A Formal Method for Building Concurrent Real-Time Software

Colin Fidge* Peter Kearney† Mark Utting‡

Introduction

Quartz is a formal development method for concurrent real-time programs. It comprises

- real-time requirements specification in a variant of the Z notation,
- concurrent and sequential program development using formal refinement rules, and
- construction of Ada-like high-level language programs annotated with precise timing constraints.

This article introduces some of the features of the Quartz method via a detailed example.

Background

Motivation. Development of concurrent real-time programs is among the greatest challenges currently facing computer science [1]. Such programs are needed for safety-critical systems, so guaranteeing their correctness is vital.

Satisfying precise timing constraints demands a great deal of rigour from programmers. Real-time software is thus expensive to manufacture to the

*Software Verification Research Centre, School of Information Technology, The University of Queensland, Queensland, Australia. E-mail: `cjf@cs.uq.edu.au`

†Software Verification Research Centre, School of Information Technology, The University of Queensland, Queensland, Australia. E-mail: `pk@cs.uq.edu.au`

‡Department of Computer Science, School of Computing and Mathematical Sciences, The University of Waikato, Hamilton, New Zealand. E-mail: `marku@cs.waikato.ac.nz`

degree of timing predictability needed. Formal methods of program specification and development (refinement) introduce mathematical rigour to software development, so new international standards are mandating their use for high-integrity applications [2].

Nevertheless, formalisms that embrace both real-time and concurrency requirements are only just emerging. Many formal specification and programming language notations have been proposed for real-time systems yet few development approaches link the two. The Quartz project is integrating and extending this past work to produce a viable method.

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

- Formal specification languages such as Z 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) 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.

In this article we demonstrate how these emerging methods can be used together in development of a real-time program.

Overview

The Quartz method encompasses real-time software development from formal requirements specification through to high-level language code.

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.

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.

High-level language program and timing constraints. The resulting HLL code is accompanied by precise 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.

Program refinement tanslates abstract 'specifications' into more concrete 'implementations' by repeated application of mathematically-based rules. The Quartz method is 'integrated' in that it seeks to use the same refinement formalism at all levels of abstraction. In practice refinement in Quartz occurs at two distinct, but interrelated, levels.

Concurrent components. At the highest level a system design is constructed in terms of a number of parallel, interacting components. The top-level specification defines the behaviour of observable system variables over all time, via their allowable traces (histories). Refinement rules used at this level manipulate whole traces [3]. The absolute time domain acts as an index to the traces. The requirement is refined to a set of parallel component specifications, from which a skeletal program design in the target programming language can be extracted [4].

Sequential development. Each concurrent component is then independently refined using rules similar to well-known sequential refinement rules [5]. This introduces the low-level state changes that, in sequence, create the traces specified above. Time can be represented via an auxiliary specification variable which is manipulated by the refinement rules like any other. This stage of development results in a description expressed in an 'executable subset' of the specification notation that corresponds to high-level language programming constructs [6]. Many side conditions involving execution times may not yet have been fully discharged, however, and are subject to further timing analysis [7].

Example

The following example shows the major steps involved during development of a small embedded program using the Quartz methodology.

Requirement. The goal is to develop an embedded program that processes data from sporadic inputs quickly enough so that no inputs are lost. Values in the range 0 to 3599 appear at irregular intervals, in variable '*in*',

with a minimum separation of 135 milliseconds. These values represent the number of seconds that have elapsed in the current 'wall time' hour. The program must divide each such 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.

To avoid too easy a solution, assume the target programming language has no 'remainder' operator (or that the implementation of division is known to be too slow for our needs).

Real-time specification. Figure 1 is a Z [8] specification representing the requirement via the allowable traces of variable values and the times at which the values appear. (A Z schema contains two parts, declarations and a predicate. It may appear as a named box, or in a horizontal format. A schema may 'include' another by naming it in the declaration part.)

A discrete notion of time is sufficient for this example, so the absolute time domain $\mathbb{T}$ is defined to be the natural numbers. Schema *In* tells us that each input value *in* is a number between 0 and 3599, and is accompanied by a timestamp *inTime* recording its moment of arrival. Similarly, schema *Out* declares each output value *out* to be a natural number, timestamped by *outTime*.

