Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Formal Change Impact Analyses for Emulated Control Software

C. J. Fidge

School of Information Technology and Electrical Engineering, The University of Queensland

Abstract. Processor emulators are a software tool for allowing legacy computer programs to be executed on a modern processor. In the past emulators have been used in trivial applications such as maintenance of video games. Now, however, processor emulation is being applied to safety-critical control systems, including military avionics. These applications demand utmost guarantees of correctness, but no verification techniques exist for proving that an emulated system preserves the original system's functional and timing properties. Here we show how this can be done by combining concepts previously used for reasoning about real-time program compilation, coupled with an understanding of the new and old software architectures. In particular, we show how both the old and new systems can be given a common semantics, thus allowing their behaviours to be compared directly.

Key words: Software maintenance – Program analysis

1 Introduction

Processor emulators are a valuable software tool for computer system maintenance. They allow a machine-code program written in an obsolete instruction set to be executed on a modern processor. In effect, a processor emulator interprets an old program on a new machine. To date, processor emulation has enjoyed considerable success in trivial applications such as maintenance of video games [18].

Now, however, the technology is being trialled for safety and mission-critical applications such as military avionics [9]. Whereas aircraft remain in service for decades, the embedded microprocessors they contain have a lifecycle measured in mere years. Processor obsolescence has thus become a major technical [8] and economic [7] problem for long-lived control systems. Maintaining computer processors that are no longer mass produced is prohibitively expensive, as is rewriting legacy software for a new machine, so processor emulation is seen as an attractive potential solution [9].

However, for processor emulation to be acceptable in situations where human life and national security are at stake, its use demands the strongest possible guarantees of correctness. Avionics software development is governed by internationally-recognised standards such as DO-178B [26], which is itself supplemented by recommended processes for avionics software maintenance [35, 36]. These stress the importance of performing *change impact analyses* to assess the potential effects of software upgrades on the functional and timing characteristics of an existing control system.

Unfortunately, standards tend to lag behind technological developments, and the avionics standards do not yet describe processes for analysing emulated control software. Therefore, this paper aims to show how real-time program proof concepts [15, 16, 28], especially those previously intended for reasoning about compiler correctness [19, 25], can be used to verify that an emulated Operational Flight Program provides a behaviour 'equivalent' to that of the legacy code. This is a challenging problem because processor emulation involves execution of both legacy machine code and new high-level language software patches, and because embedded control software interacts directly with its hardware environment.

In previous work [13] we performed a partial-correctness analysis, based on the software's underlying weakest precondition semantics. By contrast, this paper performs a much clearer and simpler total-correctness proof at the programming statement level, by modelling language primitives as relational assignments.

2 Legacy Mission Computer System Software

As a running example, this section introduces a fragment of a typical Operational Flight Program, as would be found in a

Correspondence to: Dr C. Fidge, ITEE, University of Queensland, Australia 4072. E-mail: cjf@itee.uq.edu.au. Fax: +61 7 3365 1533.

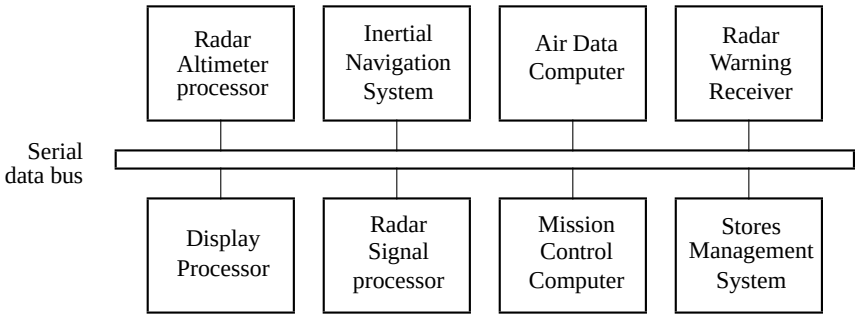

Fig. 1. Typical military avionics control system architecture

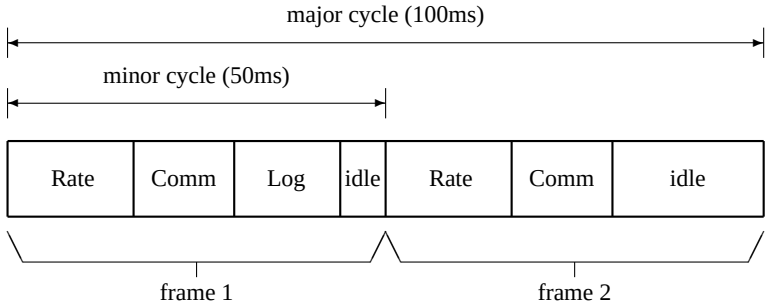

Fig. 2. Schedule for the example Operational Flight Program

military avionics system. Such programs are notoriously hard to maintain because

- they are embedded within a larger system, and thus interact directly with hardware devices,
- they must respond to multiple inputs and control multiple outputs, so they require concurrency,
- they must react to external events in a timely manner, so they are subject to rigid real-time constraints, and
- they have limited computing resources at their disposal, so there is usually no clear separation between 'system' and 'application' code [21].

A typical avionics *Mission Computer System* consists of several Remote Terminals connected via a data bus, as shown in Fig. 1 [22]. Each *Remote Terminal* is a processor board dedicated to a particular aircraft function, and is normally connected directly to one or more peripheral devices. For instance, the *Radar Altimeter* processor samples altitude readings from the corresponding sensor, whereas the *Display Processor* sends information to cockpit instruments. The *Mission Control Computer* manages overall system functions and controls bus access.

The *Operational Flight Program* executing on each Remote Terminal will usually have a 'cyclic executive' architecture [3]. This consists of several *tasks*, which perform the necessary computations, and a *Mission Computer Executive*, which controls allocation of computing resources to tasks [11]. For instance, we assume here that there are three tasks to be performed on the Radar Altimeter processor.

