)

Figure 2 then shows how timing requirements can be added to a refinement into two sequential components. The timing requirement is expressed as pre and post-conditions t_1 and t_2' (the post-condition can refer to initial values via logical constants, if necessary). On the right-hand side an intermediate timing requirement t_3 is introduced. It is expected that t_3' will be true in the post-state of T_1, and from this we can calculate the necessary pre-condition for T_2 by accounting for the overhead of sequential composition, if any (in the case where *seq* has no effect on time t_3 *before seq* becomes merely t_3). Inclusion of $\Delta Time$ means that an unsatisfiable choice for t_3 would be disallowed because $now' \geqslant now$ would be violated if time had to go backwards in T_1 or T_2.

We can easily imagine an equivalent, symmetric form of this rule in which the timing pre-condition of T_2 is t_3 and the post-condition of T_1 is t_3' *after seq*.

2.2.3 Alternatives.

The overheads associated with alternative statements are significantly more complex. Here we assume *deterministic* evaluation of guards until a true one is found, in order of their textual appearance. (Elsewhere we have given definitions for the nondeterministic case [4].)

For a *particular* alternative statement, with alternatives indexed by $i \in I$, define the overhead associated with reaching the ith alternative as

$$if_i \mathrel{\widehat{=}} \big[\, \Xi PSpace;\ \Delta Time \;\big|\; now' - now \in (\dot{\sum}_{a<i}(\mathsf{G}_a \dotplus \mathsf{cbry})) \dotplus (\mathsf{G}_i \dotplus \mathsf{cbrn}) \,\big] \ .$$

In other words, to reach the ith alternative we must evaluate all preceding false guards G_a and conditionally branch past them, and evaluate true guard G_i and decide not to conditionally branch.

Having executed an alternative we are obliged to branch around the code for other alternatives, except in the case where the chosen alternative was the last

if $\quad S \sqsubseteq \mathbf{if}\ ([]_{i \in I} \bullet G_i \rightarrow S_i)\ \mathbf{fi}$

then

$$\begin{array}{|l} \hline T \\ S;\ \Delta Time \\ \hline t_1 \\ t_2' \\ \hline \end{array} \sqsubseteq \mathbf{if}\ ([]_{i \in I} \bullet G_i \rightarrow \begin{array}{|l} \hline T_i \\ S_i;\ \Delta Time \\ \hline t_1\ \mathit{before}\ \mathit{if}_i \\ t_2'\ \mathit{after}\ \mathit{fi}_i \\ \hline \end{array})\ \mathbf{fi}$$

Figure 3: Adding timing constraints to alternatives-introduction.

one. Define this overhead as

$$\mathit{fi}_i \mathrel{\widehat{=}} [\Xi PSpace;\ \Delta Time \mid (i < max\ I \Rightarrow now' - now \in \mathsf{br}) \wedge (i = max\ I \Rightarrow now' = now)] \ .$$

Based on these definitions the rule for adding real-time constraints to alternatives-introduction is shown in Fig. 3. From the pre and post-conditions t_1 and t_2' we can determine the timing constraints for each alternative in terms of the overheads associated with choosing that alternative. For the ith alternative we expect that condition t_1 was true before the passage of time defined by if_i, so the timing pre-condition for the alternative body itself is t_1 *before* if_i. Similarly, the post-condition t_2' is expected to hold only after adding the overhead of branching out of the statement, so the timing post-condition is t_2' *after* fi_i. As before, if the timing conditions on any of the alternatives are such that time must go backwards then the refinement is incorrect.

2.2.4 Iteration.

For a *particular* iteration statement define

$$do^{\text{loop}} \mathrel{\widehat{=}} [\Xi PSpace;\ \Delta Time \mid now' - now \in \mathsf{G} \dotplus \mathsf{cbrn}]$$

as the overhead associated with reaching the loop body at each iteration, i.e., the time required to evaluate the (true) guard G and decide not to branch. When the guard becomes false, however, there is still an overhead associated with evaluating G and branching out of the iterative statement. Define this as

$$do^{\text{exit}} \mathrel{\widehat{=}} [\Xi PSpace;\ \Delta Time \mid now' - now \in \mathsf{G} \dotplus \mathsf{cbry}] \ .$$

Finally, define

$$
\text{if}\quad
\begin{array}{|l}
\hline
\multicolumn{1}{l}{S} \\
\Delta[\rho 1];\ \Xi[\rho 2] \\
\hline
\Delta s \\
\neg\, G' \\
\hline
\end{array}
\ \sqsubseteq\ \mathbf{do}\ G \rightarrow
\begin{array}{|l}
\hline
\multicolumn{1}{l}{S_1} \\
\Delta[\rho 1];\ \Xi[\rho 2] \\
\hline
\Delta s \\
S_2 \\
\hline
\end{array}
\ \mathbf{od}
$$

$$
\text{then}\quad
\left(
\begin{array}{|l}
\hline
\multicolumn{1}{l}{T} \\
S;\ \Delta \mathit{Time} \\
\hline
\Delta t \\
\hline
\end{array}
\ \fatsemi\ \mathit{do}^{\mathrm{exit}}\right)
\ \sqsubseteq\ \mathbf{do}\ G \rightarrow
\begin{array}{|l}
\hline
\multicolumn{1}{l}{T_1} \\
S_1;\ \Delta \mathit{Time} \\
\hline
t\ \mathit{before}\ \mathit{do}^{\mathrm{loop}} \\
t'\ \mathit{after}\ \mathit{od} \\
\hline
\end{array}
\ \mathbf{od}
$$

Figure 4: Adding timing constraints to iteration-introduction.

$$\mathit{od} \mathrel{\widehat{=}} [\, \Xi \mathit{PSpace};\ \Delta \mathit{Time} \mid \mathit{now}' - \mathit{now} \in \mathsf{br} \,]$$

as the time needed to branch back to the beginning of the loop each time the loop body finishes. (Again, the definitions make implicit assumptions about object code generation. Alternative definitions must be made if a different compilation strategy is anticipated, e.g., putting guard evaluation at the end of the loop body.) We then define a rule for adding real time constraints to a refinement step that introduces iteration as shown in Fig. 4.