Schema *Environ* defines the incoming data stream. It declares a sequential 'history' *inHist* of timestamped input values. The predicate states our environmental assumption that all distinct inputs have their timestamps separated by at least 135 milliseconds.

In such an environment our goal is to then develop a program which will produce outputs as defined by schema *System*. It declares a trace *outHist* of timestamped output values. The predicate has two parts. The first says that the size of the history of incoming values can only ever exceed that of the outgoing values by at most one. The second part defines the desired link between the input and output values. It says that each output value must appear within 135 milliseconds of the corresponding input, and that each output value is the remainder of dividing the input value by 60.

Concurrent component development. From such a specification we can devise an overall system design that achieves this behaviour, as shown by the action system in Figure 2.

Initially the two timestamping variables *inTime* and *outTime* are asserted to be zero. Future behaviour is then defined via two atomic actions, *Input* and *Output*.

Input can occur only when there are no unprocessed inputs, i.e., when the timestamp for the latest output is as great as that for the latest input. It implicitly produces a new input value—*in* is allowed to change to any value in its type. (By Z convention, a primed variable name, e.g., $inTime'$, denotes a final value, and an unprimed one an initial value. The 'ΔIn' notation tells us that *in* and *inTime* are both free to change. Notation 'ΞOut' says that none of the variables in schema *Out* change.) Furthermore, the predicate tells us that the timestamp associated with this event must exceed its previous value by at least 135.

Action *Output*, which may occur only when the most recent input has not yet resulted in a corresponding output, calculates a new value for *out*, and does so within 135 milliseconds of the moment when the most recent input was produced.

The system description in Figure 2 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 system could be implemented by a sequential interleaving of input and output actions, or by two appropriately synchronised concurrent actions [3].

At this level of abstraction constructs such as '**do**...**od**' are logical only. They belong to the action system model, not the target programming language, and thus incur no run-time overheads. The particular notation used here mixes Z schemata with guarded command language constructs [8].

The *Input* and *Output* abstractions do not necessarily define an execution-time duration for the operations that will implement them. Time variables *inTime* and *outTime* merely mark deadlines by which significant 'events' (externally-observable state changes) take place, *not* the passage of processor time. Thus the difference between *inTime* and $inTime'$ 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 these times.

Formal proof that Figure 2 is a valid development of Figure 1 follows from the definition of the trace-semantics underlying action systems [3]. The proof obligation is to show that the next-state relation defined by actions *Input* and *Output* can only create histories that obey the constraints expressed by *Environ* and *System*.

This simple example involved only two parallel actions. In more complex

applications, however, we introduce explicit models of multi-tasking components, such as periodic and sporadic tasks, protected shared variables, and the run-time scheduler [9]. Real-time scheduling theory is then invoked to discharge the proof obligation [4].

Sequential code development. At this point the concurrent components are independently refined to sequential high-level programming language code. Figure 3 shows a particular development of the *Output* action. We have assumed a 'shared variable' mapping [3] of the system in Figure 2 to a physical configuration in which the *Output* action is released to execute, without preemption, no more than 5 milliseconds from the arrival of each input. This small delay accounts for the synchronisation and context switching overhead between the *Input* and *Output* actions. The two actions must be mutually exclusive, since they share variable *in*, so execute in a strict interleaving.

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 in schema *Time* to denote the passage of available processor time. It is linked to the two time markers *inTime* and *outTime*, always being the larger of the two when an observable event occurs. Figure 3 also constrains $\Delta \mathit{Time}$ so that *now* cannot go backwards.

Well-known sequential refinement rules [8] can be 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 postcondition' rules.

Knowing that the target language has no suitable division capability, the programmer now decides to calculate the remainder via repeated subtraction. Anticipating the iterative code, the first step is to split the problem into two sequential components as shown in Figure 4. 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 2, where action *Output* is 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 the sum of the execution times for the two components to be compatible with the overall execution time specified in *Remainder*. A decision was made to specify the execution time of *Init* as no more than 4 milliseconds, and therfore the *Loop* requirement has less than 126 milliseconds available. This illustrates the way that timing considerations create interdependencies between otherwise independent components. For this reason, it is usually best to avoid introducing *particular* timing values too early.

