

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-26

**Quartz: An Integrated Formal
Development Method for Real-Time
Software**

C. Fidge, P. Kearney and M. Utting

**September 1994
(Revised: April 1995)**

Phone: +61 7 365 1003

Fax: +61 7 365 1533

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

Quartz: An Integrated Formal Development Method for Real-Time Software

C. Fidge P. Kearney M. Utting
Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia

Abstract

Quartz is a formal software development method for concurrent real-time systems, currently being devised by the Software Verification Research Centre. It is a program refinement theory that supports systematic production of verified real-time code from a formal specification. Its model encompasses a broad range of development steps from abstract requirements specification, through high-level language programs, down to executable assembler code with verified timing behaviour. This article illustrates the method via a detailed case study.

Keywords and phrases: real time, formal methods, software engineering

Introduction

Quartz is a development method for verified real-time software, currently being devised by the Software Verification Research Centre. It aims to be

- a formal program development method based on the ‘refinement’ approach,
- applicable to construction of concurrent systems with ‘hard’ real-time constraints, and
- capable of operating at a broad range of abstraction levels.

This article introduces the major features of the emerging Quartz method via a detailed case study.

Background

Motivation. Hard real-time systems are those in which timing behaviour is as important as functional behaviour. Producing the right answer at the wrong time is as bad as producing the wrong answer! Furthermore, real-time systems are often safety critical. Their ‘real-world’ price of failure can be very high, so formal verification of their properties is often mandated.

Satisfying such timing requirements demands a great deal of rigour from system developers. Real-time software is thus expensive to manufacture to the degree of timing predictability needed. We consider formal methods to be a way of improving the accuracy and efficiency with which such software can be produced, by treating verification as an integral part of software development.

Existing formal techniques have treated aspects of the overall problem but the results remain disjointed. Many formal specification and programming language notations have been proposed for real-time systems yet few development approaches link the two. Verification and analysis methods have been proposed for real-time program code but the methods are made very complex by the need to know low-level details of the target environment. The Quartz project is integrating and extending this past work.

Enabling technologies. The Quartz method is made possible by a number of recent advances.

- Specification languages such as Z [17] have proven sufficiently flexible for describing

Modelling concurrency

Several similar models have recently emerged for formally modelling concurrent behaviour: *action systems* [2], the *Temporal Logic of Actions* [1] and UNITY [4]. Each uses an interleaving model to conservatively build on the experience gained with sequential refinement and verification methods.

In this article we use action system notation. An action system can be represented as a loop,

$$\begin{array}{l} \{Init\} \\ \mathbf{do} \\ \quad Action_i \\ \mathbf{od} \end{array}$$

with an initial state definition followed by one or more actions. Each action is of the form

$$Guard \rightarrow Body$$

with a guard that determines under what conditions the action may occur and a body which defines its effect.

Traditionally such a loop is viewed as a sequential system, but it can also model interleaved concurrent behaviour because no ordering is imposed on independent actions. An advantage of the approach is that the same system description can map to a number of different implementations, such as a straightforward sequential one, or a concurrent implementation with shared variable communication, or a parallel implementation with message-passing communication.

The pre/post semantics traditionally used for sequential refinement are inadequate for systems that interact repeatedly with their environment. Refinements in action systems that change guards or add/remove actions are therefore defined with respect to all *traces* (sequences of global state changes) that the system may generate [2]. Refinement of individual action bodies may be done using conventional refinement laws [11].

- | | |
|--|--|
| <p>behaviour at many levels of abstraction. Agreed-upon notations for expressing concurrency and timing are still lacking, however.</p> <ul style="list-style-type: none">• ‘Refinement’ methods offer formal rules for deriving a high-level language (HLL) program from its specification; program development and verification thus proceed in lockstep [11]. To date these techniques have been limited to sequential, untimed programs, but new rules are appearing for concurrent systems (see box).• Techniques have been suggested for ‘predicting’ the real-time behaviour of HLL programs from their source code [5]. However, the accuracy of such methods is limited by | <p>the known timing properties of the target compiler, architecture and operating system.</p> <ul style="list-style-type: none">• Recent proposals have extended HLL program refinement steps with real-time proof obligations [6, 14]. Again, however, these obligations cannot be fully discharged without considerable knowledge of the target execution environment.• Recent work treats the compilation process itself as program refinement [12]. This work has to date been limited to the sequential, untimed case.• It has been shown that the real-time behaviour of modern computer architectures |
|--|--|

can be formally described and automated proofs of assembler level programs can be undertaken using such a model [7].

Features from all of these results are being integrated into the Quartz method.

The Quartz method

Figure 1 gives a broad overview of the Quartz method (the example illustrated is described in depth in the next section). Quartz encompasses the range of real-time software development activities from requirements specification, through high-level language code development, down to generation of executable assembler code.

1. Development begins with a formal description of both functional and timing requirements. A real-time specification says not only what to do, but when to do it. In this case the remainder of integer division by 60 must be computed with a worst-case execution time of 130 time units.
2. The programmer uses formal rules to derive program code, and its timing constraints, from the specification. Reasonableness checks can be applied to the timing constraints as they are produced in order to determine, as early as possible, whether development is proceeding in the right direction. In this case an iterative program is developed and, by observing that the maximum possible number of iterations n is 60, the programmer has recognised that there are at most 2 time units available for each iteration; the timing is tight but not impossible!
3. The resulting HLL code is accompanied by detailed timing constraints discovered during code development. Such constraints must be retained until formally discharged. Typically this requires detailed knowledge of the execution environment available only at compile time. In this case, for instance, we know that the time taken to evaluate the loop guard g , execute the subtraction statement in the loop s , and return to the beginning of the loop r , cannot exceed 2 time units. It is impossible to say whether

this constraint can be satisfied or not without knowing what object code will be generated for the loop.

4. Program compilation takes place using code generation rules proven to respect stated timing constraints. Timing obligations are fully discharged against a precise model of the target machine and the programmer is alerted to unsatisfiable timing constraints. In this case the code optimisation strategy has produced only two statements for each iteration, `subtract` and `brgte`, by placing the loop test after the loop body. Each of these instructions takes 1 time unit, so the timing constraint discussed above can be formally discharged.
5. The final assembler code has predictable real-time behaviour proven to conform with that of the original requirements specification. In the example illustrated, the worst case calculated timing for this code is 127 time units, thus satisfying the original requirements specification. (Quartz is not limited to just worst-case timing, however.)