The functional part is based on that of Potter et al. [16, §10.7]. Following their concise notation we have taken the liberty of allowing 'Δ' and 'Ξ' to precede schemata where the intention is clear. Let ρ_1 and ρ_2 be lists of declarations representing those variables changed and unchanged by the loop, respectively; together these two lists equal *PSpace*. Predicate s is the loop invariant and G is a predicate suitable for use as the loop guard. On the right-hand side the loop body S_1 includes a schema S_2 that effects a change on the variables in ρ_1 which must make progress towards termination.

Real-time requirements are introduced as a timing invariant t. Notice, however, that the left-hand side of the timed step is of the form $T \fatsemi \mathit{do}^{\mathrm{exit}}$, in order to account for the final overhead of exiting the loop when the guard becomes false. (Recall that '$\fatsemi$' is the Z schema composition operator, so the overall requirement is the combined behaviour of T and $\mathit{do}^{\mathrm{exit}}$.) The timing invariant is thus expected to be true at the start of the loop, when the guard is *about to be* evaluated, but may not still hold at the end of the iteration statement [4]! The loop body T_1 has a timing pre-condition which states that the invariant t must have been true before the overhead of evaluating the guard. The timing post-condition of T_1 asserts that t will be re-established after accounting for the overhead of branching back to the start of the loop.

$$\begin{array}{|l} \text{---}\ T_1\ \text{---} \\ S;\ \Delta Time \\ \hline t_1 \\ t_2' \end{array} \quad \sqsubseteq \quad \mathbf{delay}(E)\ ; \quad \begin{array}{|l} \text{---}\ T_2\ \text{---} \\ S;\ \Delta Time \\ \hline t_1\ \mathit{before\ del} \\ t_2' \end{array}$$

Figure 5: Introducing a leading idle delay.

2.3 Rules for Modifying Timing Behaviour

So far we have defined rules for traditional structured programming statements and their implicit effect on the passage of time. Real-time programming languages, however, often have special statements that explicitly refer to time [3].

2.3.1 Delay.

A statement of the form '**delay**(E)' is intended to delay execution by (at least) E time units [3]. Let its effect be defined by

$$del \mathrel{\widehat{=}} \left[\, \Xi PSpace;\ \Delta Time \;\middle|\; \mathsf{E} \dotplus \mathsf{del} \subseteq 0\,..\,E \;\wedge\; now' - now \in \{E\} \dotplus \mathsf{delerr} \,\right] .$$

Here del represents the unavoidable overhead required to effect a delay (either to suspend and reactivate the program or to devise a suitable busy-wait) and delerr represents the (hopefully small) possible "overrun". The pre-condition says the time required to evaluate the expression, i.e., E, plus the inevitable overhead of a delay, i.e., del, is less than or equal to the delay specified by E; we have not defined the effect when it takes longer to evaluate how long to delay than the requested delay itself! The overall execution time of the delay statement is E plus a possible overrun from delerr (in some situations this may be quite substantial [3, p.327]).

An idle delay can be introduced during refinement as shown in Fig. 5—the timing pre-condition is "advanced" by the length of the delay. (Of course the delay cannot be so great that it forces time to go backwards in T_2.)

2.3.2 Delay-until.

Whereas the delay statement introduces a relative delay, there is often a need for a statement that can delay until a given moment in *absolute* time [2]. For a particular delay-until statement of the form '**delay-until**(E)' define its effect to be

$$du \mathrel{\widehat{=}} \left[\, \Xi PSpace;\ \Delta Time \;\middle|\; \mathsf{E} \dotplus \mathsf{du} \subseteq 0\,..\,(E - now) \;\wedge\; now' \in \{E\} \dotplus \mathsf{duerr} \,\right] .$$

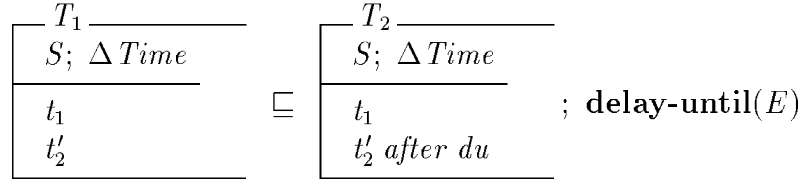


Figure 6: Introducing a trailing delay-until.

As before, we have a pre-condition requiring that the time required to implement the delay-until will not cause us to miss the deadline! The final time is required to be that specified, i.e., E, plus a small overrun from duerr, if any. (Note that du allows the range of finishing times to be narrower than the range of starting times!)

From this definition we can introduce a trailing delay-until statement using the rule in Fig. 6.

Of course, we can readily envisage symmetric rules for introducing trailing delays or leading delay-untils, or for replacing **skip**s with delays.

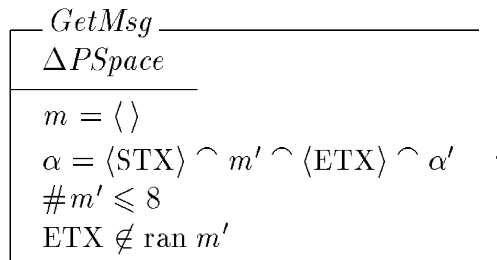
3 Example: A Message Receiver

We consider a small segment of embedded code intended to read a message. In Sect. 3.1 a refinement of the functional behaviour is presented. Section 3.2 then extends the refinement with real-time requirements.