The *Init* operation, which merely sets *out* to equal *in*, can be readily mapped to the HLL program shown in Figure 5 since it is of a form corresponding to a specification 'template' for assignment statements. The program code on the left is still accompanied by the timing specification, acting as an undischarged proof obligation, because we have not yet *proven* that the assignment can be executed 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 [10], as shown in Figure 6. 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. The loop itself is re-expressed as an operation that maintains invariant *invLoop*, 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 required remainder of integer division. Furthermore, in order to make progress towards termination while maintaining the invariant, *out* must decrease by at least 60 with each iteration. Knowing that the maximum value of *in* is 3599, the programmer can then determine the worst case number of iterations n to be 59.

The second conjunct of invariant *invLoop* defines the maximum time that can have elapsed, since $now0$, as a function of n and two time constants, i and f. These constants denote the timing overheads associated with entering and exiting a loop statement. Unlike the 'imaginary' iteration construct used in Figure 2, 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, to be instantiated

later.

Interestingly, 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 milliseconds and each iteration takes at most 2 milliseconds. 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 7 [10]. Two new invariants are easily calculated from *invLoop* 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 *invStart* is always true at the moment the loop body is about to start executing, because the guard has been evaluated one more time than the body itself has executed. Invariant *invEnd* 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.

Loop body *Subtract* can then be mapped to the target programming language as shown in Figure 8. The specific timing constraint was devised from the 'difference' between invariants *invEnd* and *invStart*. If the duration of the loop body is no more than that specified, then the two timing invariants are guaranteed to be respected. (The timing constraint on the loop body is quite demanding, requiring the subtraction to be performed in no more than 2 milliseconds, and possibly less depending on the values of g and r.)

High-level language program and timing constraints. Figure 9 collects the above steps together and shows the final HLL program developed from the specification in Figure 3, in an Ada-like syntax augmented with

timing specifications. Lines 1, 5, 6 and 9 are executable statements defining the functional behaviour of the program. Initially it sets *out* to equal *in*, then repeatedly subtracts 60 from *out* until it no longer exceeds 60.

The remaining lines (2, 3, 4, 7, 8 and 10) represent as-yet-unproven timing constraints discovered during the course of the refinement. For instance, line 2 is conjoined to the assignment statement on line 1 to tell us that this statement may take no more than 4 milliseconds. Similarly, lines 7 and 8 constrain the final implementation of the code on line 6 to satisfy invariants *inStart* and *invEnd*, and so on.

The timing variables and constants, e.g., *now* and *out*0, are logical and serve to define proof obligations only. No memory space will be required for them. Similarly, no object code will be generated for the various timing constraints. However, the program cannot be considered complete until it has been formally *proven* that the final machine code generated for the functional statements will always satisfy these constraints.

How this final timing analysis is done would fill another article! We briefly note, however, that timing prediction methods and tools are available that can potentially be applied to time-decorated HLL programs such as that produced above. These work either experimentally [11], or using proof techniques [12]. Such timing analyses can be undertaken at compile time [13], or with a special purpose tool [14]. Furthermore, new timing analysis techniques are on the horizon that will potentially make discharge of these timing properties quite accurate, even for pipelined RISC architectures [15].

Conclusion

The Quartz method treats time as a first-class concept, given importance in the development process equal to that of functional behaviour. By modelling the software development process within a unified framework it leads to increased confidence in the correctness of real-time code development.

The example shown 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 the dimension of time is smoothly integrated into this process.

The example also concluded with a number of precise, but unproven, timing obligations on the final machine code. In a companion project we are developing a refinement-based formalism for high-level language program

compilation that will be used to prove such obligations against a model of the target architecture [7]. Coupled with the results described in this article, such research promises to give us a complete formal development path from real-time specifications to executable time-verified machine code.

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] K. G. Shin and P. Ramanathan. Real-time computing: A new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, January 1994.

[2] V. Stavridou, A. Boothroyd, P. Bradley, B. Dutertre, L. Shackleton, and R. Smith. Formal methods and safety critical systems in practice. *High Integrity Systems*, 1(5):423–445, 1996.

[3] R.-J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, California Institute of Technology, 1992.

[4] 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.