The Quartz method is ‘integrated’ in the sense that the same formal modelling technique is used to define the semantics of the system at all levels of abstraction. Furthermore, the formal rules for HLL program development, and for defining the compilation process, are merely special cases of the general real-time refinement techniques (see box).

The specific target languages currently being used for Quartz research are

- the specification language Z, extended with concurrency and timing constructs, for requirements specification,
- a predictable subset of Ada 95, augmented with timing constraints, as the target high-level language, and
- assembler code for the MIPS R3000 RISC processor as the final executable code.

Z is also used as the formal modelling language for defining the Quartz development rules.

The Quartz formalism defines semantics-preserving translation rules that ensure timing

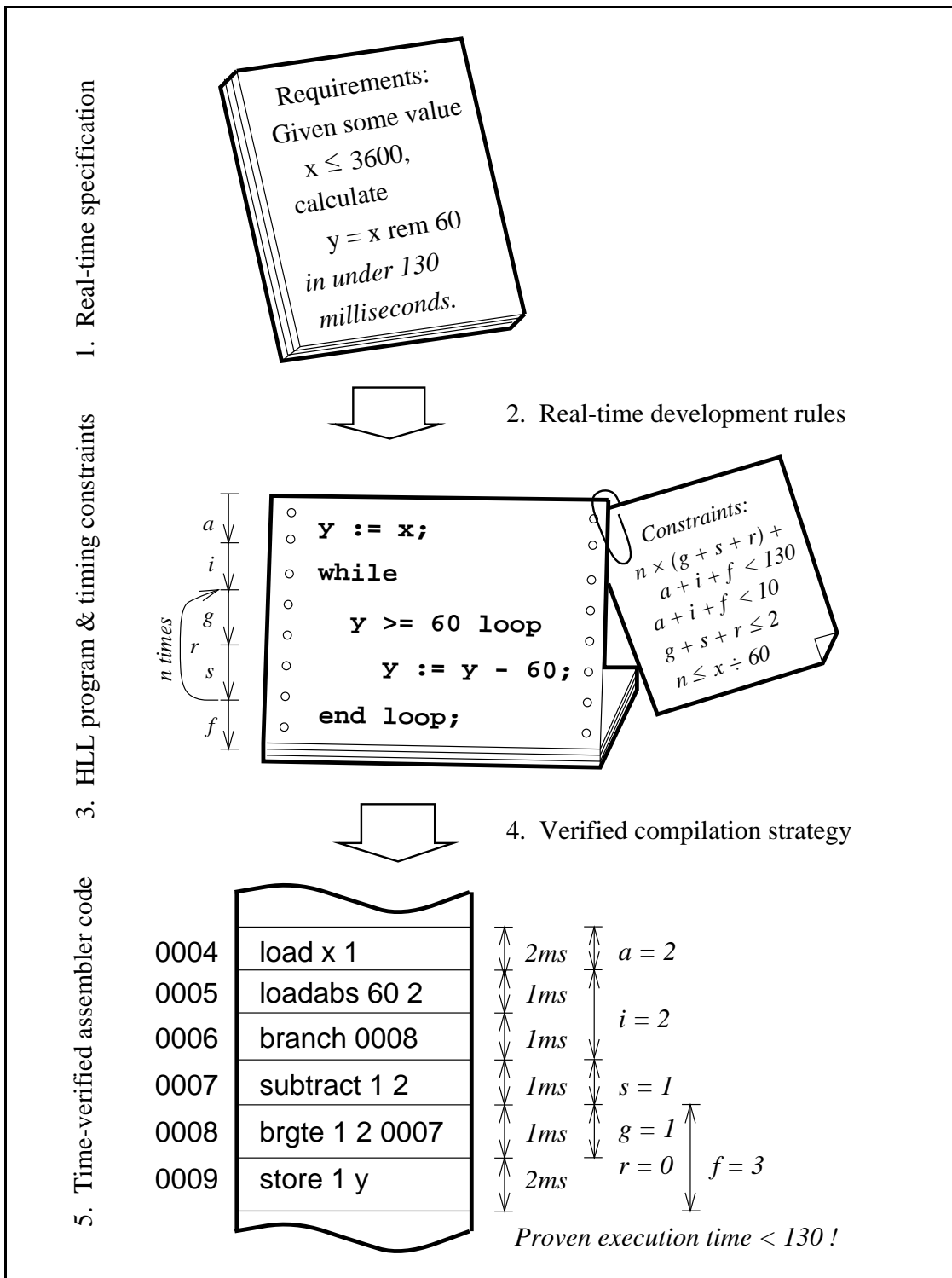


Figure 1: Overview of the Quartz method

behaviour is respected. How these rules are applied is largely up to the developer, however. Timing obligations may be discharged in either a top-down or bottom-up manner [6]. That is, the programmer may choose to *impose* timing requirements on system components as the design develops, or to use low-level timing primitives to *discover* the timing behaviour of already-completed components.

Case study

This section illustrates how the Quartz methodology would be applied. It shows what the major steps would be during development of a small embedded program. Here notations currently found in the literature are used for the purposes of illustration. A distinct syntax for the Quartz method may be defined at a later date.

Requirements. The goal is to develop an embedded program that is proven to process data from sporadic inputs quickly enough so that no inputs are lost.

Values in the range 0 to 3600 appear at irregular intervals, with a minimum separation of 135 milliseconds, in variable ‘*in*’. (These values represent the number of seconds that have elapsed in the current ‘wall time’ hour.)

The system must divide each number by 60 and place the remainder in variable ‘*out*’. (This produces a value in the range 0 to 60. Variable *out* will be used to display the number of seconds that have elapsed in the current minute.) In order to keep up with the worst-case input frequency each output must be available in under 135 milliseconds.

This is shown diagrammatically in Figure 2. Following the appearance of each input value the next input will not appear for *at least* 135 milliseconds, and there must be an appropriate output in strictly *less than* this time.

The program is expected to execute on a processor where load and store instructions each take 2 milliseconds and other instructions each take 1 millisecond. (Elsewhere we have shown how to handle more realistic timing primitives, including those with ‘uncertain’ timing [6].) Furthermore, to avoid too easy a solution, the

target machine has no multiplication or division capabilities.