- Rate: Samples the current altitude and calculates the rate of ascent
- Comm: Sends the current altitude and ascent rate to the Display Processor
- Log: Writes altitude data to the flight recorder

The Mission Computer Executive is then responsible for invoking these tasks in a timely manner. A typical 'timer-driven' executive [17] consists of a loop which at each iteration:

1. waits for a periodic timer interrupt to occur;
2. increments a *frame* counter; and
3. invokes one or more tasks, depending on the frame number.

The frequency of timer interrupts therefore governs the rate at which tasks can be performed. For the purposes of our example we assume that timer interrupts occur with a frequency of 20 hertz on the legacy Radar Altimeter processor, which means that each frame has a duration of 50 milliseconds.

The choice of which tasks to invoke in a given frame is made by following a predetermined task *schedule*. Fig. 2 shows a possible schedule for the three tasks on the Radar Altimeter processor. The critical Rate and Comm tasks are executed every 50 milliseconds, but the less important Log task is executed only every 100 milliseconds. (Non-critical diagnostic tasks may occupy the idle time at the end of frames.) At the end of each major cycle the pattern repeats indefinitely.

The tasks themselves typically read inputs, process data, and send outputs. For instance, the legacy code for the Rate task is shown in Fig. 3. For readability we use assembly notation throughout this paper, although the legacy program is likely to exist as undocumented machine code only. As an illustrative assembly language we use a two-address instruc-

Label	Op.	Args.	Comment	
	LOAD	A, 600_{dir}	get previous altitude	(1)
	LOAD	C, 1_{imm}	load constant 1	(2)
READALT	OUT	C, P	start ADC conversion	(3)
	LOAD	B, 3_{imm}	initialise busy-wait counter	(4)
ADCDELAY	SUB	B, 1_{imm}	decrement busy-wait counter	(5)
	BRGT	ADCDELAY	iterate while counter exceeds zero	(6)
	IN	B, Q	read new altitude (in feet) from ADC	(7)
SAVEALT	STORE	B, 600_{dir}	save new altitude	(8)
	SUB	B, A	compute change in altitude (in feet)	(9)
	MULT	B, 20_{imm}	convert difference to feet per second	(10)
SAVERATE	STORE	B, 601_{dir}	store calculated ascent rate	(11)

Fig. 3. Legacy assembly code for the Rate task

tion set for a simple processor with eight general-purpose registers, A to H, and a comparison register which holds the results of explicit CMPR instructions, or the difference between the last register update and zero [14, Ch. 7]. Direct and immediate addressing modes are indicated by subscripts. We also assume the existence of a set of memory-mapped input/output ports, P to W, to interface with hardware devices.

The Rate task begins by fetching the altitude saved during its previous invocation from memory location 600. Instructions 2 to 7 then read the altitude from the radar altimeter through an Analog-to-Digital Converter. The task writes '1' to the ADC's control register, via output port P, to start the conversion. It must then wait at least 30 microseconds for the conversion to be completed. Assuming an instruction execution time of 5 microseconds, the task does this by busy-waiting at label ADCDELAY (consuming 7 instruction cycles, including the initial LOAD). Instruction 7 then reads the new altitude from the ADC's data register, via input port Q, and instruction 8 saves it for the next invocation.

Instructions 9 and 10 then use the previous and current altitude readings to calculate the ascent rate. The constant in the MULT instruction reflects the fact that the executive invokes this task at a frequency of 20 hertz. Finally, instruction 11 saves the result in memory location 601. (When invoked next, the Comm and Log tasks can thus access the current altitude and ascent rate from locations 600 and 601, respectively.)

3 Processor Emulation

After a Mission Computer System such as that described above has been in service for several years it becomes increasingly difficult and expensive to maintain [7]. There is thus a strong incentive to replace obsolete processors in the Remote Terminals with modern equivalents [8], but an old Operational Flight Program will not execute on a new processor with a different instruction set.

Processor emulation offers a potential solution by introducing an emulator program to interpret the legacy code on a new processor [9]. As shown in Fig. 4, a typical legacy control system consists of machine code executing directly on the old processor and interacting directly with input/output devices. Introducing a processor emulator changes the architecture as shown in Fig. 5. The old program is now interpreted by the processor emulation software, which itself runs on a standard Real-Time Operating System (RTOS). Processor emulators are available as commercial, off-the-shelf products [23], for specific legacy instruction sets.

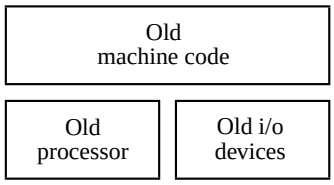

Fig. 4. Components of a legacy control system

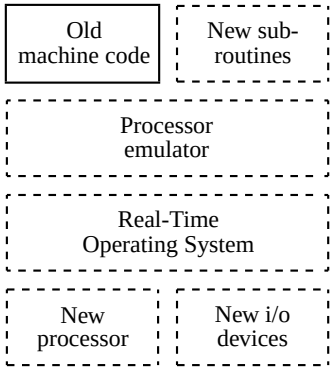

Fig. 5. Components of an emulated control system

However, despite the desire to reuse the legacy code without change, some adjustments are inevitable if an embedded control program is to work correctly in a new hardware environment. To support this, emulators provide an Application Programming Interface (API) which allows programmers to associate software patches with particular instruction memory locations or input/output ports [34]. When the emulated

```
void ReadAlt()                  // patch linked to label READALT
{ int Altitude, Busy;                                                  (12)
  WriteIOPort(T, 1);            // start ADC conversion                (13)
  do { Busy = ReadIOPort(V);    // wait while ADC busy                 (14)
     } while Busy == 1;                                                (15)
  Altitude = ReadIOPort(U);     // read ADC data (in metres)           (16)
  Altitude = Altitude * 3.28;   // convert metres to feet              (17)
  WriteReg(B, Altitude);        // put altitude (in feet) in register B (18)
  UpdateIC(SAVEALT);            // bypass legacy ADC code              (19)
  return;                                                              (20)
}
```

Fig. 6. New high-level language subroutine to read altimeter data

code reaches such a point, the emulator transfers control to a corresponding subroutine written in the emulator's native high-level programming language. This subroutine may use API operations provided by the emulation package to access the new hardware architecture, or modify the legacy processor's (emulated) state.