[5] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In *Proc. Northern Formal Methods Workshop*. Springer-Verlag, 1996. To appear.

[6] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In Jifeng He, editor, *BCS-FACS Seventh Refinement Workshop*. Springer-Verlag, 1996.

[7] K. Lermer and C. J. Fidge. A methodology for compilation of high-integrity real-time programs. Technical Report 96-18, Software Verification Research Centre, University of Queensland, December 1996.

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

[9] C. J. Fidge. Modelling real-time multi-tasking systems with timed traces. In *Proc. Third Australasian Conference on Parallel and Real-Time Systems*, pages 94–100, Brisbane, September 1996.

[10] C. J. Fidge. Adding real time to formal program development. In M. Naftalin, 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.

[11] K. Kenny and K.-J. Lin. Measuring and analyzing real-time performance. *IEEE Software*, 8(5):41–49, September 1991.

[12] R. Chapman, A. Burns, and A. J. 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.

[13] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing real-time tasks with predictable timing. *IEEE Software*, 9(5):35–44, September 1992.

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

[15] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

Action systems

Action systems [3] are one of several similar formalisms for modelling concurrent behaviour. They use an interleaving model to conservatively build on the experience gained with sequential specification and refinement methods.

An action system is represented as an initialised loop:

$$\begin{array}{l} \{I\}; \\ \textbf{do} \\ \quad ([\!]\, i \bullet A_i) \\ \textbf{od} \end{array}$$

The initial state definition I is followed by one or more actions A_i. Each such action is a guarded command:

$$G \rightarrow B$$

The guard G defines the environmental 'assumptions' under which the action may occur and the body B defines its 'effect'. 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, one per iteration.

Usually such a loop describes a sequential system, but it can also model interleaved concurrent behaviour because no ordering is imposed on independent actions. A particular advantage of the approach is that the same system description can map to a number of different implementations, such as a 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.

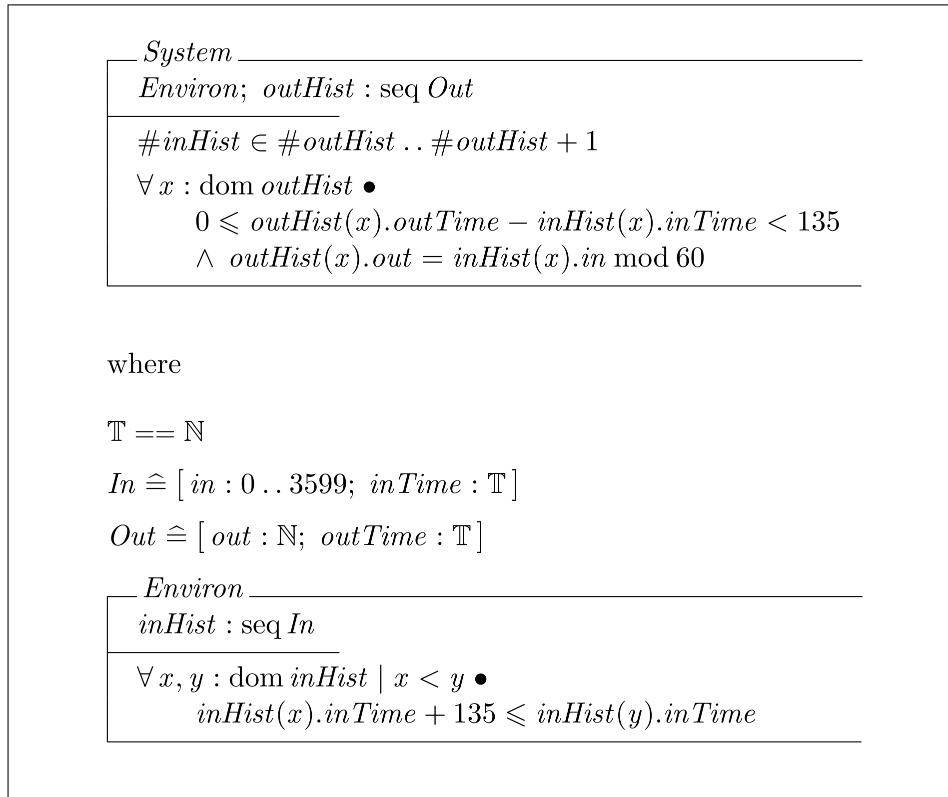

Figure 1: Real-time specification.