Formal specification. Let us declare the incoming and outgoing values to be numbers in the ranges

$$secsPastHour == 0 .. 3600$$

and

$$secsPastMin == 0 .. 60,$$

respectively. Also, a linear, discrete notion of time is sufficient for this example so we let natural numbers, i.e.,

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

represent milliseconds of absolute time.

One way to formally represent the requirement is shown in Figure 3. It is a Z [17] specification representing the system via the histories, or traces, of values appearing in the *in* and *out* variables, and the times at which the values appear, *inT* and *outT* (see box).

Schema *Environ* defines the incoming data stream. It describes a sequence *inHist* of values *in* and their arrival times *inT*. The predicate formalises our expectation that all distinct inputs are separated by at least 135 time units.

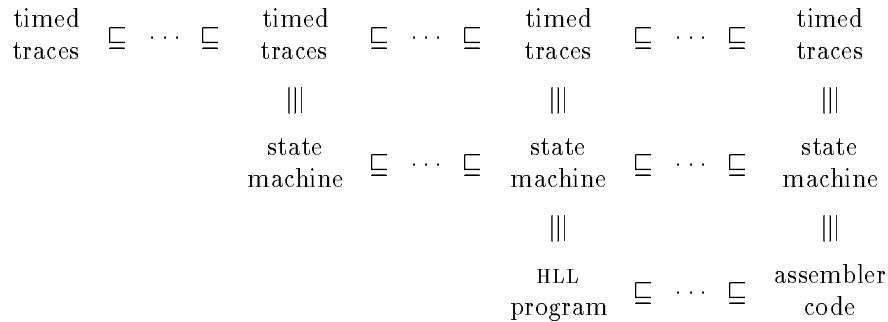
In such an environment our goal is to then develop a program which will generate values as defined by schema *System*. It introduces a sequence *outHist* of outgoing values *out*, where each value is made available for consumption by time *outT*. The predicate has two parts. The first says that the size of the history of incoming values can exceed that of the outgoing values by at most one, at any time. The second part defines the desired link between the input and output values. It says that each output value must appear within 135 time units of the corresponding input and that each output value is the remainder of dividing the input value by 60.

This is a highly abstract specification. It says nothing about how long it takes to produce the incoming values or to consume the outgoing ones (or even if they are consumed!). Nor does it say anything about how to process the incoming data—it merely notes what values are to be produced and deadlines by which they must be available.

Real-time refinement rules

Program development in Quartz uses semantics-preserving ‘refinement’ rules. In the past such rules, whether for converting specifications to HLL code [11], or HLL code to assembler [12], have preserved functional behaviour only. Quartz extends them so that specified real-time behaviour is maintained as well [6].

The Quartz refinement relation \sqsubseteq is founded in a notion of ‘timed’ traces.



Development proceeds from abstract trace-level descriptions at the top left, to executable code at the bottom right, although the particular path taken is up to the programmer. A mapping \equiv is defined between timed trace descriptions and equivalent timed, concurrent state machines, represented in this article as action systems. At these levels the definitions and rules are generic; they apply to a wide range of target languages. The specific target languages of interest are introduced when a system description at the level of common structured programming constructs, such as assignment, sequence, iteration and choice, is mapped to a high-level language program code, or one built from simple operations on ‘memory’ and ‘register’ arrays maps to an assembler program.

These semantics deliberately blur the distinction between ‘specifications’, ‘high-level language programs’ and ‘assembler code’. The generic model and proof methods are the same at all levels. Specific development rules must be derived from the generic theory to allow particular target HLL and assembler constructs to be efficiently introduced into a design, however.

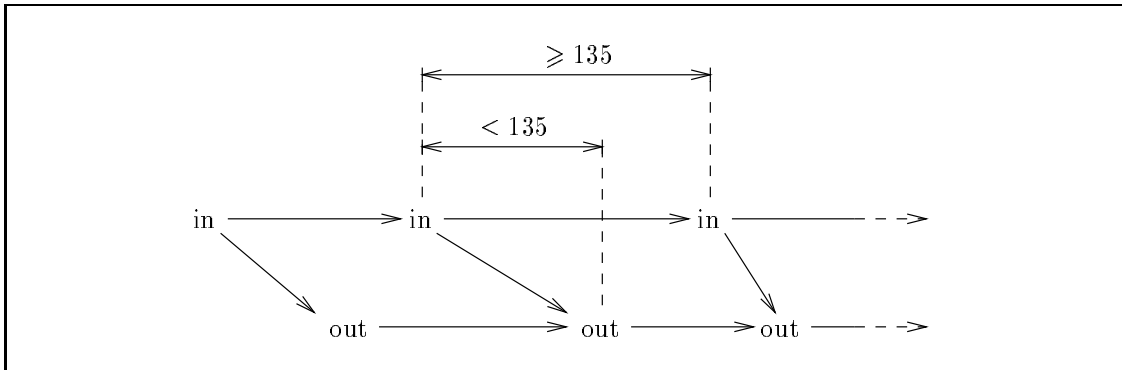


Figure 2: Desired timing behaviour.

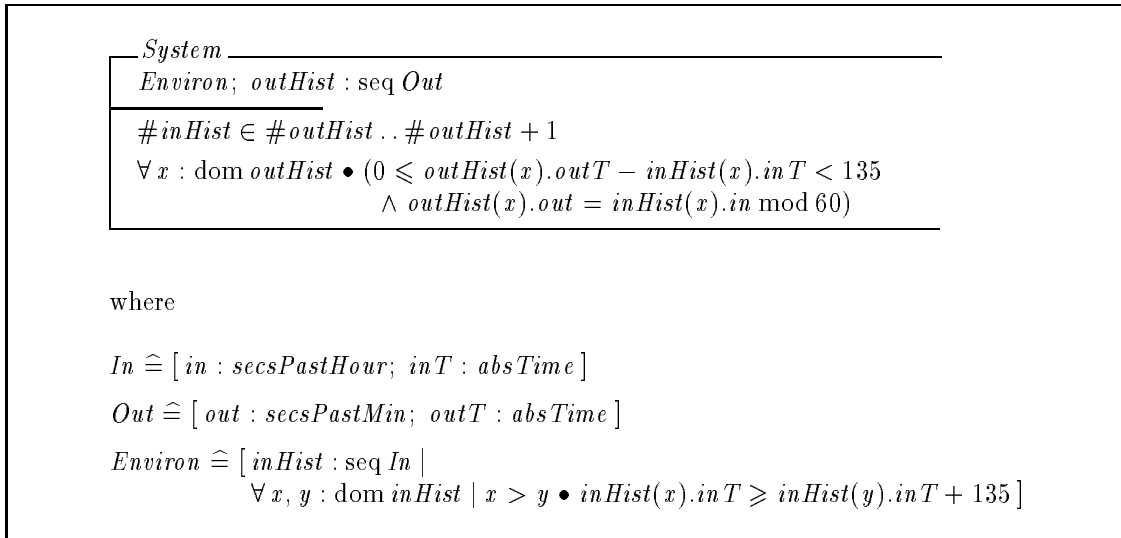


Figure 3: History-based description.