In our example, for instance, we will assume that the outmoded altimeter and Analog-to-Digital Converter are replaced as part of the hardware upgrade. This invalidates instructions 3 to 7 in Fig. 3 because this code is specific to the original altimeter and ADC. We assume instead that the new ADC provides a 'busy' signal (with a known worst-case delay), as a more robust alternative to busy-waiting [33], and that the new altimeter is calibrated in metres, rather than feet.

To account for this change, the parameterless C++ subroutine in Fig. 6 uses emulator API operations to interface with the new altimeter and ADC. It is called whenever the emulated code reaches label READALT. Statement 13 starts the data conversion by writing to the new ADC's control register at output port T. Statements 14 and 15 then wait while the ADC's busy signal at input port V equals 1 [20, p. 400]. Statement 16 then reads the result from the ADC's data register via input port U. Since the new altitude is in metres, rather than feet as expected by the legacy code, this value is converted to the appropriate units by statement 17. Statement 18 puts the altitude in (emulated) register B. Statement 19 then updates the (emulated) processor's instruction counter, so that control will bypass the legacy instructions for accessing the ADC, and will go to label SAVEALT rather than back to READALT, when the subroutine returns.

The need to introduce the software patch above was obvious. More worrying are necessary changes that may be overlooked when installing an emulator. For instance, assume now that the new processor generates timer interrupts at a frequency of 25 hertz, rather than the legacy processor's 20 hertz. Because emulators can interpret instructions much faster than they could be executed on the old processor [23], the legacy program may still execute 'correctly' in this situation, in the sense that each task still completes all of its necessary computations within each frame. Indeed, the emulated system could be extensively tested without any problems being detected.

However, consider the purpose of instruction 10 in Fig. 3. The legacy system's programmer used constant '20' in this multiplication on the assumption that the Rate task is invoked 20 times per second—any change to this frequency also requires the corresponding arithmetic to be changed. If this is not done in the situation described above, the calculated ascent rate will be inaccurate by 25%. Thus legacy code which is time-sensitive, but not recognised as such, presents a significant danger. (To avoid this risk some emulation packages provide a 'speed matching' option [23], even though its use wastes available processor time.)

In this case a solution is to add another patch, associated with label SAVERATE, as shown in Fig. 7. (New subroutines are executed *before* the instruction at the corresponding label [34], so the patch is called between instructions 10 and 11.) The new code simply increases the calculated ascent rate by 25% to correct the distortion caused by the changed frequency.

With the addition of these subroutines the upgraded Mission Computer System's programmer aims to achieve a behaviour equivalent to that of the original Rate task. The challenge now is to convincingly verify that no unintended changes in behaviour have been introduced and that all necessary software patches have been made.

4 Approach to Verifying Emulated Control Software

Our overall goal, therefore, is to formally prove that the emulated legacy code provides the 'same' functional and timing behaviour as it did when executed on the original processor. However, we already know that this will *not* be the case. The emulator will interpret instructions faster than the original processor [23], and the emulated program contains software patches to accommodate hardware changes. Thus the 'equivalence' to be proven is an approximate one that must accept harmless differences.

(Ideally, it should be possible to show that both the legacy and emulated systems satisfy the original system's require-

```
void SaveRate()                    // patch linked to label SAVERATE
{ int AscentRate;                                                    (21)
  ReadReg(B, AscentRate);          // get skewed ascent rate         (22)
  AscentRate = AscentRate * 1.25;  // compensate for higher frame rate (23)
  WriteReg(B, AscentRate);         // put corrected ascent rate      (24)
  return;                                                            (25)
}
```

Fig. 7. New high-level language subroutine to correct saved ascent rate

ments specification. However, adequate requirements documentation is unlikely to exist for a legacy control system, especially if its behaviour was calibrated experimentally, or if it was modified substantially during its operational lifetime.)

Furthermore, fully verifying the equivalence of the two systems is impossible with contemporary proof technology. Fig. 5 shows that the new system interposes both a proprietary processor emulator and a commercial Real-Time Operating System between the legacy code and the new processor. Verifying the correctness of these large-scale software components is well beyond the capabilities of current formalisms. As a compromise, we therefore *assume* that the manufacturer of the emulation package can provide evidence that the emulator has passed the original test suite for the legacy processor's instruction set, as is usually required in safety-critical applications [23]. We also *assume* that the underlying RTOS is one intended for safety-critical applications and that its manufacturer can thus provide evidence that its development met relevant standards [4]. Although neither of these assumptions mean that the software has been formally verified correct, they give us sufficient confidence in the integrity of these system components that we need not attempt to verify them ourselves, and can instead concentrate on comparing the two versions of the Operational Flight Program.

(Interestingly, we may also usually assume that the legacy software has a 'clean' in-service history—any serious problems with it will already have been detected and eliminated during its operational lifetime—so we do not need to reason about the legacy code's 'correctness,' merely our ability to emulate its behaviour.)