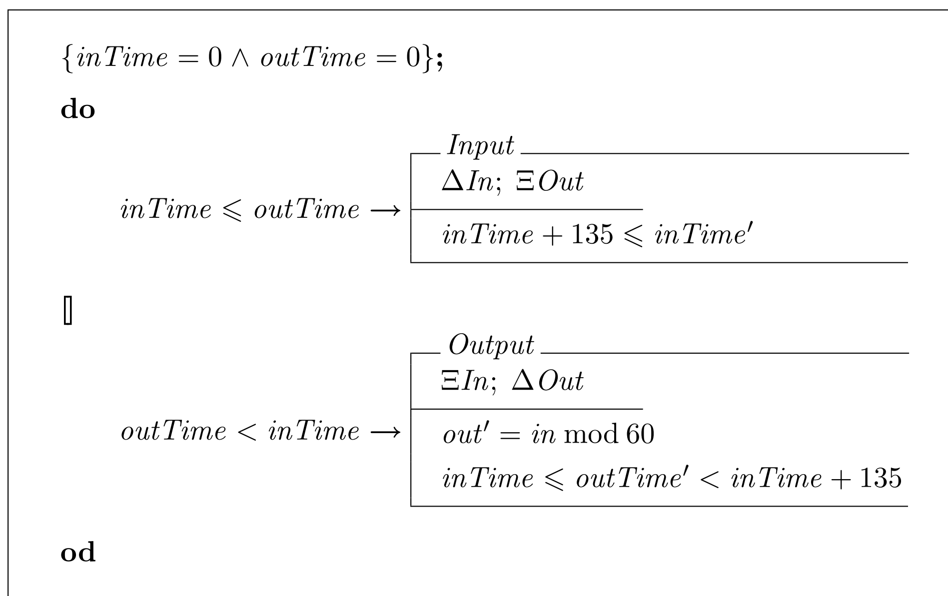

Figure 2: Development of *System* as two actions.

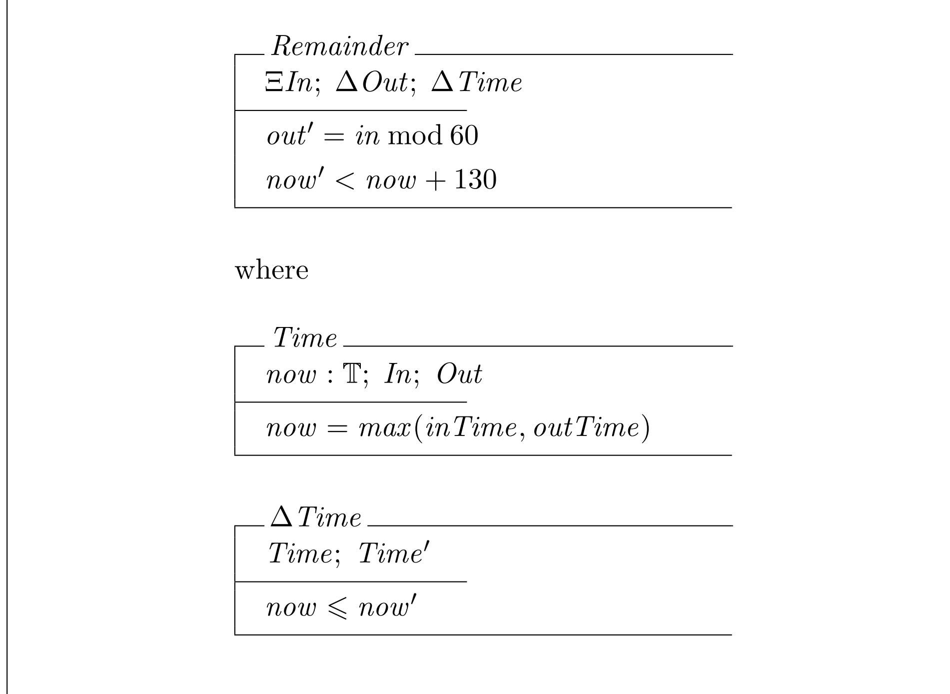

Figure 3: Development of *Output* as a specification with duration.