3.1 A Functional Refinement

3.1.1 Functional Specification.

The aim is to read up to eight ASCII characters, delimited by STX and ETX, from a memory-mapped i/o location. Let m hold the message and α represent the incoming sequence of characters, i.e., $PSpace \mathrel{\widehat{=}} [\, m, \alpha : \mathrm{seq\,char} \,]$. The required behaviour is then specified as



$$\begin{array}{|l} \hline \textit{Input} \\ \Delta \mathit{InVars};\ \Xi \mathit{Others} \\ \hline \alpha = \langle c' \rangle \frown \alpha' \\ \hline \end{array} \quad \sqsubseteq \quad \mathbf{input}(\alpha, c)$$

Figure 7: Introducing an (untimed) input operation.

Initially m is empty and there is an implicit pre-condition that α does indeed begin with an appropriate message. The last predicate ensures that we find the first message of the correct form.

3.1.2 A Functional Rule for Input.

Refinement calculi do not usually contain rules for low-level constructs such as i/o, treating this as application-specific. We therefore define a rule for input from α, based on Morgan's precedent [13, ch.15]. Let *PSpace* be divided into those variables unchanged by an input operation, denoted *Others*, and those variables affected by the input, i.e., the incoming stream α and some receiving variable c, denoted $\mathit{InVars} \mathrel{\widehat{=}} [\alpha : \mathrm{seq}\,\mathrm{char};\ c : \mathrm{char}]$. A rule for introducing an input operation is then shown in Fig. 7.

3.1.3 Functional Refinement Steps.

Here we present, without proof, a refinement of specification *GetMsg* to code.

1. Introduce a variable c to hold the most recently read character, and a temporary logical constant $\alpha 0$ for referring to the initial value of α. Let $\mathit{PSpace}1 \mathrel{\widehat{=}} [\mathit{PSpace};\ c : \mathrm{char}]$ in

$$\mathit{GetMsg} \quad \sqsubseteq \quad \begin{array}{l} \mathbf{var}\ c : \mathrm{char} \bullet \\ \mathbf{con}\ \alpha 0 = \alpha \bullet \\ \begin{array}{|l} \hline \mathit{GetMsg}1 \\ \Delta \mathit{PSpace}1 \\ \mathit{GetMsg} \\ \hline \end{array} \end{array} .$$

2. Separate input of the first character from the rest:

$$GetMsg1 \sqsubseteq \begin{array}{|l}\hline \multicolumn{1}{l}{GetSTX} \\ \Delta PSpace1 \\ \hline m' = m \\ \alpha 0 = \langle c' \rangle \frown \alpha' \\ \hline \end{array} \; ; \; \begin{array}{|l}\hline \multicolumn{1}{l}{GetRest} \\ \Delta PSpace1 \\ \hline \alpha 0 = m \frown \langle c \rangle \frown \alpha \\ \alpha 0 = m' \frown \langle c' \rangle \frown \alpha' \\ \mathrm{ETX} \notin \mathrm{ran}\ m' \\ c' = \mathrm{ETX} \\ \hline \end{array} \; .$$

In *GetRest* we have expressed the pre and post-conditions involving α in the same form in anticipation of step **4** ($m = \langle\,\rangle$ and $c = \mathrm{STX}$ in the initial state).

3. *GetSTX* is now in a form where we can apply the rule from Fig. 7:

$$GetSTX \sqsubseteq \mathbf{input}(\alpha, c) \; .$$

4. *GetRest* can be refined into a while loop:

$$GetRest \sqsubseteq \mathbf{do}\ c \neq \mathrm{ETX} \rightarrow \begin{array}{|l}\hline \multicolumn{1}{l}{NextChar} \\ \Delta PSpace1 \\ \hline \alpha 0 = m \frown \langle c \rangle \frown \alpha \\ \alpha 0 = m' \frown \langle c' \rangle \frown \alpha' \\ c = \mathrm{STX} \Rightarrow m' = m \\ c \neq \mathrm{STX} \Rightarrow m' = m \frown \langle c \rangle \\ \hline \end{array} \; \mathbf{od} \; .$$

Since m is not meant to contain the STX and ETX delimiters, c is not added to m the first time around the loop, i.e., when $c = \mathrm{STX}$. (The loop body is not executed when $c = \mathrm{ETX}$ so that value is not added to m either.)

5. Separate *NextChar* into two parts to add the current character to the accumulating message, and to read the next character, i.e.,

$$NextChar \sqsubseteq \begin{array}{|l}\hline \multicolumn{1}{l}{AddToMsg} \\ \Delta PSpace1 \\ \hline \alpha' = \alpha \\ c = \mathrm{STX} \Rightarrow m' = m \\ c \neq \mathrm{STX} \Rightarrow m' = m \frown \langle c \rangle \\ \hline \end{array} \; ; \; \begin{array}{|l}\hline \multicolumn{1}{l}{InChar} \\ \Delta PSpace1 \\ \hline m' = m \\ \alpha = \langle c' \rangle \frown \alpha' \\ \hline \end{array} \; .$$

6. Refine the two behaviours of *AddToMsg* into alternatives, i.e.,

$$
AddToMsg \sqsubseteq \begin{array}{l}
\textbf{if} \\
\quad c \neq \text{STX} \rightarrow \begin{array}{|l}
\hline AddChar \\
\Delta PSpace1 \\
\hline \alpha' = \alpha \\
m' = m \frown \langle c \rangle \\
\hline
\end{array} \\
[] \\
\quad c = \text{STX} \rightarrow \textbf{skip} \\
\textbf{fi} \;\; .
\end{array}
$$