Overall design. From such a history-oriented description we can devise an overall system design that achieves this behaviour. We illustrate this with action system notation.

Figure 4 is a state-machine representation of the above specification. It describes a system of atomic actions that obeys the description in Figure 3. Initially the two time variables are asserted to be zero. Future behaviour is then defined via two distinct actions.

In the first, which can occur only when there are no unprocessed inputs, some new input value is produced at least 135 time units since its predecessor. (In Z schemata, a ‘primed’ variable name, e.g., ‘*out'*’, denotes a final value, and an unprimed one an initial value. The ‘ ΔIn ’ notation tells us that *in* and *inT* can change, within the constraints defined by their declared types in *In*. Notation ‘ $\exists Out$ ’ says that none of the variables in schema *Out* change.)

The second action, which may occur only when the most recent input has not yet resulted in a corresponding output, calculates the new value, and does so within 135 time units from the moment when the most recent input was produced.

The system description in Figure 4 says nothing about how synchronisation between the input and output actions is to be achieved. The

guards merely denote conditions that must hold for the actions to occur. The behaviour could be achieved by a sequential interleaving of input and output actions, or by two appropriately synchronised concurrent actions. Concurrency and communication constructs in the final program are introduced as part of this mapping.

At this level of abstraction program-like constructs such as ‘**do . . od**’ are logical only. They belong to the Quartz model, not to the target programming language, and thus incur no run-time overheads. The particular notation used here mixes Z schemata with guarded command language constructs [17].

Furthermore, the *Input* and *Output* abstractions do not necessarily define an execution-time ‘duration’ for actions that implement them. The time variables mark deadlines by which significant ‘events’ (externally-observable state changes) take place, *not* the passage of time. Thus the difference between *inT* and *inT'* in the first action does not suggest that an implementation of this action *must* consume more than 135 milliseconds of CPU time, merely that an implementation must perform state changes that are observable at times *inT* and *inT'*.

Program code development. At this point each action can be ‘refined’ to sequential HLL

Modelling time

In the Quartz model ‘time’ is represented through auxiliary specification variables [8]. This simplifies the formal development rules considerably because new verification techniques are not needed for reasoning about time [1]. Although founded in an atomic-action specification formalism, the Quartz model allows time-consuming, overlapping, and even pre-emptive, actions to be modelled. Actions occur ‘instantaneously’ in the underlying model but to the specifier they appear to consume time because they mark ‘resources’, such as shared variables or a shared processor, as inaccessible by any other action for their specified duration.

Care is taken in the underlying model to ensure that time variables adequately represent physical reality. For instance, constraints ensure that only one action can access a shared resource at a time, invariants guarantee that time cannot go backwards, and concrete code cannot directly access logical time variables.

The specifier can mix timing and functional variables in system descriptions. Thus situations where the real-time and functional behaviour are inter-related can be described, e.g., when the execution time is dependent on the input data, or when the program changes its behaviour in response to a deadline expiring.

code. Figure 5 shows a particular specification of the *Output* behaviour. It anticipates a ‘shared variable’ mapping of the system in Figure 4 to a physical configuration in which the *Output* action is capable of executing without interruption, no more than 5 milliseconds from the availability of each input. This slight delay accounts for the synchronisation overhead between the *Input* and *Output* actions. Under these circumstances the rest of the time remaining before the *Output* deadline can be treated as the acceptable worst-case execution time for the implementation. This is represented in *Remainder* by expressing the duration of the action, i.e., its finishing time minus its starting time, as being less than 130 milliseconds.

Time variable *now* is here used to denote the passage of available processor time. It is linked to the two local time markers *inT* and *outT*, always being the larger of the two when an observable event occurs. Figure 5 also constrains $\Delta Time$ so that *now* cannot go backwards.

Development of a HLL program from this specification proceeds by application of ‘real-time refinement’ rules [6]. Knowing that the target machine has no multiplication capability, the programmer decides to calculate the remainder via repeated subtraction. Anticipating the iterative code, the first step undertaken is

to split the problem into two sequential components as shown in Figure 6. The first component is intended to establish the loop invariant, while the second represents the loop itself. (The final, primed, state of the first component becomes the initial, unprimed, state of the second.)

Any such development step carries with it a real-time proof obligation, in this case requiring that the sum of the execution times for the two components is compatible with the overall execution time specified in *Remainder*. Anxious to check timing feasibility, and knowing that loads and stores each take two time units on the target architecture, the programmer has assumed that the ultimate implementation of *Init* will involve loading the value of *in* into a register and then storing this value into location *out*. Therefore a decision was made to specify the execution time of *Init* as exactly 4 time units, and hence the *Loop* action must execute in less than 126 time units if the combined behaviours are to satisfy *Remainder*.

In fact, this was an unwise move by the programmer. It prematurely introduced detailed machine-level assumptions and, more seriously, overly constrains future development of the actions. An implementation of *Init* is now obliged to take exactly 4 time units, preventing potential optimisations that might have performed the action in less time. For example, if it could

be proven that the value of *out* was *already* equal to *in* at this point in the program then *Init* need take no time at all! Although the timing requirement in Figure 6 is not ‘wrong’, choosing $now' \leq now + 4$ would have given the compiler greater freedom. This illustrates the inevitable tension between the desire to perform timing feasibility checks early, in order to avoid spending time on ‘dead-end’ developments, and the need to leave timing constraints as loose as possible, in order to give the greatest amount of freedom for code rearrangement at either the HLL or assembler levels.

The initialisation action, which merely sets *out* to equal *in*, can be easily mapped to the HLL program shown in Figure 7 (see box). The program code itself is accompanied by the programmer’s specified timing behaviour, in the form of an undischarged proof obligation.

