

A Formal Method for Building Concurrent 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. It is a program refinement theory that supports systematic production of verified real-time code. It uses a wide-spectrum model that 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 example.

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 Quartz method via a detailed example.

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 constraints demands a great deal of rigour from system developers. Real-time software is thus expensive to manufacture to the degree of

Action systems

Action systems [1] are one of several similar models that have recently emerged for formally modelling concurrent behaviour. They use an interleaving model to conservatively build on the experience gained with sequential specification and development methods.

An action system is represented as a loop,

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

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

$$Guard \rightarrow Body$$

with a guard that defines the environmental ‘assumptions’ under which the action may occur and a body which defines its ‘effect’. This separation of concerns is essential for defining the behaviour of systems that react with their environment. The semantics of such a loop is determined by the set of traces it may exhibit, where each trace is a history of externally observable state changes.

Traditionally such a loop describes sequential behaviour, 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, a concurrent implementation with variables shared between mutually-exclusive actions, or a parallel implementation with process interaction represented by joint actions shared among processes.

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.

- Formal specification languages such as Z [13] have proven flexible enough to describe behaviour at many levels of abstraction. Agreed-upon notations for

expressing concurrency and timing are still lacking, but simple extensions such as ‘action systems’ allow reactive systems to be modelled (see box).

- ‘Refinement’ methods offer formal rules for deriving a high-level language (HLL) [8] or assembler [9] program from its specification; program development and verification thus proceed in lockstep. To date these techniques have been limited to sequential, untimed programs, but new rules are appearing for concurrent and real-time systems.
- Recent proposals have extended HLL program refinement methods with low-level real-time proof obligations [3]. Although the accuracy of such methods is limited by the known timing properties of the target compiler, architecture and operating system, we have shown elsewhere that the timing behaviour of modern computer architectures can be formally described and automated proofs of assembler level programs can be undertaken using such a model [4].

Overview

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

1. Real-time specification. 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 response time under 130 time units.
2. Real-time program development. 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 59, the programmer has recognised that there may be as little as 2 time units available for each iteration; the timing is tight but not impossible!
3. High-level language program and timing constraints. 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 available to evaluate the loop guard g , execute the subtraction statement in the loop s , and return to the beginning of the loop r , may be no more than 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. Verified compilation strategy. Program compilation takes place using code generation rules proven to respect stated timing constraints. Timing obligations