7. Before refining *AddChar* perform a data refinement to map character sequence m onto two concrete variables, a character array msg and an index idx representing its current length. Let $PSpace2 \mathrel{\widehat=} [\, c : \text{char};\ \alpha, msg : \operatorname{seq} \text{char};\ idx : \mathbb{N} \,]$ and the former pre-condition $m = \langle\,\rangle$ is now equivalent to $idx = 0$. We can now represent the act of adding a new element to the array via two operations, one to increment the index and the other to store the character, i.e.,

$$
AddChar \sqsubseteq \begin{array}{|l}
\hline Incr \\
\Delta PSpace2 \\
\hline \alpha' = \alpha \\
msg' = msg \\
c' = c \\
idx' = idx + 1 \\
\hline
\end{array} \; ; \; \begin{array}{|l}
\hline AddToArray \\
\Delta PSpace2 \\
\hline \alpha' = \alpha \\
msg' = msg \oplus \{ idx \mapsto c \} \\
idx' = idx \\
\hline
\end{array} \; .
$$

8. *Incr* is then easily refined to an assignment, i.e.,

$$Incr \sqsubseteq idx := idx + 1 \;\; .$$

9. Similarly for *AddToArray*, i.e.,

$$AddToArray \sqsubseteq msg[idx] := c \;\; .$$

10. The functional refinement is completed by rewriting *InChar* as an input using the rule in Fig. 7:

$$InChar \sqsubseteq \textbf{input}(\alpha, c) \;\; .$$

3.2 A Real-time Refinement

Here we add real time to the above refinement. Apart from the rules in Sect. 2 we admit other common refinement laws as required. In particular, we appeal implicitly to the ability to weaken pre-conditions, strengthen post-conditions, etc [13]. An advantage of introducing time as an ordinary specification variable is that no new rules are needed to "strengthen timing post-condition", etc.

$$
\begin{array}{|l}
\hline
\textit{tInput} \\
\Delta \textit{InVars};\ \Xi \textit{Others};\ \Delta \textit{Time} \\
\hline
\textit{now}' - \textit{now} \in \mathsf{io} \\
\exists\, n : \mathbb{N} \bullet (\textit{now} \in n * 50 + 25 \mathrel{..} n * 50 + 68 \land \\
\qquad c' = \alpha 0(n) \land \\
\qquad \alpha' = (n \mathrel{..} \# \alpha 0) \upharpoonright \alpha 0) \\
\hline
\end{array}
\quad \sqsubseteq \ \mathbf{input}(\alpha, c)
$$

Figure 8: Introducing a real-time input operation.

3.2.1 Real-time Specification.

Let

$$
\begin{array}{|l}
\hline
\textit{tGetMsg} \\
\textit{GetMsg};\ \Delta \textit{Time} \\
\hline
\textit{now} \in 25 \mathrel{..} 35 \\
\textit{now}' \leqslant \# m' * 50 + 135 \\
\hline
\end{array}
$$

be the real-time specification. It includes functional specification *GetMsg* but adds a timing pre-condition stating that the operation is expected to be started somewhere between absolute times 25 and 35. The maximum execution time is specified to be proportional to the length of the message received. Up to 50 time units are allowed for each character in the message, plus an additional 100 for the STX and ETX delimiters, plus 35 to allow for the worst-case starting time.

3.2.2 A Real-time Rule for Input.

For this embedded application further timing information is needed, however. If we are to successfully read from the i/o location we must know when each character is available. To model this we extend the refinement rule of Fig. 7 to account for the timing behaviour of the particular i/o location denoted by α.

Figure 8 defines a refinement for a *successful* input from α. It assumes that, starting at absolute time 25, characters appear at location α every 50 time units. However the value is only readable for the next 45 time units—there is a 5 time unit interval between the appearance of each character during which the value in the location is ill-defined. Thus the first character is available only from time 25 to 70, the second from 75 to 120, etc. Furthermore, because an input operation takes 2 time units (io in Appendix A) we must *begin* the first input between 25 and 68, the second between 75 and 118, etc.

The last two lines in *tInput* tell us that a successful input in the nth such time period will yield the nth character from the incoming stream and that the incoming stream is truncated by $n-1$ characters at that point. For conciseness, we have taken the liberty of expressing this using constant $\alpha 0$.

3.2.3 Real-time Refinement Steps.

We now augment the above refinement to account for these real-time constraints. The steps below are not presented in the same order as before—the rules let us start at any convenient point in the refinement and calculate other timing requirements in terms of the starting point.

4′. In iterative code it is, as always, crucial that we establish a satisfactory loop invariant, so we consider step **4** first. Firstly we extend the definition of *GetRest* to get it in a form suitable for the timed iteration rule in Fig. 4.

A satisfactory timing invariant is one in which we are at the start of the loop body every 50 time units in absolute time—this places us centrally in the available time for each character. To give some flexibility we include up to 2 time units leeway at each iteration:

$$tGetRest \mathrel{\hat=} \left[\begin{array}{l} tLoop \\ \hline GetRest;\ \Delta Time \\ \hline now \in (\#\alpha 0 - \#\alpha) * 50 \pm 2 \\ now' \in (\#\alpha 0 - \#\alpha') * 50 \pm 2 \end{array}\right] \mathbin{;} do^{\text{exit}} \quad .$$

Expression $\#\alpha 0 - \#\alpha$ represents the total number of characters read, including STX and ETX.