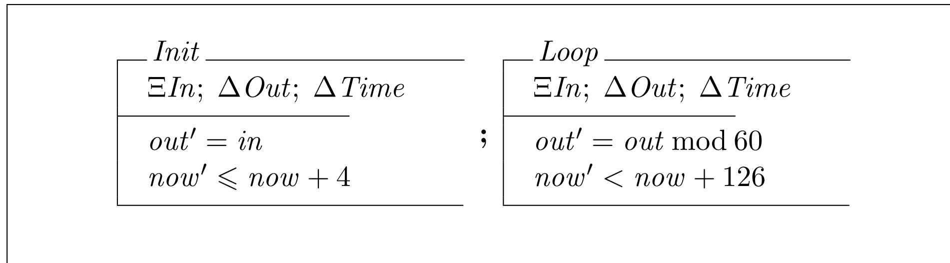

Figure 4: Development of *Remainder* by introducing sequential composition.

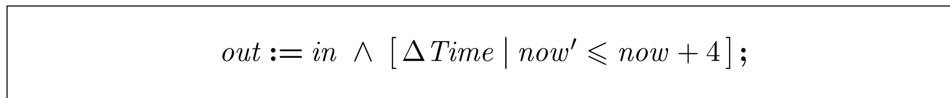

Figure 5: High-level language description of *Init*.

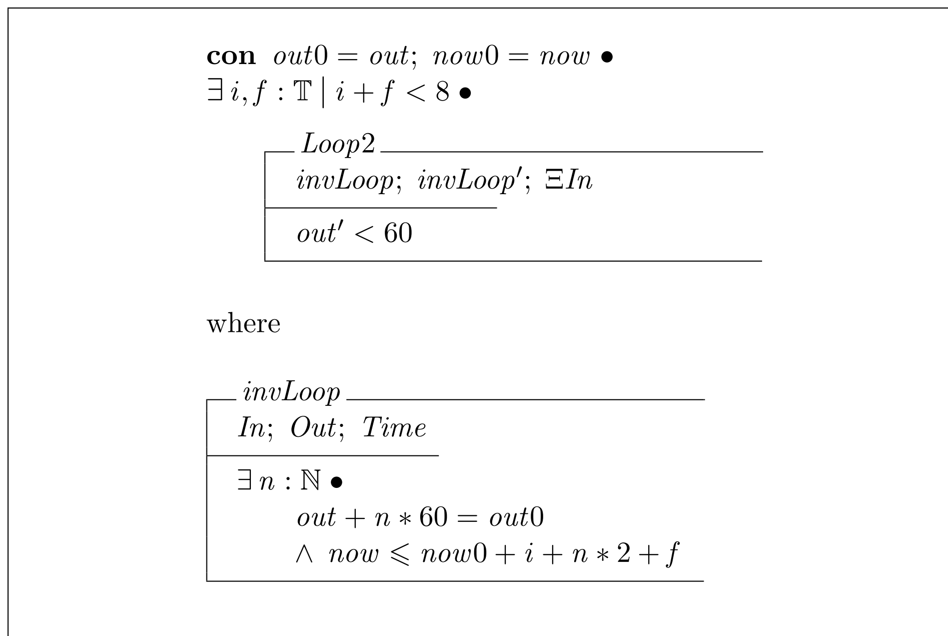

Figure 6: Development of *Loop* using a real-time invariant.