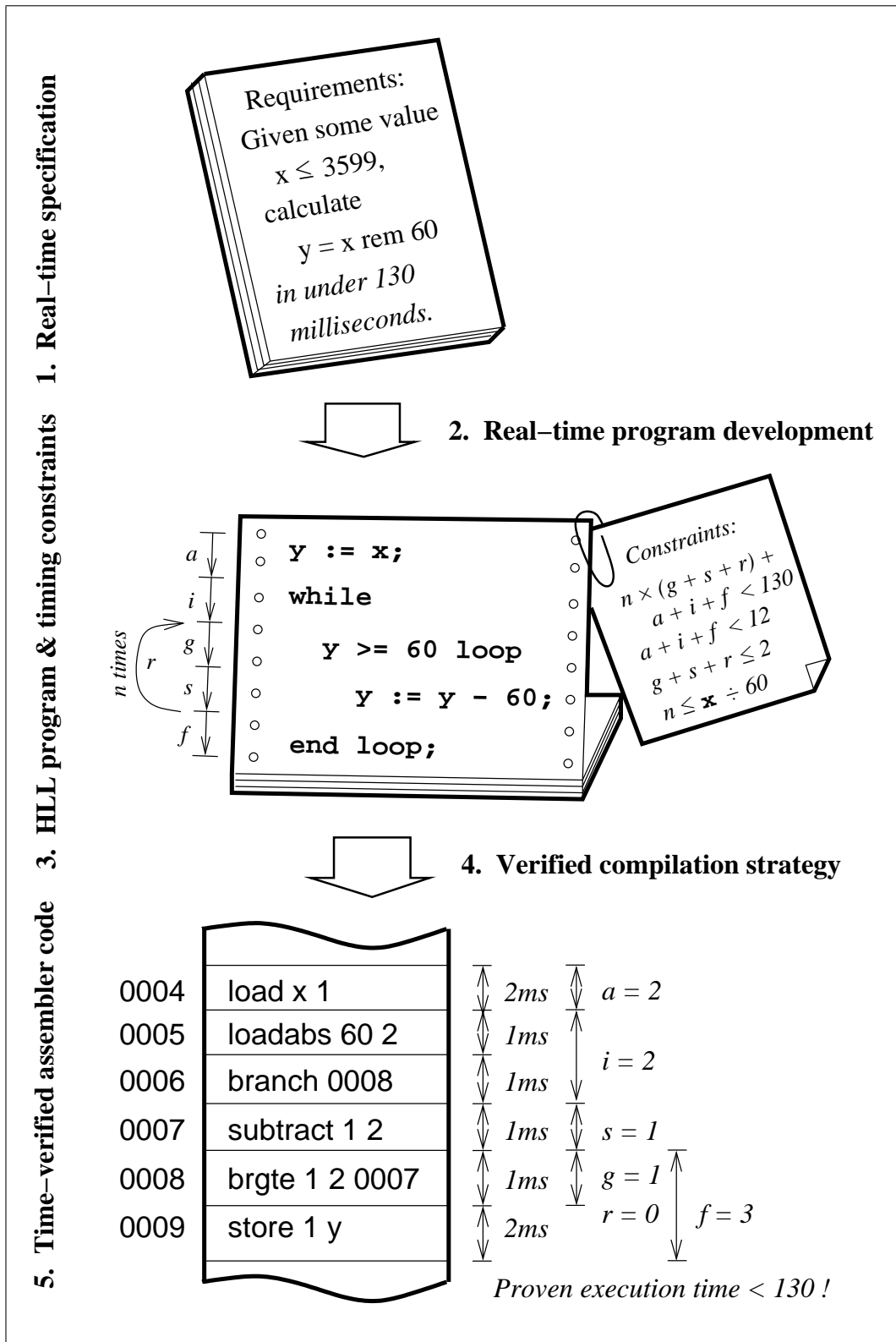


Figure 1: Overview of the Quartz method

Real-time refinement rules

Program refinement is a formal method of system development in which abstract ‘specifications’ are translated into more concrete ‘implementations’ by repeated application of mathematically-based rules [8]. The rules include side-conditions that ensure their application results in verifiably correct system development at each step. Typically each ‘implementation’ is a more detailed and deterministic development of the preceding ‘specification’.

In the Quartz method refinement rules are applied at three distinct levels.

1. Concurrent components. At the highest level the system specification defines the behaviour of observable system variables over all time. Concurrent variable ‘histories’ are represented as a trace where ‘time’ is the trace index. The refinement rules used at this level manipulate whole traces; each development step is valid only if the set of traces allowed by the implementation is a subset of those defined by the preceding specification [1]. At this level the system description is partitioned into concurrent behaviours, from which a set of skeletal processes in the target programming language can be identified [5].
2. Sequential development. Each concurrent component can then be refined separately using familiar sequential refinement rules [8, 13]. This stage refines the individual observable state changes that, in sequence, create the traces specified above. The current time in each state is represented via an auxiliary specification variable which is manipulated by the refinement rules like any other. Ultimately this stage of development results in a description expressed in an ‘executable subset’ of the specification notation that maps to high-level language programming constructs. Many side conditions involving timing behaviour may not yet have been fully discharged, however.
3. Low-level operations. Data refinement is a generalised form of program refinement that allows local system variables, and the operations applied to them, to be replaced with other variables and operations [8, 13]. These rules are used to refine each sequential high-level language description to a lower-level one in a more primitive executable subset that corresponds to assembler language constructs. High-level language variables and operations are replaced by the processor-specific features that implement them. All timing side-conditions can then be discharged against the timing behaviour for primitives defined in the assembler-level model [4].

are fully discharged against a precise model of the target processor 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. Time-verified assembler code. 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 125 time units, thus satisfying the original requirements specification.

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 modelling and 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 processor¹ as the final executable code.

Detailed example

The following example illustrates how the Quartz methodology is applied. It shows the major steps needed during development of a small embedded program.

Our 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 3599 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 59; variable *out* will be used to display the number of seconds that have elapsed in the current minute. Inputs are not buffered so, in order to keep up with the worst-case input frequency, each output must be computed in under 135 milliseconds.

This requirement is shown 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.

¹MIPS is a trademark of MIPS Computer Systems, Inc.