Therefore, assuming that the emulator interprets legacy instructions correctly, our specific goal is to prove an (approximate) equivalence between the legacy assembly code (Fig. 3), and the legacy code augmented with hardware-specific patches. Even this presents a challenge because the emulated program interleaves legacy assembly-level instructions and high-level language statements, as shown in Fig. 8. The closest analogy to this situation occurs in formalisms for modelling program compilation [27, 32] or decompilation [6], where relationships between high and assembly-level programs are proven. Our model below therefore adapts various ideas from these formalisms, especially those used for reasoning about compilation of real-time programs [19, 25] since we need the ability to analyse time-sensitive code.

```
LOAD A, 600_dir ;
LOAD C, 1_imm ;
call ;
{ int Altitude, Busy;
      ⋮ (see Fig. 6)
  return;
} ;
STORE B, 600_dir ;
SUB B, A ;
MULT B, 20_imm ;
call ;
{ int AscentRate;
      ⋮ (see Fig. 7)
  return;
} ;
STORE B, 601_dir
```

Fig. 8. Behaviour of the emulated Rate task

5 Modelling Legacy and Emulated code

To prove any relationship between two programs we must first give them a common semantic basis. This section introduces a set of primitives for modelling program fragments, and uses them to give a semantics to the legacy instructions and emulator operations in our case study.

5.1 Modelling Language Primitives

The basic modelling language contains the following well-known primitives [10, 1]. (Formally, they can all be given a weakest-precondition semantics as described in Appendix A, but it is not necessary to understand this underlying semantics for the proof below.)

– An *assumption*, $[P]$, states that property P is expected to hold at this point in the program [1, p. 189]. Let P be a predicate on the state variables currently in scope. Intuitively, an assumption is used to document conditions that are expected to have been established by the program's environment [1, p. 5].

– An *assertion*, $\{P\}$, similarly states that predicate P is expected to hold at the point where it appears [1, p. 189]. In this case however, the condition is one which the program itself is obliged to make true [1, p. 4].
– A *relational assignment*, $[v := v' \mid G]$, models assignment of a value v', defined by predicate G, to variable v [2, §4.7]. Predicate G may refer to state variables and the primed value 'v'' which denotes variable v's value when the assignment terminates.
– *Sequential composition*, '$S_1 ; S_2$', of two statements models execution of statement S_1 followed by execution of statement S_2 in the usual way [10, p. 137].
– A local *variable declaration*, '**var** $v : T \bullet S$', adds a fresh variable named 'v' of type T to statement S's state space [1, p. 227].
– A (demonic) *nondeterministic choice*, '$S_1 \sqcap S_2$' denotes an arbitrary choice between statements S_1 and S_2 [1, p. 189].
– A *recursive statement*, '**rec** $X \bullet S(X)$', denotes statement $S(X)$ with a copy of itself substituted whenever program-valued variable X is encountered [1, Ch. 20]. Here $S(X)$ is a compound statement in which 'X' may appear where a statement is normally expected. (Formally, the recursive statement denotes the least fixed point in the weakest-precondition ordering of programs.)

Each of these primitives serves a distinct purpose. Assertions and assumptions provide ways of interrogating the state variables. Assignments allow the state to be updated. Sequential composition allows statements to be constructed from substatements. Variable declarations allow the state space to be extended. Choice coupled with recursion allows modelling of iteration [1, p. 15].

In particular, the relational assignment supports especially concise models of program fragments. We allow it to be generalised to multiple simultaneous assignments, e.g., $[v, w := v', w' \mid G]$, in the usual way [1, p. 156]. As a shorthand we also allow the predicate part to be omitted when it is trivial. Let E be an expression on (unprimed) state variables which is type-compatible with variable v.

$$[v := E] \stackrel{\text{def}}{=} [v := v' \mid v' = E]$$

We also allow both of these representations to be freely mixed in multiple assignments.

5.2 Legacy Instruction Semantics

Using this modelling language, Table 1 defines the meaning of assembly instructions as executed on the legacy processor. This is done by describing the effect of each instruction on the legacy processor's state, which consists here of general-purpose registers, the special comparison register, and data memory. As in previous real-time formalisms [15,16], we also explicitly model the current time by a special variable, τ.

In Table 1 let S be a basic block of instructions in our assembly language that does not contain branch instructions or labels that are the targets of branch instructions; r be a register; ℓ be a label; z be an integer (representable on the legacy processor); c be the comparison register; m be the data memory array; a be a data memory address; Y be a device-dependent external input value; and τ be the current time.

Table 1. Semantics of legacy assembly code instructions

Instruction or basic block S	Equivalent modelling language statement
`LOAD` r, z_{imm}	$[r, c, \tau := z, z, \tau + 5\mu\text{s}]$
`LOAD` r, a_{dir}	$[r, c, \tau := m(a), m(a), \tau + 10\mu\text{s}]$
`STORE` r, a_{dir}	$[m, \tau := m \oplus \{a \mapsto r\}, \tau + 10\mu\text{s}]$
`SUB` r_1, r_2	$[r_1, c, \tau := r_1 - r_2, r_1 - r_2, \tau + 5\mu\text{s}]$
`SUB` r, z_{imm}	$[r, c, \tau := r - z, r - z, \tau + 5\mu\text{s}]$
`MULT` r, z_{imm}	$[r, c, \tau := r * z, r * z, \tau + 10\mu\text{s}]$
'S_1 S_2'	$S_1 ; S_2$
'ℓ S `BRGT` ℓ'	**rec** $X \bullet (S ; [\tau := \tau + 5\mu\text{s}] ; (([c > 0] ; X) \sqcap [c \leqslant 0]))$
`OUT` r, P	$[\mathsf{P}^v, \mathsf{P}^t, \tau := r, \mathsf{P}^{t'}, \tau + 5\mu\text{s} \mid \tau < \mathsf{P}^{t'} \leqslant \tau']$
`IN` r, Q	$\{\mathsf{P}^v = 1 \wedge \tau - \mathsf{P}^t \geqslant 30\mu\text{s}\} ; [r, c, \tau := Y, Y, \tau + 5\mu\text{s}]$

The first group in Table 1 defines the effect of simple instructions, each of which can be modelled as a multiple assignment. For instance, the '`LOAD`' instruction with an immediate operand z changes three variables, r, c and τ. It sets both registers r and c equal to integer z, and adds 5 microseconds to current time τ, to account for the instruction's execution time. The other instructions follow similarly. Instructions that access memory or perform complex arithmetic are assumed to take 10 microseconds. Data memory m is represented as a function from addresses to values. In the '`STORE`' definition, functional overriding $m \oplus \{a \mapsto r\}$ denotes function m with domain element a mapped to value r [31, p. 128].