We must account for the overhead of exiting the loop. For the particular loop introduced in step **4** we can use the definition in Sect. 2.2 to determine that

$$do^{\text{exit}} \mathrel{\hat=} \left[\Xi PSpace1;\ \Delta Time \mid now' - now \in \{6, 7\}\right] \quad .$$

The overhead is equal to the time required to evaluate the loop guard $c \neq$ ETX and to conditionally branch out of the loop, i.e., using the figures from Appendix A,

$$(\mathsf{lvar} \dotplus \mathsf{lcon} \dotplus \mathsf{ccmp}) \dotplus \mathsf{cbry} = (\{0\} \dotplus \{1,2\} \dotplus \{3\}) \dotplus \{2\} = \{6,7\} \quad .$$

Although the invariant in *tLoop* allows a cumulative error in the timing behaviour, it can be seen that this invariant is satisfactory given the upper bound on the length of messages. In the worst case, where $\#m' = 8$ and the loop takes the maximum amount of time at iteration, as does do^{exit}, we have a final time of $(8+2) * 52 + 7 = 527$ whereas the specified maximum finishing time is $8 * 50 + 135 = 535$. (The invariant also places a bound on the *minimum* time.

This is not necessary for satisfying the constraint in *tGetMsg*, but is used for the timed input rule.)

We now apply the timed iteration rule (Fig. 4) to step **4**, to refine *tGetRest*:

$$tGetRest \sqsubseteq \textbf{do } c \neq \text{ETX} \rightarrow$$

$$\begin{array}{|l} \hline \textit{tNextChar} \\ \textit{NextChar};\ \Delta \textit{Time} \\ \hline (now \in (\#\alpha 0 - \#\alpha) * 50 \pm 2)\ \textit{before}\ do^{\text{loop}} \\ (now' \in (\#\alpha 0 - \#\alpha') * 50 \pm 2)\ \textit{after}\ od \\ \hline \end{array}$$

$$\textbf{od} \ .$$

We have already calculated do^{exit} above. The other timing calculations for the code in step **4** follow similarly, using the definitions in Sect. 2.2, to yield

$$do^{\text{loop}} \mathrel{\widehat=} \left[\Xi PSpace1;\ \Delta Time \;\middle|\; now' - now \in \{5, 6\}\right]$$

and

$$od \mathrel{\widehat=} \left[\Xi PSpace1;\ \Delta Time \;\middle|\; now' - now = 1\right] \ .$$

Substituting these predicates in the pre and post-conditions of *tNextChar* allows us to re-express it as

$$\begin{array}{|l} \hline \textit{tNextChar} \\ \textit{NextChar};\ \Delta \textit{Time} \\ \hline now \in (\#\alpha 0 - \#\alpha) * 48 + 5 \mathrel{..} (\#\alpha 0 - \#\alpha) * 52 + 6 \\ now' \in (\#\alpha 0 - \#\alpha') * 48 - 1 \mathrel{..} (\#\alpha 0 - \#\alpha') * 52 - 1 \\ \hline \end{array} \ .$$

Notice how the timing conditions have been tightened to account for the overheads of reaching the loop body (5 or 6 time units) and branching back to the start of the loop (1 time unit).

2′. With a definition of *tGetRest* available we can now apply the real-time sequence rule (Fig. 2) to step **2**, to calculate timing constraints for *GetSTX*:

$$tGetMsg1 \sqsubseteq \begin{array}{|l} \hline \textit{tGetSTX} \\ \textit{GetSTX};\ \Delta \textit{Time} \\ \hline now \in 25 \mathrel{..} 35 \\ (now' \in (\#\alpha 0 - \#\alpha') * 50 \pm 2)\ \textit{after}\ seq \\ \hline \end{array} \ ;\ tGetRest \ .$$

The timing pre-condition comes from *tGetMsg*. The timing post-condition can be simplified using the assumption that ';' incurs no run-time penalty, and the knowledge from *GetSTX* that $\#\alpha 0 - \#\alpha' = 1$, to give us a finishing time of 50 ± 2, i.e.,

$$\begin{array}{|l}
\textit{tGetSTX} \\ \hline
\textit{GetSTX};\ \Delta \textit{Time} \\ \hline
\textit{now} \in 25 \mathrel{..} 35 \\
\textit{now}' \in 48 \mathrel{..} 52 \\ \hline
\end{array} \quad .$$

So far we have been working top-down, *calculating* necessary timing requirements. To illustrate the bottom-up approach we will undertake the timed refinement of *AddToMsg* in reverse to *discover*, rather than specify, the timing behaviour of this operation's implementation.

8′. At the lowest level there are two concrete assignments corresponding to adding a value to m. From step **8** and the assignment rule (Fig. 1) we can state

$$\begin{array}{|l}
\textit{tIncr} \\ \hline
\textit{Incr};\ \Delta \textit{Time} \\ \hline
\textit{now}' - \textit{now} \in \{2, 3\} \\ \hline
\end{array} \quad \sqsubseteq \quad idx := idx + 1$$

because

$$\begin{aligned}
& \mathsf{gadd} \dotplus (\mathsf{lvar} \dotplus \mathsf{lcon} \dotplus \mathsf{iadd}) \dotplus \mathsf{stor} \\
= {} & \{0\} \dotplus (\{0\} \dotplus \{1,2\} \dotplus \{1\}) \dotplus \{0\} \\
= {} & \{2, 3\} \quad .
\end{aligned}$$

9′. Similarly,

$$\begin{array}{|l}
\textit{tAddToArray} \\ \hline
\textit{AddToArray};\ \Delta \textit{Time} \\ \hline
\textit{now}' - \textit{now} \in \{3, 4\} \\ \hline
\end{array} \quad \sqsubseteq \quad msg[idx] := c$$

because

$$\mathsf{gadd} \dotplus \mathsf{lvar} \dotplus (\mathsf{lvar} \dotplus \mathsf{sarr}) = \{0\} \dotplus \{0\} \dotplus (\{0\} \dotplus \{3, 4\}) = \{3, 4\} \quad .$$