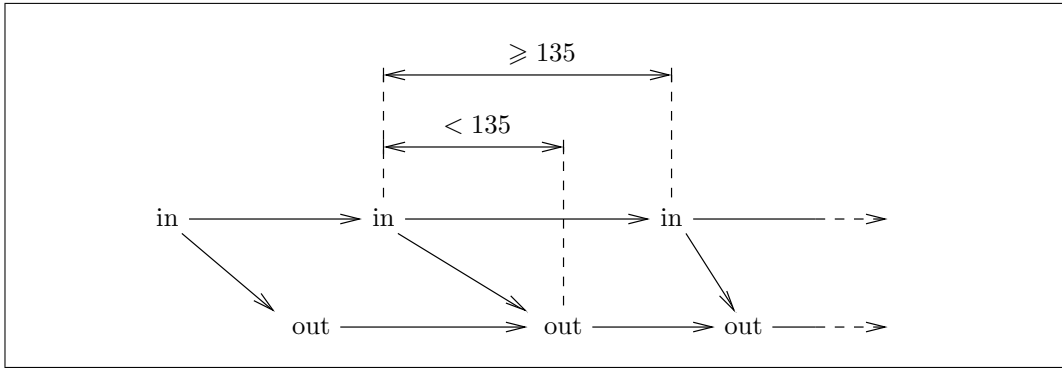


Figure 2: Desired timing behaviour.

(Elsewhere we have shown how to handle more realistic timing primitives, including those with ‘uncertain’ timing [3].) Furthermore, to avoid too easy a solution, the target machine has no multiplication or division capabilities.

Real-time specification. Figure 3 is a Z [13] specification representing the requirement via the allowable traces of values appearing in the variables and the times at which the values appear. A discrete notion of time is sufficient for this example, so absolute time is defined to be the natural numbers.

Schema *Environ* defines the incoming data stream. It declares a ‘history’ *inHist* of values *in* and their arrival times *inT*. The predicate formalises our expectation that all distinct inputs are in the range 0 to 3599 and 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 declares a trace *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.

Real-time program development. From such a specification we can devise an overall system design that achieves this behaviour.

Figure 4 is an action system representation of the above specification. Initially the two time variables are asserted to be zero. Future behaviour is then defined via two atomic actions.

In the first, which can occur only when there are no unprocessed inputs, some new input value in the appropriate range 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* are free to change. Notation ‘ $\exists Out$ ’ says that none of the variables in schema *Out* change.)

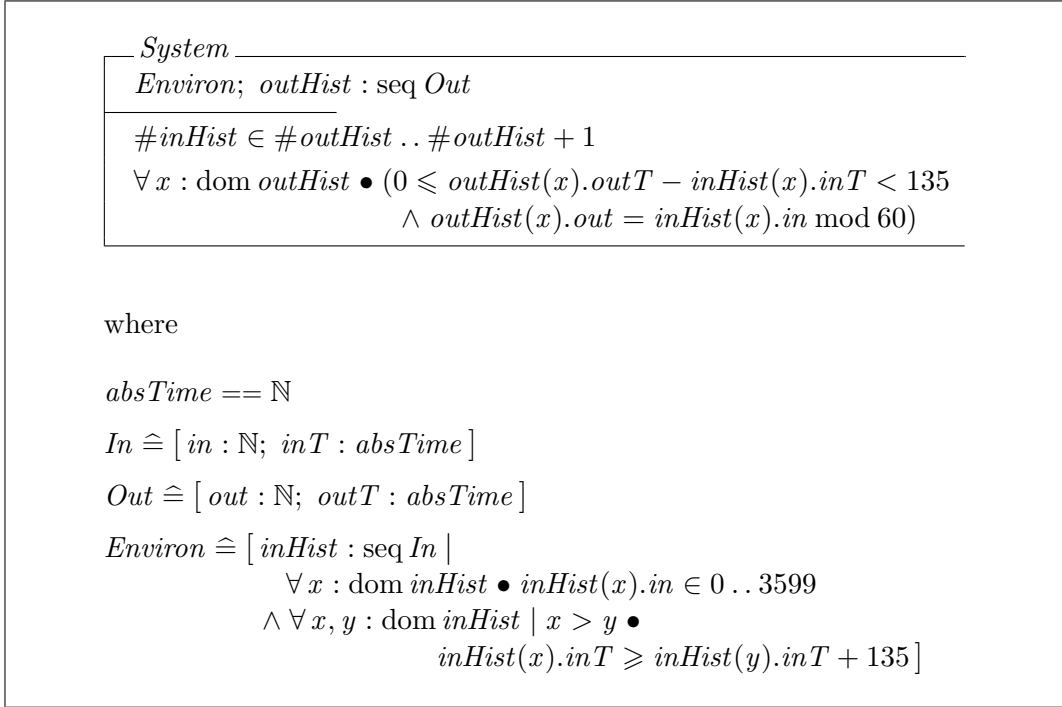


Figure 3: Real-time specification.