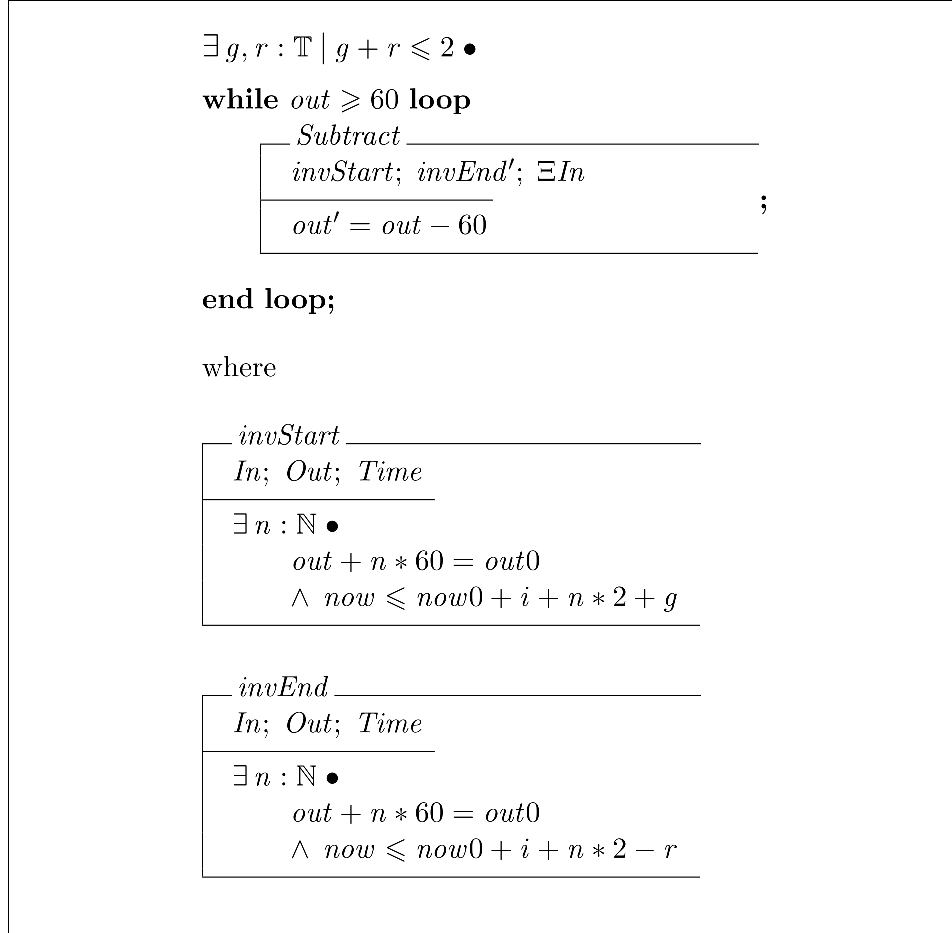

Figure 7: Development of *Loop*2 by introducing a while loop.

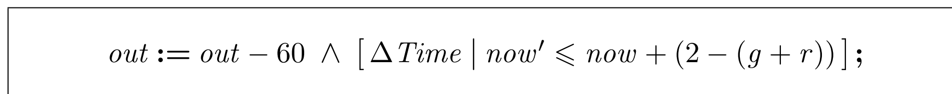

Figure 8: High-level language description of *Subtract*.

```
1   out := in
2   ∧ [ ΔTime | now′ ⩽ now + 4 ] ;
3   con  out0 = out; now0 = now •
4   ∃ i, f, g, r : 𝕋 | i + f < 8 ∧ g + r ⩽ 2 •
5       while out ⩾ 60 loop
6           out := out − 60
7           ∧ [ ΔTime; invStart; invEnd′ |
8               now′ ⩽ now + (2 − (g + r)) ] ;
9       end loop
10      ∧ [ ΔTime; invLoop; invLoop′ ] ;
```

Figure 9: High-level language program and timing constraints.

Colin Fidge

Colin Fidge is a Senior Research Fellow with the Software Verification Research Centre, University of Queensland. He is coordinator of several real-time systems research projects, with activities covering formal specification, development, compilation and analysis of concurrent real-time software. He was awarded a Ph.D. in computer science by the Australian National University in 1990.

Peter Kearney

Following undergraduate studies in mathematics, Peter Kearney did a Master's degree in computer science at the University of New England and a Ph.D. in computer science at the University of Queensland. His research interests include the semantics of concurrency, verification of real-time systems, and integrated formal development methods and systems. He is currently a Senior Research Fellow at the Software Verification Research Centre, University of Queensland.

Mark Utting

Mark Utting is a Lecturer with the Department of Computer Science, University of Waikato. He completed a Ph.D. on modular refinement in object-oriented languages at the University of New South Wales. He has worked in industry as an analyst/programmer for several years and spent four years at the Software Verification Research Centre, working on the Quartz project and coordinating the Ergo proof tool project.