7′. We then use the real-time sequencing rule (Fig. 2), together with our assumption that sequence does not consume time, to augment step **7** as

$$\begin{array}{|l} \multicolumn{1}{l}{__ \mathit{tAddChar} ________} \\ \mathit{AddChar};\ \Delta \mathit{Time} \\ \hline \mathit{now}' - \mathit{now} \in \{5,6,7\} \\ \hline \end{array} \sqsubseteq\ \mathit{tIncr}\ ;\ \mathit{tAddToArray}\ ,$$

where the execution time was found by adding $\{2,3\}$ and $\{3,4\}$ from *tIncr* and *tAddToArray*, respectively.

6′. We are now ready to augment step **6** in order to discover the provable timing behaviour for the alternative statement. Firstly, we calculate the timing overheads of the alternative construct using the definitions in Sect. 2.2. To reach the two alternatives we calculate that

$$\mathit{if}_1 \mathrel{\widehat{=}} \left[\Xi PSpace1;\ \Delta Time \mid now' - now \in \{5,6\}\right]$$

because

$$(\mathsf{lcon} \dotplus \mathsf{lvar} \dotplus \mathsf{ccmp}) \dotplus \mathsf{cbrn} = (\{1,2\} \dotplus \{0\} \dotplus \{3\}) \dotplus \{1\} = \{5,6\}$$

and

$$\mathit{if}_2 \mathrel{\widehat{=}} \left[\Xi PSpace1;\ \Delta Time \mid now' - now \in 11 \mathinner{..} 13\right]$$

because

$$\begin{aligned} & ((\mathsf{lcon} \dotplus \mathsf{lvar} \dotplus \mathsf{ccmp}) \dotplus \mathsf{cbry}) \dotplus (\mathsf{lcon} \dotplus \mathsf{lvar} \dotplus \mathsf{ccmp}) \dotplus \mathsf{cbrn} \\ = {} & ((\{1,2\} \dotplus \{0\} \dotplus \{3\}) \dotplus \{2\}) \dotplus (\{1,2\} \dotplus \{0\} \dotplus \{3\}) \dotplus \{1\} \\ = {} & 11 \mathinner{..} 13\ . \end{aligned}$$

After one of the two alternatives has executed the overheads are either

$$\mathit{fi}_1 \mathrel{\widehat{=}} \left[\Xi PSpace1;\ \Delta Time \mid now' - now = 1\right]$$

or

$$\mathit{fi}_2 \mathrel{\widehat{=}} \left[\Xi PSpace1;\ \Xi Time\right]\ .$$

If the first alternative is chosen we can compute the overall execution time of *tAddToMsg* as that of if_1, *tAddChar* and fi_1, i.e.,

$$\{5,6\} \dotplus \{5,6,7\} \dotplus \{1\} = 11 \mathinner{..} 14\ .$$

Similarly, for the second alternative (assuming that **skip** $\equiv [\Xi PSpace;\ \Xi Time]$) we add times from if_2, **skip** and fi_2 to get

$$11 \mathinner{..} 13 \dotplus \{0\} \dotplus \{0\} = 11 \mathinner{..} 13\ .$$

The possible execution times for *tAddToMsg* are thus

$$11\,..\,14 \cup 11\,..\,13 = 11\,..\,14$$

and we can write

$$\begin{array}{|l} \hline \textit{tAddToMsg} \\ \hline \textit{AddToMsg};\ \Delta \textit{Time} \\ \hline \textit{now}' - \textit{now} \in 11\,..\,14 \\ \hline \end{array} \quad \sqsubseteq \quad \begin{array}{l} \textbf{if} \\ \quad c \neq \mathrm{STX} \rightarrow \textit{tAddChar} \\ [] \\ \quad c = \mathrm{STX} \rightarrow \textbf{skip} \\ \textbf{fi}\ \ . \end{array}$$

We complete the refinement in the top-down style.

5′. Applying the sequence rule (Fig. 2) to step **5**, and using our computed timings for *tAddToMsg*, gives

$$\begin{array}{ll} & \textit{tNextChar} \\ \sqsubseteq & \textit{tAddToMsg}; \\ & \begin{array}{|l} \hline \textit{tInChar} \\ \hline \textit{InChar};\ \Delta \textit{Time} \\ \hline \textit{now} \in (\#\alpha 0 - \#\alpha) * 48 + 16\,..\,(\#\alpha 0 - \#\alpha) * 52 + 20 \\ \textit{now}' \in (\#\alpha 0 - \#\alpha') * 48 - 1\,..\,(\#\alpha 0 - \#\alpha') * 52 - 1 \\ \hline \end{array}\ . \end{array}$$

The timing post-condition of *tInChar* is the same as that of *tNextChar*. The pre-condition of *tInChar* is the same as the timing pre-condition of *tNextChar* with the duration of *tAddToMsg* added, i.e., $11\,..\,14$ time units from $(\#\alpha 0 - \#\alpha) * 48 + 5\,..\,(\#\alpha 0 - \#\alpha) * 52 + 6$.

11. At this point we want to refine *tInChar* to an input statement, as in step **10**. However when the real-time requirements are considered we see that the timing pre-condition of *tInChar* may not satisfy our rule for reading from the incoming stream (Fig. 8). For instance, when $\#\alpha 0 - \#\alpha = 1$, *tInChar* may attempt to read at time 70, "between" characters. Therefore we must modify the timing behaviour of *tInChar* to get it into a suitable form.