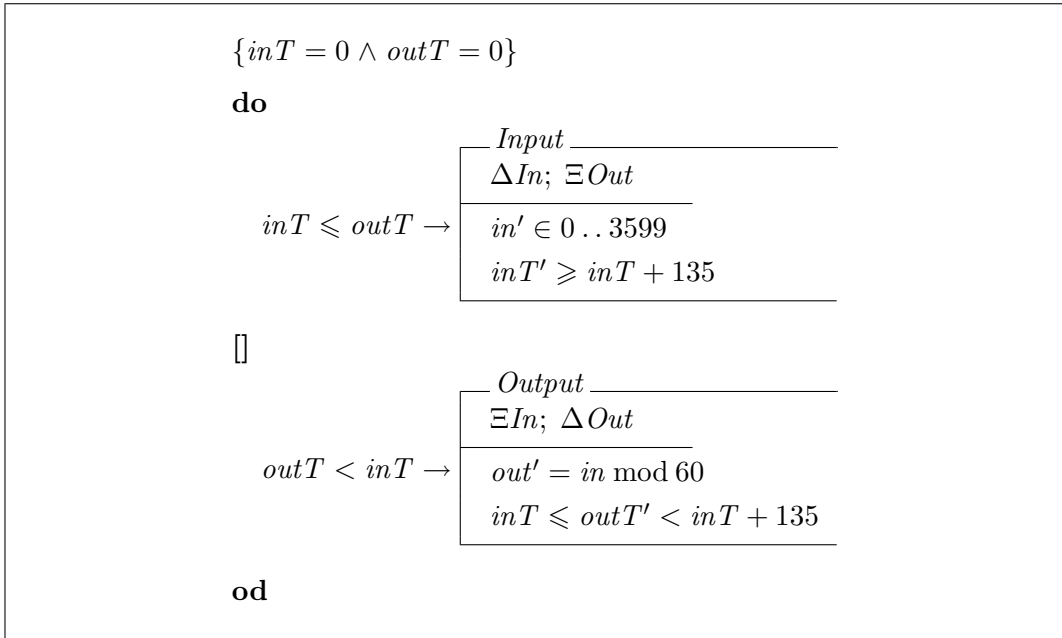


Figure 4: Development of *System* as two actions.

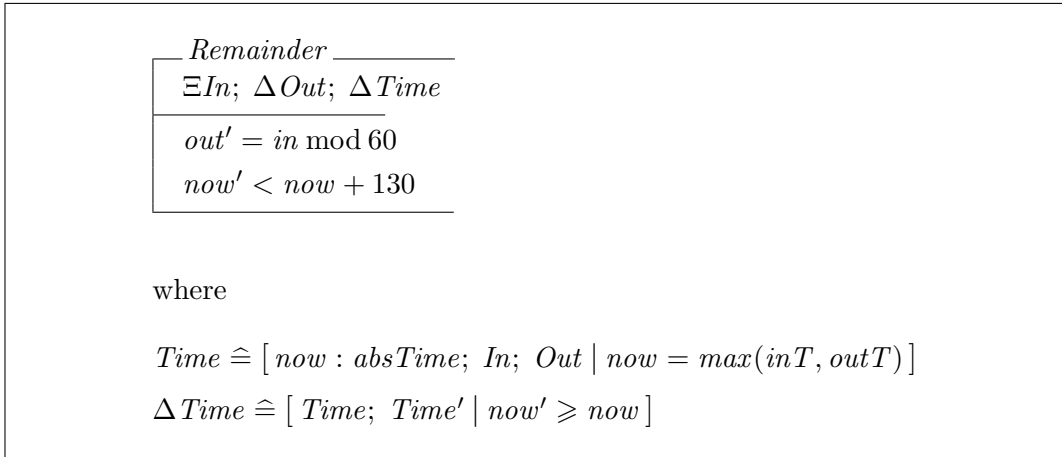


Figure 5: Development of *Output* as a specification with duration.