The next two groups in Table 1 define the behaviour of two commonly-occurring patterns of instructions. The first simply reminds us that vertically displayed blocks of assembly instructions are interpreted as being sequentially composed (provided that the sequence does not contain branches, or labels that are the targets of branches). By giving a meaning to particular control-flow patterns in this way, we avoid the need to explicitly model the legacy processor's instruction counter [12].

The next pattern consists of a basic block S, labelled by ℓ, which is followed by a conditional branch '`BRGT`' to ℓ. In effect, this is a loop which performs one or more instances of block S, terminating when the comparison register c is not positive. Modelling such patterns as a unit avoids the challenging problem of defining a separate semantics for branch instructions. (In the past this has been done by introducing an explicit interpreter of instruction sequences [27,25], or by adding **goto** statements to the modelling language [32,5].)

The semantics on the right uses the recursion primitive to model iteration. Basic block S is followed by an assignment that adds 5 microseconds to the current time, to model the overhead of performing the '`BRGT`' instruction at the end of each iteration. This is followed by a choice between two statements containing complementary assumptions (which makes the choice deterministic). If comparison register c is positive the left-hand alternative is followed, and the whole '**rec**' statement recurs. If c is not positive then the recursive statement terminates. Similar definitions can be given for other branching or iterative patterns of instructions [12].

The final group in Table 1 defines the meaning of input/output instructions for two specific memory-mapped i/o ports. Since the properties of external devices are essential for reasoning about the behaviour of an embedded program, we model important device characteristics in the semantics of the instructions that access the devices. Inspired by the Temporal Agent Model [28], each output port `X` is modelled by two variables, X^v, which denotes the value currently stored in the port's location, and X^t, which is a timestamp holding the time at which this value last changed. The latter variable is an auxiliary one—it cannot be accessed by the executable code, but is used to reason about time-sensitive behaviours.

Thus, instruction '`OUT` r, `P`' in Table 1 writes the value in register r to the ADC's control register, and takes 5 microseconds to do so. Timestamp P^t tells us that the register will be updated after the instruction's starting time τ, but no later than its finishing time τ' (i.e., $\tau + 5\mu$s), although we do not know here exactly when in this interval the update will be completed. This shows how relational assignments can be used to model nondeterministic behaviour.

Finally, the '`IN` r, `Q`' instruction reads from the ADC's data register and is modelled by two primitives. The initial assertion accounts for the time required for the ADC to perform the conversion. It obliges the programmer to ensure that value '1' was written to control register `P` at least 30 microseconds ago. If so, the following assignment sets register r (and c) equal to some value Y produced by the external environment. In our case study this value represents an altitude reading produced by the legacy system's radar altimeter. Since external inputs are beyond the control of the Operational Flight Program, we cannot predict what values they denote and must model them symbolically.

5.3 Emulated Statement Semantics

To define the meaning of emulated code we need to define a semantics for all the operations performed by the processor emulator, including emulated legacy instructions, Application Programming Interface subroutines, and actions of the emulator itself. Since the emulator maintains its own representation of the legacy processor's state, these operations can be defined with respect to this state, together with locally-scoped high-level language variables.

We have assumed that the emulator interprets instructions correctly, so emulated instructions will have the same functional behaviour as they did on the original processor.

Table 2. Semantics of emulator operations and statements

Operation or statement S	Equivalent modelling language statement
v = E	$[v, \tau := E, \tau + D^E \mu\text{s}]$
{ **int** v; S; }	**var** $v : \mathbb{Z} \bullet ([\tau := \tau + D^{\mathbb{Z}} \mu\text{s}] \,;\, S \,;\, [\tau := \tau + D^{\mathbb{Z}} \mu\text{s}])$
S_1; S_2	$S_1 \,;\, S_2$
do { S } **while** B	**rec** $X \bullet (S \,;\, [\tau := \tau + D^B \mu\text{s}] \,;\, (([B] \,;\, X) \sqcap [\neg B]))$
call	$[\tau := \tau + 4\mu\text{s}]$
return	$[\tau := \tau + 4\mu\text{s}]$
ReadReg(r, v)	$[v, \tau := r, \tau + 1\mu\text{s}]$
WriteReg(r, v)	$[r, c, \tau := v, v, \tau + 1\mu\text{s}]$
UpdateIC(ℓ)	$[\tau := \tau + 1\mu\text{s}]$
WriteIOPort(`T`, z)	$[\mathsf{T}^v, \mathsf{T}^t, \tau := z, \mathsf{T}^{t'}, \tau + 2\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau']$
v = ReadIOPort(`V`)	$\{\mathsf{T}^v = 1\} \,;$ $[v, \tau := v', \tau + 2\mu\text{s} \mid ((\tau - \mathsf{T}^t < 10\mu\text{s}) \Rightarrow v' = 1) \wedge ((\tau - \mathsf{T}^t \geqslant 10\mu\text{s}) \Rightarrow v' = 0)]$
v = ReadIOPort(`U`)	$\{\mathsf{T}^v = 1 \wedge \tau - \mathsf{T}^t \geqslant 10\mu\text{s}\} \,;$ $[v, \tau := Z, \tau + 2\mu\text{s}]$

The only difference is that emulated instructions will execute faster [23]. Thus the emulated semantics for simple assembly instructions is the same as that shown in the first group in Table 1, except that we assume all 5 microsecond execution times are replaced by 2 microseconds, and all 10 microsecond execution times are replaced by 4 microseconds.