We will introduce an initial idle delay, using the rule in Fig. 5:

$$
\begin{array}{ll}
& tInChar \\
\sqsubseteq & \mathbf{delay}(29); \\
& \begin{array}{|l}
\hline
tInChar1 \\
InChar;\ \Delta Time \\
\hline
now \in (\#\alpha 0 - \#\alpha) * 48 + 45 \mathrel{..} (\#\alpha 0 - \#\alpha) * 52 + 49 \\
now' \in (\#\alpha 0 - \#\alpha') * 48 - 1 \mathrel{..} (\#\alpha 0 - \#\alpha') * 52 - 1 \\
\hline
\end{array} \ .
\end{array}
$$

The value of 29 time units was chosen to make the timing behaviour of *tInChar*1 consistent with that of *tInput* in Fig. 8. For instance, let $\#\alpha 0 - \#\alpha = 1$ and hence, by *InChar*, $\#\alpha 0 - \#\alpha' = 2$. The specified starting time of *tInChar* is then $64 \mathrel{..} 72$, but *tInput* tells us that to read the second character we must begin in the range $75..118$. Adding the delay makes the starting time of *tInChar*1 $93..101$ as needed. Similarly, when $\#\alpha 0 - \#\alpha = 8$ the starting times of *tInChar*1 are $429..465$ whereas those specified for a successful input are $425..468$, etc. This particular delay also makes the specified finishing times of *tInChar*1 compatible with a duration of 2 time units.

10′. We can now redo step **10** because the timing behaviour of *tInChar*1 is compatible with the stated timing behaviour of α given in Fig. 8:

$$tInChar1 \ \sqsubseteq \ \mathbf{input}(\alpha, c) \ .$$

12. We have now completed timed refinement of the loop. The remaining specification, *tGetSTX*, merely involves reading the first character from α. However we cannot directly refine *tGetSTX* to an input even though its pre-condition already satisfies the timing requirement in Fig. 8 because an input takes only 2 time units and the final time would not establish the necessary post-condition $now' \in 48 \mathrel{..} 52$. Introducing an idle delay as we did above will not help because, with such a wide range of possible starting times $(25 \mathrel{..} 35)$ there is no single delay period that will always guarantee that we meet the required finishing time. Therefore we must introduce an *absolute* delay, using the rule in Fig. 6, in order to guarantee that the post-condition is satisfied:

$$
tGetSTX \ \sqsubseteq \ \begin{array}{|l}
\hline
tGetSTX1 \\
GetSTX;\ \Delta Time \\
\hline
now \in 25 \mathrel{..} 35 \\
(now' \in 48 \mathrel{..} 52)\ \mathit{after}\ du \\
\hline
\end{array} \ ; \ \mathbf{delay\text{-}until}(48) \ .
$$

We have chosen to delay until absolute time 48 to ensure that the required post-condition is established, even given the possible over-run of this statement by up to 2 time units (duerr in Appendix A).

```
var c : char •
var msg : array of char •
var idx : int := 0 •

input(α, c);
delay-until(48);
do
  c ≠ ETX → if
                c ≠ STX → idx := idx + 1;
                          msg[idx] := c
              []
                c = STX → skip
              fi ;
              delay(29);
              input(α, c)
od .
```

Figure 9: Verified implementation of *tGetMsg*.

3′. From the definition of du in Sect. 2.3 we can determine that the minimum overhead associated with this particular **delay-until** statement is

$$\mathsf{lcon} \dotplus \mathsf{du} = \{1, 2\} \dotplus \{2, 3, 4\} = 3 \mathinner{..} 6 \ .$$

Thus, as long as $tGetSTX1$ finishes before time 42, the delay-until statement can achieve the timing post-condition in $tGetSTX$.

Strengthening the timing post-condition in $tGetSTX1$ allows it to satisfy the timing requirements for a successful input from α and we can re-perform step **3** using the rule from Fig. 8 to give

$$tGetSTX1 \ \sqsubseteq \ \mathbf{input}(\alpha, c) \ .$$

Figure 9 shows the final program, proven correct with respect to the functional requirements in *GetMsg* and the real-time requirements in *tGetMsg*.

4 Related Work

The rules presented above are derived from an earlier set of detailed proof obligations for real-time refinement [4]. (We have not given proofs of the rules here, but their derivation from the proof obligations is obvious.) These are more general than the rules presented above; they do not require the specifications to be in any

particular form, and allow the timing and functional requirements to be arbitrarily inter-related. Nevertheless the proof obligations are complex and discharging them is onerous. The rules presented here can be applied much more easily, as long as the specifications can be manipulated into a suitable form.

So far we have concentrated on sequential code segments. Nevertheless our long-term intention is to scale-up the techniques, particularly for concurrent, communicating systems. The "action system" model of Back and Sere [1] is felt to be a promising ally for our work, offering a natural way of extending the model. Our methods to date could be seen as suitable for particular atomic actions, with the real-time refinement of the communication and concurrency constructs as a distinct exercise. Indeed, a real-time version of action systems has already been proposed [11].

Hooman's methods [10] have much in common with ours. He uses a '*time*' variable, equivalent to our '*now*', and presents real-time verification rules for structured programming statements and idle delays. He also considers statements not covered here, notably a real-time clock read and rules for parallelism and multi-processing in an occam-like programming language. Proofs are based on Hoare triples extended with a "commitment" which specifies the interface between a statement and its parallel environment. However, the timing model for program statements is not as precise as ours. A single time value is associated with all assignments and all guarded commands—there is assumed to be "no overhead for other compound statements."

Hehner [9] discusses proof of real-time properties during formal program development but does not explore it in depth, noting the "disadvantage of requiring intimate knowledge of the implementation (hardware and software)."