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.

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 [13].

Furthermore, the *Input* and *Output* abstractions do not necessarily define an execution-time ‘duration’ for operations 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 processor time, merely that an implementation must perform state changes that are observable at times inT and inT' .

Formal proof that Figure 4 is a valid development of Figure 3 follows from the definition of the trace-semantics underlying action systems [1]. The proof obligation is to show that the next-state relation defined by actions *Input* and *Output* can only create traces of the declared variables that obey the constraints expressed by *Environ* and *System*.

At this point the separate actions can be ‘refined’ to sequential high-level programming language code. Figure 5 shows a particular development of the *Output* behaviour. We have assumed 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 small delay accounts for the synchronisation overhead between the *Input* and *Output* actions, e.g., for context switching. The two actions are mutually exclusive,

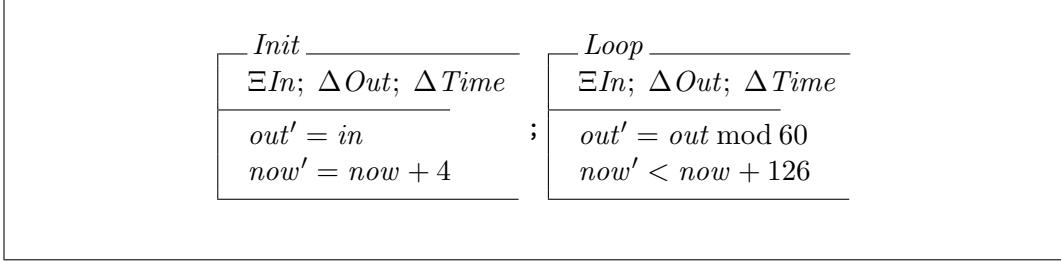


Figure 6: A development of *Remainder* by introducing sequential composition (the timing constraint on *Init* is overspecified).

since they share variable *in*, and execute in a strict interleaving. (In more complex situations we use real-time scheduling theory to confirm the correctness of such specifications.)

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 requirement, i.e., the difference between its finishing and starting times, as being less than 130 milliseconds.

Time variable *now* is introduced here 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 $\Delta \textit{Time}$ so that *now* cannot go backwards.

Well-known sequential refinement rules [8, 13] are used from this point onwards. In this instance development of schema *Remainder* from *Output* is justified in terms of traditional ‘add new variable’ and ‘strengthen post-condition’ rules.

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. (This intermediate state does not appear in the traces defined by the action system of Figure 4, where the actions are considered atomic, so there is no need to justify this step against the trace model; sequential refinement rules are sufficient.)

In this step the proof obligation requires 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 memory access takes 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* requirement 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 de-

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

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

con $out0 = out \bullet$
con $now0 = now \bullet$
 $\exists i, f : absTime \mid i + f < 8 \bullet$

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

where

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

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

velopment of the specifications. An implementation of *Init* is now obliged to take exactly 4 time units, preventing potential optimisations that might perform the task 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 constraint in Figure 6 is not ‘wrong’, specifying $now' \leq now + 4$ 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 wasting effort 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 later code rearrangement.

The *Init* operation, which merely sets *out* to equal *in*, can be easily mapped to the HLL program shown in Figure 7 since it is of a form corresponding to a specification ‘template’ for assignment statements. The program code itself is accompanied by the programmer’s specified timing behaviour, acting as an undischarged proof obligation, because we have not yet *proven* that the assignment can be implemented in the specified time.

Development of the *Loop* requirement is more challenging. Proving real-time properties of iterative code typically involves identifying a ‘timed’ loop invariant [3], 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 [8]. The loop itself is re-expressed as an operation 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 its initial value *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! Furthermore, 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 3599, the programmer can determine the worst case number of iterations,

$$n \leq 59.$$

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 ‘imaginary’ 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 take, 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 under 126 milliseconds, and knowing that there may be up to 59 iterations, the programmer has stated that the overall timing constraint can be satisfied if the loop entry and exit overheads are under 8 time units and each iteration takes at most 2 time units. By defining this timing invariant the programmer has partitioned the timing constraint 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 processor *can* meet the requirement.