Development of the *Loop* action is more challenging. Proving real-time properties of iterative code typically involves identifying a ‘timed’ loop invariant [6], as shown in Figure 8. Here the programmer has introduced two temporary logical constants *out0* and *now0* with values fixed to the initial values of the corresponding variables when the loop began [11]. The loop itself is re-expressed as an action that maintains invariant *invL*, and terminates with the final value of *out* strictly less than 60.

The invariant defines the functional requirement by stating that *out* is always less than the value of *out0* by some whole multiple of 60. In the final state, when *out* is less than 60 and this invariant is true, then *out* must equal the remainder of dividing *out0* by 60. In order to make progress towards termination, while maintaining the invariant, *out* must decrease by at least 60 with each iteration and, knowing that the maximum value of *in* is 3600, the programmer can tell that in the worst case there will be 60 iterations.

The second conjunct of invariant *invL* defines the maximum time that can have elapsed, since *now0*, as a function of *n* and two time constants. These constants denote the timing overheads associated with entering and exiting a loop statement. Unlike the iteration construct used in Figure 4, a programming language loop will incur real time penalties in its implementation.

The two times mentioned here are the initial overhead of reaching the loop guard for the first time *i*, and the final overhead of evaluating the (false) guard and exiting the loop *f*. In this instance the programmer has wisely chosen not to guess what values these overheads will eventually have, but has left them as symbolic constants.

The real-time and functional behaviours are inextricably linked. The invariant states that the timing behaviour is proportional to ‘*n*’ times the overhead of each loop, where *n* is determined by the program variables.

Given the maximum execution time for *Loop* of 126 milliseconds, and knowing that there may be up to 60 iterations, the programmer has stated that the overall timing requirement can be satisfied if the loop entry and exit overheads do not exceed 6 time units and each iteration takes at most 2 time units. By defining this timing invariant the programmer has partitioned the timing requirement even further, choosing a particular method of satisfying *Loop*, and can undertake some ‘reasonableness’ checks on the specified timing behaviour, even without knowing the actual values associated with the constants. Since no *negative* values are required to meet the goal, the programmer is given some confidence that the timing obligation is satisfiable. However, there is not yet a guarantee that the particular combination of target languages, compiler and machine *can* meet this requirement.

A specification in such a form can be readily translated into equivalent HLL code as shown in Figure 9 [6]. Two new invariants are easily calculated, using the overall loop invariant and allowing for the time *g* required to evaluate the (true) guard and reach the loop body, and the time *r* to return from the end of the loop body to re-evaluate the guard. Invariant *invB* is always true at the moment the loop body is about to begin executing, because the guard has been evaluated one more time than the body itself has executed. Invariant *invE* is true at the end of the body, when the guard and body have been executed the same number of times, but the overhead of returning to the guard has yet to be encountered. (The timing invariant is still expressed relative to *now0*, when *Loop2* began,

$$out := in \wedge [\Delta Time \mid now' = now + 4];$$

Figure 7: High-level language description of *Init*.

Modelling code

For each target language construct a mapping must be formally defined between the construct and its corresponding meaning in the Quartz model. High-level language and assembler statements are thus defined through their effect on functional and time-related specification variables.

Unfortunately, contemporary programming languages and architectures were not designed with predictable timing in mind. Quartz therefore acknowledges that the timing behaviour of some primitive actions must ultimately contain a degree of uncertainty. This is done by assigning *sets* of possible execution times to primitive actions. This allows both minimum and maximum execution times to be modelled.

The run-time overheads associated with each language construct must be defined with sufficient accuracy to minimise the spread of timing estimates. The extent to which this is practical involves an application-specific trade-off between the degree of precision to which we model the effect of the target architecture on language constructs [15], and the acceptable spread of timing estimates in proven results. Typically, any imprecision in timing estimates for primitives rapidly accumulates, particularly in iterative code, making proof of timing properties progressively more difficult.

Features of the target architecture strongly influence the assembler language model, via their impact on timing predictability. In particular, the RISC architectures currently popular for implementing real-time systems have features such as instruction pipelines, memory caches and interrupts that must all be formally modelled [15].

```

con out0 = out •
con now0 = now •
 $\exists i, f : absTime \mid i + f < 6 \bullet$ 

```

$Loop2$
$invL; invL'; \exists In$
$out' < 60$

where

$$invL \hat{=} [In; Out; Time \mid \exists n : \mathbb{N} \bullet (out + n * 60 = out0 \wedge now \leq now0 + i + n * 2 + f)]$$

Figure 8: Development of *Loop* using a real-time invariant.

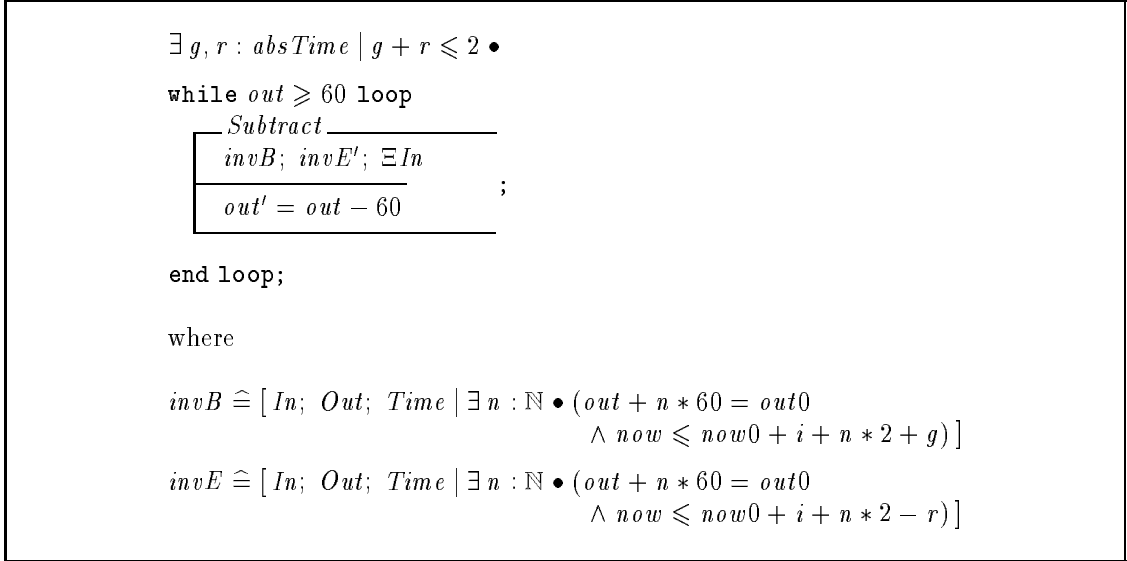


Figure 9: Development of *Loop2* by introducing a while loop.