The semantics for other emulator operations is shown in Table 2. Let S be a (compound) statement in the emulator's API language (C++ here); v be a high-level language variable; ℓ be an assembly language label; r be a legacy processor register; z be an integer; E be a high-level language expression; B be a boolean-valued expression; D^F be a non-negative duration in microseconds whose magnitude depends on language construct F's structure; Z be a device-dependent external input value; and τ be the current time.

The first group in Table 2 consists of C++ statements for assignment, declaring an integer, sequential composition and **do-while** iteration, as used in API subroutines. Their semantics follows that of other real-time formalisms [15]. The assignment statement's execution time is represented symbolically by a duration D^E whose value depends on the structure of expression E. Ways of predicting execution times for high-level language statements and expressions, based on their syntactic structure, have been well explored in the real-time literature [30]. (However, this approach produces different results depending on which compiler code generation strategies and optimisations are anticipated. Therefore, such an analysis may produce a range of possible durations for

D^E, from the best to the worst-case execution time. In this situation the equivalence proof below should be checked for each such value.)

Similarly in the **do-while** semantics, duration D^B denotes the time required to evaluate expression B and branch accordingly. Otherwise the definition is similar to that for the BRGT loop in Table 1.

The high-level language variable block includes a timing overhead before and after the enclosed statement to respectively allocate and deallocate space for the newly-declared variable. In Table 2 let $\mathbb{Z}$ denote the integer type, and $D^{\mathbb{Z}}$ be the time required to (de)allocate stack space for a variable of this type.

The '**call**' and '**return**' operations in Table 2 represent the actions taken by the emulator to respectively transfer control to and from a parameterless Application Programming Interface subroutine. These operations do not change the emulated legacy processor's state, so their only impact on our model is the 4 microsecond delay they introduce.

The third group in Table 2 includes operations provided by the emulator's Application Programming Interface for modifying the (emulated) legacy processor's state. The first two allow the value of a high-level language variable v to be read from and written to a legacy processor register r, respectively. The 'UpdateIC' operation updates the (emulated) instruction counter. However, since we have avoided modelling the legacy instruction counter explicitly [12], the only effect of the operation here is a 1 microsecond delay.

The final group of emulator operations in Table 2 are specific to the new Analog-to-Digital Converter. The first writes an integer to the ADC's control register at output port T. The operation as modelled simply stores the value, and takes 2 microseconds to do so. In reality, however, this operation also affects the hardware device itself, causing it to perform an action whose outcome is observable later. In our model, the effects of this action are described in the next two operations in the table, which read from the device.

The first reads the ADC's busy signal, from input port V. The initial assertion obliges us to ensure that the control register at port T has previously been assigned value '1'. If so, then the statement's outcome depends on how much time has elapsed since port T was updated. Within 10 microseconds of this time the busy signal reads '1'. After this time it returns '0', indicating that it is now safe to read from the data register. The second operation reads from the ADC's data register at input port U. The assertion says that we are obliged to ensure that at least 10 microseconds have elapsed since '1' was written to the control register at port T. If so, the operation reads an application-specific external input value Z. In our case study Z denotes an altitude produced by the new radar altimeter.

6 Equivalence of the Legacy and Emulated Tasks

Having given a meaning to basic legacy instructions and emulator operations above, we now want to apply these semantics to characterise the behaviour of the whole legacy and emulated tasks. This section presents a number of algebraic laws for calculating the semantics of program fragments, uses them to determine the semantics of the two tasks, and then compares the results.

6.1 Reasoning Laws

The semantics above makes extensive use of (multiple, demonically-nondeterministic) relational assignments to model atomic actions. In our example programs these actions are composed sequentially, or in loops and declarative blocks. Therefore, to calculate the semantics of the tasks, we need laws for combining relational assignments via these constructors.

The following laws are the main ones used in the proof below. They are based on well-known principles from the 'program refinement' literature [29,24] but we have reexpressed them in relational assignment [2] form. (They can all be proven using the semantics in Appendix A.) For clarity they are shown for single-variable assignments only, but the principles extend readily to multiple assignments.

The first law states that an assumption before a relational assignment can be absorbed into the assignment. Here P is a predicate on the state space and G is a predicate on the state space that can additionally refer to final value v'.

Law 1 (Preceding assumption)

$$[P] \,;\, [v := v' \mid G] = [v := v' \mid P \wedge G]$$

In a complementary situation, an assertion following an assignment can be eliminated if we are certain that the assignment will establish the assertion [24, p. 66]. Let $P[t/v]$ be predicate P with all free occurrences of variable v replaced by term t [24, §A.2.1].

Law 2 (Succeeding assertion)

$$\begin{aligned} &[v := v' \mid G \wedge P[v'/v]] \,;\, \{P\} \\ = \; &[v := v' \mid G \wedge P[v'/v]] \end{aligned}$$

The next two laws show how consecutive assignments can be merged. The first applies to two assignments to the same variable. Let G and H be predicates on the state space which may refer to final value v'. Let '$\forall v \bullet P$' and '$\exists v \bullet P$' denote universal and existential quantification of variable v over predicate P, respectively.