A specification in such a form can be readily translated into equivalent HLL code as shown in Figure 9 [3]. Two new invariants are easily calculated from *invL* by 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, so the initial overhead *i* must be included.)

The loop body *Subtract* can then be mapped to the target programming language as shown in Figure 10. The specific timing constraint was devised from the ‘difference’ between invariants *invE* and *invB*. If the duration of the loop body is as specified, then the two timing invariants are guaranteed to be respected. Obviously the timing

```

 $\exists g, r : \text{absTime} \mid g + r \leq 2 \bullet$ 
while out  $\geq$  60 loop
   $\frac{\text{Subtract} \quad \text{---}}{\text{invB}; \text{invE}'; \exists In}$ 
   $\frac{\text{---}}{\text{out}' = \text{out} - 60}$  ;
end loop;

where

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

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

```

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

```

out := out - 60  $\wedge$  [  $\Delta$ Time  $\mid \text{now}' \leq \text{now} + (2 - (g + r))$  ] ;

```

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

constraint on the loop body is very demanding, requiring the subtraction to be performed in no more than 2 time units, and possibly less depending on the values of g and r !

High-level language program and timing constraints. 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 constraints 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 define the allowable overheads of implementing the **while** construct.

Verified compilation strategy. Development of executable assembler code from the HLL program in Figure 11 proceeds by systematic application of data refinement rules.

Consider the first assignment statement. 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’ [13] can thus be performed to map this statement to its intended effect on physical memory, as shown

$out := in$	$\wedge [\Delta Time \mid now' = now + 4];$
	con $out0 = out; now0 = now \bullet$
	$\exists i, f, g, r : absTime \mid i + f < 8 \wedge g + r \leq 2 \bullet$
while $out \geq 60$ loop	
$out := out - 60$	$\wedge [\Delta Time; invB; invE' \mid now' \leq now + (2 - (g + r))];$
end loop	$\wedge [\Delta Time; invL; invL'];$

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

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

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

in Figure 12. Memory is defined as a mapping from addresses to values and *oloc* and *iloc* are constants representing the memory addresses allocated for variables *out* and *in* respectively. (The assignment statement is formally defined so that the values of all variables and fields are unchanged unless specifically mentioned on the left-hand side.)

Further data refinements introduce other machine-specific features. An array of CPU registers is added next. Figure 13 shows how the update to memory can be achieved via two operations 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 constraint has been divided evenly between the two requirements.

The assembler-level development of *Init* is completed by introducing the program counter which, if incremented by each operation, allows the description in Figure 13 to be mapped into two executable assembler instructions

```
load iloc 1
store 1 oloc
```

using our definitions of the effect of load and store instructions and their execution times. Referring to the overview in Figure 1 this tells us

$$a = 2 + 2 = 4$$

$reg(1) := mem(iloc) \wedge [\Delta Time \mid now' = now + 2];$
$mem(oloc) := reg(1) \wedge [\Delta Time \mid now' = now + 2]$

Figure 13: Development of *Init* using registers.

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

where

$$\begin{aligned}
Memory & \hat{=} [mem : address \rightarrow value] \\
Regs & \hat{=} [reg : regnum \rightarrow value] \\
PC & \hat{=} [pc : address]
\end{aligned}$$

Figure 14: Underlying assembler-level model of *Init*.

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

Figure 15: Development of the **while** loop.

in this case.

Formally these two assembler instructions are defined in the Quartz model as shown in Figure 14. Obviously the degree of complexity has increased dramatically as the level of abstraction has lowered, even for the simple target machine assumed for this example. Accurately modelling a real machine, with features such as pipelining, caching, interrupts, etc. is possible [4], but managing the resulting detail poses a formidable challenge.

Development of the while loop follows similarly, again using data refinement rules. Figure 15 shows an intermediate description of the loop, following introduction of the memory and register variables. This complex description is mid-way 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 directly 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.