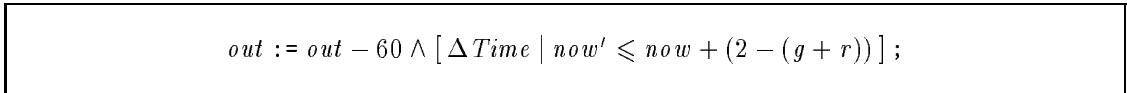


Figure 10: High-level language description of *Subtract*.

so the initial overhead i must be included.)

The loop body can then be mapped to the target programming language as shown in Figure 10. The specific timing requirement was devised from the ‘difference’ between the two invariants in Figure 9. If the duration of the loop body is as specified then these timing invariants are guaranteed to be respected. Here the programmer has stated that each iteration takes the same amount of time, a decision not necessarily required by the invariants. Obviously this timing requirement on the loop body is very demanding!

Figure 11 collects the above steps together and shows the final HLL program developed from the specification in Figure 5, in an Ada-like syntax. Notice that the executable statements on the left are accompanied by, as yet, undischarged timing requirements discovered during the course of the development on the right. The time variables, e.g., now , and the temporary constants, e.g., $out0$, are logical and serve to define proof obligations only. No object code

will be generated for them. The two assignment statements have defined durations, while the loop invariants serve to define the allowable overheads of implementing the **while** construct.

Assembler code development. Development of executable assembler code from the HLL program in Figure 11 also proceeds by systematic application of Quartz development rules. This section illustrates the major steps involved in such a compilation process.

Consider the HLL statement in Figure 7. It says that the memory location corresponding to program variable out is to be updated with the value of variable in , and that this is to be done in four time units. A ‘data refinement’ [17] can thus be performed to map this statement to its intended effect on physical memory. Let memory be defined as a mapping from addresses to values, as follows:

$$\text{Memory} \hat{=} [mem : address \rightarrow value] .$$

The HLL statement can then be expressed as

<code>out := in</code>	$\wedge [\Delta Time \mid now' = now + 4];$
	con $out0 = out; now0 = now \bullet$
	$\exists i, f, g, r : absTime \mid i + f < 6 \wedge g + r \leq 2 \bullet$
while <code>out \geq 60</code> loop	
<code>out := out - 60</code>	$\wedge [\Delta Time; invB; invE' \mid now' \leq now + (2 - (g + r))];$
end loop	$\wedge [\Delta Time; invL; invL'];$

Figure 11: High-level language program (left) and calculated timing requirements (right).

$mem(oloc) := mem(iloc) \wedge [\Delta Time \mid now' = now + 4]$

Figure 12: Development of *Init* in terms of its effect on memory.

shown in Figure 12, where *oloc* and *iloc* are constants representing the memory addresses allocated for variables *out* and *in* respectively. It states that the values in all memory locations are unchanged, except that the value at address *oloc* must be updated to be the same as that at *iloc*.

Further data refinements introduce other machine-specific features. An array of CPU registers

$$Regs \hat{=} [reg : regnum \rightarrow value]$$

is added next. Figure 13 shows how the update to memory can be achieved via two actions in sequence. The first places the value of memory location *iloc* into register 1, and the second then places the value in register 1 into memory location *oloc*. The timing requirement is divided evenly between the two actions.

The assembler-level development of *Init* is completed by introducing the program counter,

$$PC \hat{=} [pc : address],$$

which, if incremented by each action, allows the description in Figure 13 to be mapped into executable assembler code as shown in Figure 14.

Formally these two assembler instructions are defined in the Quartz model as shown in Figure 15. Obviously the degree of complexity has increased dramatically as the level of abstraction has lowered, even for the simple target machine assumed for this case study. Accurately

modelling a real machine, with features such as pipelining, caching, interrupts, etc. is possible [15], but managing the resulting detail poses a formidable challenge. Appropriate tool support will be important to achieve adequate productivity in practice.

Development of the while loop follows similarly, using the same data refinements. Figure 16 shows an intermediate description of the loop, in the Quartz modelling language, following introduction of the memory and register abstractions. This complex description is midway between the HLL program and assembler code. It has machine-level concepts, such as registers, not found in a HLL program. It is not yet mappable to executable assembler code, however, because the updates to *now* are still not uniquely determined and the program counter has yet to be introduced.

In Figure 11 the variable *out* was accessed during every loop iteration. Rather than repeatedly moving its value to and from memory, an optimisation has been performed so that the value is kept in register 1 throughout execution of the loop body. This was done when introducing the register-level abstraction via ‘encoding’ and ‘decoding’ actions added before and after the loop respectively. Before the loop the value in memory location *oloc* is placed in register 1 and after the loop the value in register 1 is stored in location *oloc*. Within the loop itself the data

```

reg(1) := mem(iloc) ∧ [ ΔTime | now' = now + 2 ];
mem(oloc) := reg(1) ∧ [ ΔTime | now' = now + 2 ]

```

Figure 13: Development of *Init* using registers.

```

load iloc 1
store 1 oloc

```

Figure 14: Assembler code developed from *Init*.

refinement thus replaces references to ‘*out*’ with ‘*reg*(1)’. Similarly, the constant ‘60’ has been recognised as a ‘loop-constant expression’ [16] and is therefore ‘loaded’ into register 2 in the first action only.

The **while** construct has been replaced by its underlying **do..od** model with explicit operations to model the timing overheads of the compiled code. The action system model thus serves to model both top-level concurrency and low-level assembler execution! The stated durations ensure that the programmer’s invariants from Figure 11 are maintained. The operation immediately after the loop in Figure 16 denotes the overhead of evaluating the false guard for the last time—this overhead must be taken into account when determining the loop finalisation time *f*.

Development of the loop is completed by introducing the program counter abstraction and using it to represent the effect of iteration on control flow. The description in Figure 17 does this, for a particular compilation strategy in which the guard test is placed immediately after the loop body.