Law 3 (Assignments to same variable)

$$\begin{aligned} &[v := v' \mid G] \,;\, [v := v' \mid H] \\ = \; &[v := v' \mid (\forall v'' \bullet G[v''/v'] \Rightarrow (\exists v' \bullet H[v''/v])) \wedge \\ &\qquad (\exists v'' \bullet G[v''/v'] \wedge H[v''/v])] \end{aligned}$$

On the right-hand side v'' represents an intermediate value of variable v, between the two assignments on the left. (Variables v and v' on the right denote v's value before and after the whole assignment in the usual way). There are two conjuncts in the right-hand statement's predicate part [29]. The

first ensures that given any value of v produced by the first statement on the left-hand side that the second statement on the left can perform a computation. Otherwise, the first statement on the left could leave v in a state from which the second statement could never succeed. (Although intimidating, this condition is satisfied trivially throughout the proof below because none of our atomic statements has significant preconditions for successful execution.) The second conjunct defines the behaviour of the assignment on the right-hand side to be the combined behaviour of the two assignments on the left, with predicate G's final value for variable v unified with H's initial value for v.

In the case of consecutive assignments to distinct variables v and w the law is essentially the same. Here H is a predicate on the state space in which variables w, w' and v' may appear free, but *not* v. This is because, on the left-hand side, the original value of v before the first assignment is not accessible by the second assignment. The 'v'' in H thus refers to variable v's value between the two assignments on the left, which is also its final value in the combined assignment on the right. (On the left-hand side $H[v/v']$ is thus a predicate in which only variable w may appear in primed form, as usual.)

Law 4 (Assignments to different variables)

$$\begin{aligned}&[v := v' \mid G] \,;\, [w := w' \mid H[v/v']]\\ =\ &[v, w := v', w' \mid (\forall v' \bullet G \Rightarrow (\exists w' \bullet H)) \wedge G \wedge H]\end{aligned}$$

Since our definition of iteration makes use of a nondeterministic choice operator, it is also convenient to have a law that shows how choices between relational assignments can be combined [29]. Let G and H be predicates on the state space in which v' may appear free.

Law 5 (Choice between assignments to same variable)

$$\begin{aligned}&[v := v' \mid G] \sqcap [v := v' \mid H]\\ =\ &[v := v' \mid (\exists v' \bullet G) \wedge (\exists v' \bullet H) \wedge (G \vee H)]\end{aligned}$$

The first two conjuncts in the predicate on the right-hand side ensure that both alternatives lead to executable computations. The final conjunct simply defines the effect of the combined statement to be that of either alternative [29].

Finally, the following law shows how an initialised variable block can be introduced. Its proviso avoids name clashes with existing variables. Let E be an expression of type T.

Law 6 (Introduce fresh variable)

$$\begin{aligned}&[v := v' \mid G]\\ =\ &\text{'provided } w \text{ does not appear free in } G\text{'}\\ &\mathbf{var}\ w : T \bullet ([w := E] \,;\, [v := v' \mid G])\end{aligned}$$

On the right the second assignment makes no use of variable w, but Law 4 above can then be applied to merge the two assignments inside the '**var**' declaration, and thus allow variable v's final value to be defined in terms of w.

6.2 Semantics of the Legacy Task

We now want to calculate the meaning of the legacy code fragment in Fig. 3 using the semantics in Section 5.2 and the laws in Section 6.1. Firstly, however, we must decide how to model external inputs, i.e., value 'Y' in Table 1. Since these values are sampled from the external environment, the Rate task makes no particular assumptions about them. Therefore, we merely denote the altitude sampled by the i^{th} invocation of the legacy Rate task by symbolic constant A_i.

Also, we must consider the state in which the code fragment of interest begins. Again, this is application-specific. The Rate task's only assumption about the initial system state is that the altitude sampled during its previous invocation must be available in memory location 600. (A dummy value is assumed to have been stored in this location before the Rate task's first invocation by the Operational Flight Program's initialisation code [13].) Therefore, we introduce an assumption to document this requirement of the task's environment.

$$[m(600) = A_{i-1}] \tag{26}$$

For convenience, we also introduce an assumption that the starting time τ of the legacy (and emulated) task is zero.

$$[\tau = 0] \tag{27}$$

This assumption is not essential, but it simplifies the arithmetic during the calculations below.

Thus, the legacy code fragment of interest consists of the sequence of instructions in Fig. 3 preceded by the two assumptions above. Our goal is to use the semantics in Table 1 and the laws in Section 6.1 to calculate the semantics of the whole task. We begin by combining the assumptions above with the semantics of the initial `LOAD` instruction.

$$\begin{aligned}&\text{(assumption 26) ; (assumption 27) ; (instruction 1)}\\ =\ &\text{'by Table 1 and Law 1 (twice)'}\\ &[\mathsf{A}, c, \tau := m(600), m(600), \tau + 10\mu\text{s} \mid\\ &\ m(600) = A_{i-1} \wedge \tau = 0]\\ =\ &[\mathsf{A}, c, \tau := A_{i-1}, A_{i-1}, 10\mu\text{s} \mid\\ &\ m(600) = A_{i-1} \wedge \tau = 0]\\ =\ &\text{'by Law 1'}\\ &[m(600) = A_{i-1} \wedge \tau = 0] \,;\\ &[\mathsf{A}, c, \tau := A_{i-1}, A_{i-1}, 10\mu\text{s}]\end{aligned} \tag{28}$$

In the final step we separated the assumptions again because they are not needed any further below.