Gordon [7] outlines part of the "safemos" project. High-level language programs are represented as state-transition automata. These are then used to verify the timing correctness of assembler-level code modelled as if running on an idealised stack machine. This work complements ours—our future plans include extensions to cover real-time refinement of programs to assembler code, in order to directly control the "uncertainties" in our timing primitives.

Mahony and Hayes [12] also present work that complements our own, but this time at a *higher* level of abstraction. Their "timed refinement" methods involve capturing real-time requirements as continuous functions. These are then refined to a top-level system design via decomposition into pipe-lined or parallel processes. Like Hooman, they use a model that explictly states "assumptions" about the environment.

Scholefield et al. [17] present rules for developing a detailed design from a specification in a timed CCS-like language. Timing constraints are expressed in an extended form of first order logic. However, the primitive timing assumptions are not as precise as ours. For instance, the rule for alternatives merely assumes

a "window" in which all guards are evaluated, and there is no equivalent of our fi_i overhead. More seriously, the rule for iteration assumes a fixed limit for the number of loops. Nevertheless, rules are given for a "timeout" operator and for concurrency, lacking in our work.

Automated tools have often been proposed to assist with real-time proofs of program code. There are two basic approaches, analysis of the program source [15], and instrumentation of the compiler [5, 6]. Our rules are compatible with these methods, and we feel that the latter approach is particularly promising, but the field has yet to mature.

Also relevant is the work of Gries [8] and Nielson & Nielson [14]. They present rules for re-arranging program code in order to improve "performance," although no proofs are made of *hard* real-time properties.

5 Conclusion

We have presented rules that allow real-time requirements to be added to the formal program refinement process. The rules are simple yet realistically account for the timing overheads associated with implementing typical programming constructs. Furthermore, we have shown by example how the demands of low-level embedded programming can be satisfied within this framework.

We contend that there is nothing fundamentally difficult about proving real-time properties. Apart from some trivial syntactic conveniences we have used standard specification and refinement methods only. The real challenge lies with the overwhelming level of detail that must be taken into account when reasoning about the real-time behaviour of code.

Acknowledgements.

I wish to thank the FME'94 referees for their comments and Peter Kearney for reviewing this paper. The Information Technology Division of the Australian Defence Science and Technology Organisation contributed towards presentation of this work.

References

[1] R-J. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

[2] V. Blázquez, L. Redondo, and J.L. Freniche. Experiences with delay until for avionics computers. *Ada Letters*, XII(1):65–72, Jan/Feb 1992.

[3] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages.* Addison-Wesley, 1990.

[4] C. Fidge. Proof obligations for real-time refinement. In *Proc. Sixth BCS FACS Refinement Workshop*, London, January 1994.

[5] E. Gabber, A. Averbuch, and A. Yehudai. Portable, parallelizing pascal compiler. *IEEE Software*, pages 71–81, March 1993.

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

[7] M.J.C. Gordon. State transition assertions: A case study. Draft chapter for Towards Verified Systems edited by J. Bowen, October 1993.

[8] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[9] E.C.R. Hehner. *A Practical Theory of Programming.* Springer-Verlag, 1993.

[10] J. Hooman. Compositional verification of distributed real-time systems. In *Proc. School on Formal Techniques in Real-Time and Fault-Tolerant Systems*, University of Nijmegen, The Netherlands, January 1992.

[11] R. Kurki-Suonio. Stepwise design of real-time systems. *IEEE Trans. Software Engineering*, 19(1):56–69, January 1993.

[12] 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.

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

[14] F. Nielson and H.R. Nielson. Forced transformation of occam programs. *Information and Software Technology*, 34(2):91–96, February 1992.

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

[16] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, 1991.

[17] D. Scholefield, H. Zedan, and H. Jifeng. Real-time refinement: Semantics and application. In A. Borzyszkowski and S. Sokolowski, editors, *Mathematical Foundations of Computer Science 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 693–702. Springer-Verlag, 1993.

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

Table 1: Primitive execution times.

Primitive	Times	Description
gadd	$\{0\}$	Get the address of a variable
lvar	$\{0\}$	Load a variable (of simple type)
stor	$\{0\}$	Store a variable (of simple type)
larr	$\{3,4\}$	Load an array element
sarr	$\{3,4\}$	Store an array element
lcon	$\{1,2\}$	Load a constant (of simple type)
ccmp	$\{3\}$	Character comparison operation
iadd	$\{1\}$	Integer addition or subtraction
br	$\{1\}$	Unconditional branch
cbry	$\{2\}$	Conditional branch (taken)
cbrn	$\{1\}$	Conditional branch (not taken)
del	$\{2,3,4\}$	Overhead of delay operation
delerr	$\{0\}$	Possible delay overrun
du	$\{2,3,4\}$	Overhead of delay-until operation
duerr	$\{0,1,2\}$	Possible delay-until overrun
io	$\{2\}$	Memory-mapped input/output

A Primitive Times

For the purposes of the example in Sect. 3 the target program is expected to run in an environment with the timing properties shown in Table 1.

These values incorporate considerable knowledge of the anticipated architecture, operating system and compiler. In particular they assume that addresses of all variables are known at compile-time (so gadd incurs no run-time overhead), the compiler does not keep track of which constants are held in registers (so constants are always loaded and lcon is never 0), the variables of "simple" types (e.g., integers and characters) in our target program are held in registers throughout execution of this particular code segment (so they do not need to be moved into or out of memory and lvar and stor involve no overhead during the execution of this code segment), elements of array variables are always loaded from, and stored to, memory (so larr and sarr are never 0), and no other activity is occurring on the processor (so delerr and duerr are insignificant).