The first three of the four actions manipulate the program counter in order to explicitly achieve the desired loop. The first initially sets the program counter to be *test*, the address at which the code for testing the guard is to be placed. The second evaluates the loop guard. If the guard is true it sets the program counter to *body*, where the code for the loop body is to be found. If the guard is false the program counter is set to the loop exit address *exit*. The third

action is the loop body itself which ends by setting the program counter so that the guard will be evaluated again.

In this compilation strategy there is no overhead associated with returning from the end of the loop body to the guard test, so

$$r = 0$$

and references to it could be omitted from Figure 17.

Development of the third action in Figure 17, i.e., the loop body, can be finished straightforwardly. Its specified timing behaviour can be strengthened by adding $now' = now + 1$. Letting $test = body + 1$, it then maps to a subtract instruction in the target assembler language:

```

subtract 1 2 .

```

This step has further constrained the overheads associated with each iteration so that

$$g + r \leq 1 .$$

Therefore the second action in Figure 17 has to execute in one time unit or less. Again the timing goal is very demanding, but not impossible!

These inter-relationships between time constants highlight a formidable challenge in the development of real-time programs. Parts of the system whose functional behaviour is independent may still affect one another’s timing behaviour, thus significantly complicating the program development process. Here the loop test and body cannot be refined in isolation because their individual timing behaviours both

$$\begin{aligned}
& [\exists Mem; \Delta Time; \Delta Regs; \Delta PC \mid now' - now = 2 \\
& \quad \wedge reg' = reg \oplus \{1 \mapsto mem(oloc)\} \\
& \quad \wedge pc' = pc + 1]; \\
& [\Delta Mem; \Delta Time; \exists Regs; \Delta PC \mid now' - now = 2 \\
& \quad \wedge mem' = mem \oplus \{oloc \mapsto reg(1)\} \\
& \quad \wedge pc' = pc + 1]
\end{aligned}$$

Figure 15: Underlying assembler-level model of *Init*.

$$\begin{aligned}
& (reg(1) := mem(oloc); reg(2) := 60) \wedge [\Delta Time \mid now' \leq now + i]; \\
& \mathbf{do} \quad reg(1) \geq reg(2) \rightarrow \\
& \quad [\exists Mem; \exists Regs; \Delta Time \mid now' \leq now + g]; \\
& \quad reg(1) := reg(1) - reg(2) \wedge [\Delta Time \mid now' \leq now + (2 - (g + r))] \\
& \quad [\exists Mem; \exists Regs; \Delta Time \mid now' \leq now + r] \\
& \mathbf{od}; \\
& [\exists Mem; \exists Regs; \Delta Time \mid now' \leq now + g]; \\
& mem(oloc) := reg(1) \wedge [\Delta Time \mid now' \leq now + (f - g)];
\end{aligned}$$

Figure 16: Development of the **while** loop.

$$\begin{aligned}
& \exists test, body, exit : addr \mid pc < body < test < exit \bullet \\
& [\exists Mem; \Delta Time; \Delta Regs; \Delta PC \mid reg' = reg \oplus \{1 \mapsto mem(oloc), 2 \mapsto 60\} \\
& \quad \wedge pc' = test \wedge now' \leq now + i]; \\
& \mathbf{do} \\
& \quad pc = test \rightarrow [\exists Mem; \Delta Time; \exists Regs; \Delta PC \mid \\
& \quad \quad now' \leq now + g \\
& \quad \quad \wedge (((reg(1) \geq reg(2)) \wedge pc' = body) \vee \\
& \quad \quad ((reg(1) < reg(2)) \wedge pc' = exit))] \\
& \quad \square \\
& \quad pc = body \rightarrow [\exists Mem; \Delta Time; \Delta Regs; \Delta PC \mid \\
& \quad \quad now' \leq now + (2 - g) \\
& \quad \quad \wedge reg' = reg \oplus \{1 \mapsto (reg(1) - reg(2))\} \\
& \quad \quad \wedge pc' = test] \\
& \mathbf{od}; \\
& [\Delta Mem; \Delta Time; \exists Regs; \Delta PC \mid mem' = mem \oplus \{oloc \mapsto reg(1)\} \\
& \quad \wedge pc = exit \wedge now' \leq now + (f - g)];
\end{aligned}$$

Figure 17: Assembler-level model of the **while** construct.

contribute to satisfaction of the overall timing goal. The formally documented constraints required by the Quartz discipline at least make these relationships explicit.

(Interestingly, if we had not adopted the policy of keeping ‘out’ in a register then the loop body would have developed as shown below:

```
load oloc 1
subtract 1 2
store 1 oloc .
```

Since this code takes $2 + 1 + 2 = 5$ time units the only way to meet the timing requirement in Figure 10 would have been for at least one of g or r to be negative! This impossibility would have made it obvious that the compilation was following the wrong path, even without compiling the remainder of the code.)

The timing requirement of the second action in Figure 17, testing the loop guard, can be strengthened by choosing

$$g = 1 .$$

Letting $exit = test + 1$ then allows the action to be immediately mapped to a conditional branch in our target instruction set:

```
brgte 1 2 body .
```

To complete the compilation the first and last actions in Figure 17 must be developed further. The last one, i.e., the ‘decoding’ action, can have its timing post-condition strengthened with $now' = now + 2$. A refinement to define the change in the program counter as $pc' = pc + 1$ then yields a mapping to a store instruction:

```
store 1 oloc .
```

This step gives a value of

$$f = 2 + g = 2 + 1 = 3$$

for the loop exit overhead introduced in Figure 8 when the time g needed to evaluate the guard for the last time is included. This in turn defines the maximum allowable value for i , the time to reach the loop guard on the first occasion, as 2 (because $i + f < 6$ is required).

The last compilation task is thus to develop to assembler the first action in Figure 17. Given that $i \leq 2$ we face a dilemma because there are three actions to perform, loading the value at location *oloc* into register 1, loading the constant 60 into register 2, and branching to location *test*. Implementing this as a memory load, followed by a load absolute, and a branch, i.e.,

```
load oloc 1
loadabs 60 2
branch test
```

would take $2 + 1 + 1 = 4$ time units and thus violate the timing requirement. Furthermore, there is no optimisation of our development steps from Figure 16 onwards that will help recover the necessary time. No redundant instructions have been generated.