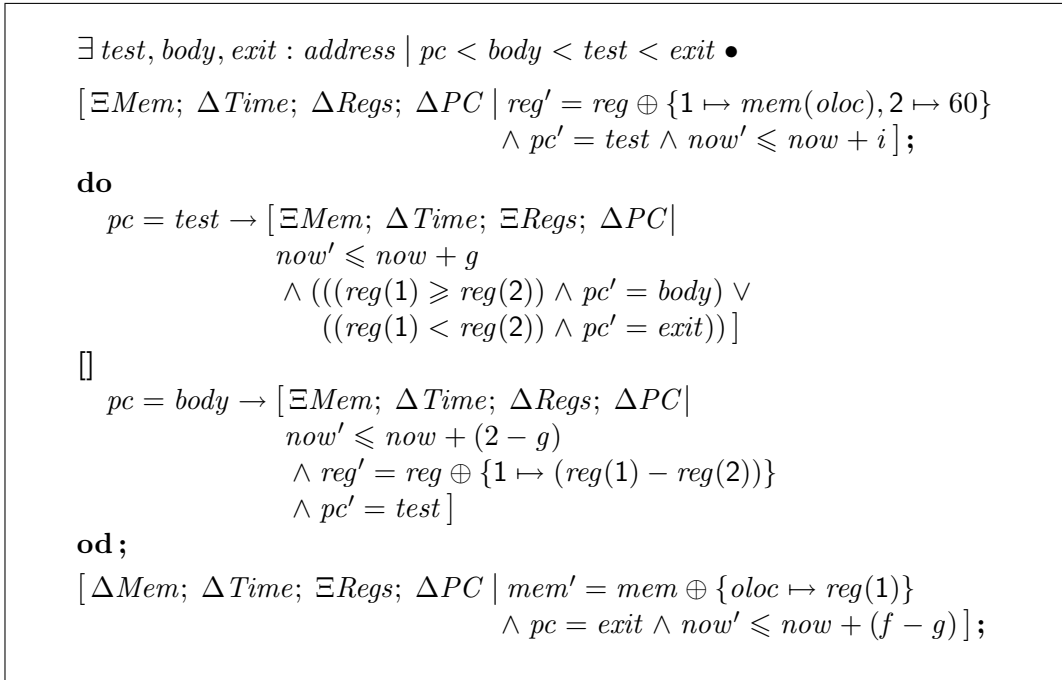


Figure 16: Assembler-level model of the **while** construct.

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 as part of the data refinement by introducing ‘encoding’ and ‘decoding’ operations [8] 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 refinement thus replaces references to ‘*out*’ with ‘*reg(1)*’. Similarly, the constant ‘60’ has been recognised as a ‘loop-constant expression’ [12] and is therefore ‘loaded’ into register 2 by the first requirement 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. Our action system model thus serves to model both top-level concurrency and low-level assembler execution. The stated durations have been contrived to ensure that the programmer’s invariants from Figure 11 are maintained, as required by the data refinement proof obligations. The operation immediately after the loop in Figure 15 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 16 does this, for a particular compilation strategy in which the guard test is placed immediately after the loop body.

The specified behaviours manipulate the program counter in order to explicitly achieve the desired iteration. Initially the part before the **do** statement sets the program counter to *test*, the address at which the code for testing the guard is to

be placed. The first action inside the **do** statement 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 second action is the loop body itself which ends with the program counter equal *test* 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 were omitted from Figure 16.

Development of the ‘loop body’ action from Figure 16 can be finished straightforwardly. Its specified timing behaviour can be strengthened by adding $now' = now + 1$. Letting $test = body + 1$, it then corresponds to a subtract instruction in the target assembler language:

```
subtract 1 2 .
```

Referring to the overview in Figure 1 this tells us

$$s = 1 .$$

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

$$g + r \leq 1 .$$

Therefore the ‘loop guard’ action in Figure 16 has to execute in one time unit or less. Again the timing goal is very demanding, but not yet impossible.

These inter-relationships between time constants highlight another 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 program development. Here the loop test and body cannot be refined in isolation because their individual timing behaviours both contribute to satisfaction of the overall timing goal. The formally documented constraints required by the Quartz discipline make these relationships explicit.

(Interestingly, if we had not adopted the policy of keeping ‘*out*’ in a register, then the loop body would have been refined to

```
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 constraint 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 development was following the wrong path, even without compiling the remainder of the code.)

The timing constraint on the loop guard action in Figure 16 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 executable specification subset:

```
brgte 1 2 body .
```

To complete the compilation the first and last operations in Figure 16 must be developed further. The last one, i.e., the ‘decoding’ requirement, can have its timing post-condition strengthened with $now' = now + 2$ and $pc' = pc + 1$, thus yielding the specification of a store instruction:

```
store 1 oloc .
```

This step gives a value for the loop exit overhead introduced in Figure 8 of

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

This in turn defines the maximum allowable value for i , the time to reach the loop guard on the first occasion, as 4 because $i + f < 8$ is required.

The last compilation task is to refine the encoding specification in Figure 16. There are three requirements to meet, 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
```

takes $2 + 1 + 1 = 4$ time units and thus satisfies the last timing constraint.