The solution will be to introduce a necessary pre-condition stating that (at least) the first of these values is already in a register when the loop begins. By adding

$$reg(1) = mem(oloc)$$

to the pre-condition of the first action in Figure 16, it will then be unnecessary to load this memory value and we need to generate only

```
loadabs 60 2
branch test
```

for the schema, giving a final value of

$$i = 2$$

and satisfying the last of the programmer’s timing requirements.

Since a pre-condition cannot be arbitrarily strengthened, this must be done by a program transformation rule, which checks that the post-condition of the preceding action satisfies this new condition. In this case the behaviour introduced in Figure 13 does indeed have the necessary post-condition

$$reg'(1) = mem'(oloc) ,$$

so the transformation step is valid.

Finally, collecting the above steps together (and eliminating the logical constant declarations and existential quantifiers, all of which

```

                                load iloc 1
                                store 1 oloc
                                loadabs 60 2
                                branch test
                                body: subtract 1 2
                                test:  brgte 1 2 body
                                exit:  store 1 oloc

```

Figure 18: Time-verified assembler code

are now redundant or instantiated, respectively) gives the final, time-verified assembler-level description which can be displayed in a familiar notation as shown in Figure 18. (Unlike the code in Figure 1, the second instruction here is redundant but cannot be removed due to the programmer’s overspecification stating that the first assignment statement in Figure 11 must take *exactly* 4 time units.)

All of the timing obligations from the HLL program in Figure 11 have been proven for this assembler code and hence, so has the timing behaviour in the specification from Figure 5. Furthermore, in the context of the environment defined by Figure 4, this code will achieve the user’s original requirement from Figure 3.

Related work

A number of contemporary projects have goals that match, or complement, those of Quartz.

The University of York is producing a method for development of real-time systems [14]. It is based on a single wide-spectrum language, TAM, which mixes specification and real-time HLL programming concepts. It has an associated refinement calculus and proof theory. This work is well advanced and the results to date are impressive. However, the approach does not model the compilation process and is thus forced to make assumptions about the execution time of HLL constructs.

Cambridge’s **safemos** project [3] involves development of real-time HLL programs from state-transition based specifications. The meaning of these programs is given by a translation to

sequences of machine instructions, interpreted as if executing on an idealised stack machine. To solve the need to know low-level information in order to reason about real-time behaviour at the HLL program level it uses automated tools to derive “behavioural abstractions” of the behaviour. The methodology, still under development, intends to emphasise “conventional verification plus a verified compiler” coupled with “pure machine code verification”.

The ESPRIT ProCoS II project is also treating compilation of real-time HLL programs formally via a set of translation theorems [10]. An occam-like programming language is augmented with constructs for expressing timing requirements such as worst-case execution times, time-out alternatives and acceptable clock inaccuracies. Machine code programs are represented as sequences of instructions and their meaning defined via their interpretation on a model machine. This project does not target an already existing programming language, nor has it yet modelled the timing behaviour of a real machine.

Augmented high-level language compilers are an obvious way of automating the discharge of timing obligations, relieving programmers from the burden of ‘cycle counting’. A number of compilers and tools that can predict timing behaviour [13] have been proposed, some of which rearrange code in order to improve performance [9].

Indeed, performance-enhancing heuristics for guiding real-time software development are a valuable adjunct to the ‘raw’ Quartz development rules. There is still a great deal of cre-

activity involved in undertaking formal program developments, despite the existence of formal rules. Such heuristics have been discussed in the literature [16], although mainly for ‘soft’, rather than hard, real-time goals.

Conclusion

The Quartz project began in April 1994. It is producing a methodology that encompasses real-time requirements specification, design of high-level language programs and generation of time-verified assembler code. It treats time as a first-class concept, given importance in the development process equal to that of functional behaviour. By modelling the entire software development process within a unified framework it simplifies and increases the precision of timing proofs.

The case study above involved a considerable degree of complexity to develop what ultimately proved to be a small program. Nevertheless, while small, the example was far from trivial. Iterative code typically involves subtle reasoning about loop invariants. In this case we have shown how time is smoothly integrated into this reasoning.

Furthermore, most of the steps were eminently suited to automated support.

- A ‘refinement tool’ or automated theorem prover could assist with application of the HLL program development rules.
- The theory for translation of timed HLL programs to assembler code could be used either to verify an existing compiler or to derive a new, verified real-time compilation strategy.

It would be unusual to undertake development of verified real-time software entirely by hand in practice! Future work will hopefully see the development of such tools from the Quartz theory.

Acknowledgements

We wish to thank John Staples, Andy Wellings and the anonymous referees for their comments on drafts of this article. This work was funded by the Information Technology Division of the Defence Science and Technology Organisation.

References

- [1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Language and Systems*, 16(5):1543–1571, September 1994.
- [2] R.J.R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *CONCUR’94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 367–384. Springer-Verlag, 1994.
- [3] J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
- [4] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [5] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *ACM Workshop on language, compiler and tool support for real-time systems*. ACM Press, 1994.
- [6] C. Fidge. Adding real time to formal program development. In M. Naftalin, T. Denzvir, and M. Bertran, editors, *FME’94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 618–638. Springer-Verlag, 1994.
- [7] C. Fidge, P. Kearney, and M. Utting. Interactively verifying a simple real-time scheduler. In *Proc. Computer-Aided Verification ’95*, Liege, Belgium, July 1995. To appear.
- [8] C. Fidge and A. Wellings. An action-based formal model for concurrent, real-time systems. Technical Report TR 95-1, Software Verification Research Centre, January 1995.
- [9] P. Gopinath, T. Bihari, and R. Gupta. Compiler support for object-oriented real-time software. *IEEE Software*, 9(5):45–50, September 1992.
- [10] H. Jifeng. *Provably Correct Systems*. McGraw-Hill, 1995.
- [11] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [12] T.S. Norvell. Machine code programs are predicates too. In *Proc. Sixth BCS FACS Refinement Workshop*, London, January 1994.
- [13] G. Pospischil, P. Puschner, A. Vrchticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.

- [14] 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.
- [15] M. Utting and P. Kearney. Specification issues for real-time behaviour of RISC processors. In *Proc. Australasian Workshop on Parallel and Real-Time Systems*, pages 400–411, Melbourne, July 1994. Victoria University of Technology.
- [16] M.T. Vandevoorde. Specifications can make programs run faster. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 1993.
- [17] J.B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.