As a final enhancement, however, we observe that the first of these instructions is redundant because the value loaded was already left in register 1 by the code in Figure 13. The redundancy can be removed by emulating the behaviour of an optimising compiler and introducing a necessary pre-condition stating that the value is already in a register when the loop begins. By adding

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

as a pre-condition of the first requirement in Figure 16, it is then unnecessary to load this memory value and only the last two instructions above need be generated, giving an initial loop overhead of

$$i = 2 .$$

However, since there is no refinement rule that allows a pre-condition to be arbitrarily strengthened [8], this must be done by a ‘program transformation’ rule, derived from our basic real-time refinement rules, which checks that the post-condition of the preceding specification satisfies this new condition. In this case the first behaviour specified in Figure 14 does indeed have the necessary post-condition

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

so the transformation step is valid.

```

                                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 17: Time-verified assembler code

Time-verified assembler code. Finally, collecting the above steps together (and eliminating the logical constant declarations and existentially quantified variables, all of which 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 17.

All of the timing obligations for the high-level language program in Figure 11 have been proven for this assembler code and hence, so has the timing behaviour in the specification from Figure 5. In the worst case the execution time is

$$\begin{aligned}
 & n \times (g + s + r) + a + i + f \\
 & = 59 \times (1 + 1 + 0) + 4 + 2 + 3 \\
 & = 127 \\
 & < 130
 \end{aligned}$$

as required! (Our final code is not as optimal as that in Figure 1; the second instruction in Figure 17 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.) Furthermore, in the context of the environment defined by Figure 4, this implementation 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 refinement of real-time systems, based on a wide-spectrum language TAM [11]. However, the approach does not model the compilation process and is thus forced to make simplistic assumptions about the execution time of HLL constructs.

Cambridge’s *safemos* project [2] 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 of HLL programs it proposes using automated tools to derive “behavioural abstractions”. The methodology intends to emphasise

“conventional verification plus a verified compiler” coupled with “pure machine code verification”.

The ESPRIT ProCoS project is also treating compilation of real-time programs formally via a set of translation theorems [7]. An occam-like programming language is augmented with constructs for expressing timing constraints such as worst-case execution times, timeout 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 or processor.

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 [10] have been proposed, some of which rearrange code in order to improve performance [6]. 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 creativity involved in undertaking formal program developments, despite the existence of formal rules. Such heuristics have been discussed in the literature [12], although mainly for ‘soft’, rather than hard, real-time goals.

Conclusion

The Quartz project began in April 1994. It is producing a formal method 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 increases the precision of verified real-time code development.

The example illustrated in this article involved a considerable degree of complexity to produce 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 compiler.

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 *Software* referees for their many helpful comments on drafts of this article. This work was funded by the

Information Technology Division of the Australian Defence Science and Technology Organisation.

References

- [1] 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.
- [2] J. Bowen, editor. *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*. Elsevier, 1994.
- [3] C. J. Fidge. Adding real time to formal program development. In M. Nafatalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 618–638. Springer-Verlag, 1994.
- [4] C. J. Fidge, P. Kearney, and M. Utting. Interactively verifying a simple real-time scheduler. In P. Wolper, editor, *Computer-Aided Verification: 7th International Conference, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 395–408, Liège, Belgium, July 1995. Springer-Verlag.
- [5] C. J. Fidge, M. Utting, P. Kearney, and I. J. Hayes. Integrating real-time scheduling theory and program refinement. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 327–346. Springer-Verlag, 1996.
- [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] Jifeng He. *Provably Correct Systems: Modelling of Communication Languages and Design of Optimized Compilers*. McGraw-Hill, 1995.
- [8] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [9] T. S. Norvell. Machine code programs are predicates too. In D. Till, editor, *Sixth Refinement Workshop*, pages 188–204. Springer-Verlag, 1994.
- [10] G. Pospischil, P. Puschner, A. Vrhoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.
- [11] D. Scholefield, H. Zedan, and He 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.
- [12] 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.

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