This result can then be combined with the next three instructions straightforwardly. The second `LOAD` instruction simply updates register `C`.

$$\begin{aligned}&\text{(statement 28) ; (instruction 2)}\\ =\ &\text{'by Table 1 and Laws 3 and 4'}\\ &[\mathsf{A}, \mathsf{C}, c, \tau := A_{i-1}, 1, 1, 15\mu\text{s}]\end{aligned} \tag{29}$$

The OUT instruction at label READALT writes to port P and in doing so introduces a nondeterministically-defined timestamp.

$$\begin{aligned}&\text{(statement 29)} \mathbin{;} \text{(instruction 3)}\\ =\ &\text{'by Table 1 and Laws 3 and 4'}\\ &[\mathsf{A}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, c, \tau := A_{i-1}, 1, 1, \mathsf{P}^{t'}, 1, 20\mu\text{s} \mid \\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{30}$$

The next LOAD instruction then puts a constant in register B.

$$\begin{aligned}&\text{(statement 30)} \mathbin{;} \text{(instruction 4)}\\ =\ &\text{'by Table 1 and Laws 3 and 4'}\\ &[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, c, \tau := A_{i-1}, 3, 1, 1, \mathsf{P}^{t'}, 3, 25\mu\text{s} \mid \\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{31}$$

At this point we encounter the busy-wait loop at label ADCDELAY. With respect to the semantics in Table 1, we first combine the SUB instruction in the loop body with the delay introduced by the BRGT instruction, to determine the semantics of a single iteration.

$$\begin{aligned}&\text{(instruction 5)} \mathbin{;} [\tau := \tau + 5\mu\text{s}]\\ =\ &\text{'by Table 1 and Law 3'}\\ &[\mathsf{B}, c, \tau := \mathsf{B} - 1, \mathsf{B} - 1, \tau + 10\mu\text{s}]\end{aligned} \tag{32}$$

From Table 1 we therefore have the following semantics for the two instructions at label ADCDELAY.

$$\textbf{rec}\ X \bullet (\text{(statement 32)} \mathbin{;} (([c > 0] \mathbin{;} X) \sqcap [c \leqslant 0]))$$

The semantics of this whole construct is a statement X which makes the statement inside the '**rec**' construct equal X. Based on our understanding of the code's behaviour, we propose the following statement for this purpose.

$$\begin{aligned}&[\mathsf{B}, c, \tau := \min(\mathsf{B} - 1, 0), \min(\mathsf{B} - 1, 0),\\ &\qquad \tau + \max(1, \mathsf{B}) * 10\mu\text{s}]\end{aligned} \tag{33}$$

To understand this, consider that there are two possible behaviours for each iteration of the loop at label ADCDELAY. In the case where the loop condition is already satisfied we decrement register B, taking 10 microseconds to do so, and then exit. This is defined by the first argument in the '$\min$' and '$\max$' operators above. In the second, recursive case, we still do the same decrement, but then loop back to repeat the process. This will ultimately result in register B being decremented until it reaches zero, and will take B 10-microsecond iterations to do so. This is defined by the second argument in the '$\min$' and '$\max$' operators above.

We can confirm that this statement is a suitable fixed point of the recursive construct as follows.

$$\begin{aligned}&\text{(statement 32)} \mathbin{;}\\ &(([c > 0] \mathbin{;} \text{(statement 33)}) \sqcap [c \leqslant 0])\\ =\ &\text{'by Laws 1 and 5'}\\ &[\mathsf{B}, c, \tau := \mathsf{B} - 1, \mathsf{B} - 1, \tau + 10\mu\text{s}] \mathbin{;}\\ &[\mathsf{B}, c, \tau := \mathsf{B}', c', \tau' \mid\\ &\ (c > 0 \wedge \mathsf{B}' = \min(\mathsf{B} - 1, 0) \wedge\\ &\ \ c' = \mathsf{B}' \wedge \tau' = \tau + \max(1, \mathsf{B}) * 10\mu\text{s}) \vee\\ &\ (c \leqslant 0 \wedge \mathsf{B}' = \mathsf{B} \wedge c' = c \wedge \tau' = \tau)]\\ =\ &\text{'by Law 3'}\\ &\text{(statement 33)}\end{aligned}$$

(Indeed, it is the only fixed point because the loop in question is deterministic and terminates from any given starting state.) When applying Law 5 above we treated assumption $[c \leqslant 0]$ as a vacuous assignment $[\mathsf{B}, c, \tau := \mathsf{B}, c, \tau \mid c \leqslant 0]$.

This semantics for the loop can then be combined with that of the preceding instructions.

$$\begin{aligned}&\text{(statement 31)} \mathbin{;} \text{(statement 33)}\\ =\ &\text{'by Law 3'}\\ &[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, c, \tau := A_{i-1}, 0, 1, 1, \mathsf{P}^{t'}, 0, 55\mu\text{s} \mid \\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{34}$$

In particular, statement 31's assignment of 3 to register B is used in the above step to simplify the $\min$ and $\max$ expressions in statement 33. In effect, it tells us that the loop body is executed exactly 3 times.

Instruction 7 then reads a value from input port Q. Recall from Table 1 that its semantics represents this symbolically as 'Y'. Here we instantiate this with 'A_i', to denote the altitude sampled in the Rate task's i^{th} invocation. The IN instruction's semantics also has a preceding assertion. This is eliminated in the first step below since the preceding statement guarantees that the requirement is satisfied.

$$\begin{aligned}&\text{(statement 34)} \mathbin{;} \text{(instruction 7)}\\ =\ &\text{'by Table 1 and Law 2'}\\ &\text{(statement 34)} \mathbin{;}\\ &[\mathsf{B}, c, \tau := A_i, A_i, \tau + 5\mu\text{s}]\\ =\ &\text{'by Law 3'}\\ &[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, c, \tau :=\\ &\ A_{i-1}, A_i, 1, 1, \mathsf{P}^{t'}, A_i, 60\mu\text{s} \mid\\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{35}$$

Although the remaining four instructions complicate the expressions, they are all straightforward state updates. The STORE instruction at label SAVEALT updates the memory array m.

$$\begin{aligned}&\text{(statement 35)} \mathbin{;} \text{(instruction 8)}\\ =\ &\text{'by Table 1 and Laws 3 and 4'}\\ &[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, m, c, \tau :=\\ &\ A_{i-1}, A_i, 1, 1, \mathsf{P}^{t'}, m \oplus \{600 \mapsto A_i\}, A_i, 70\mu\text{s} \mid\\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{36}$$

The next two instructions perform arithmetic on register B.

$$\begin{aligned}&\text{(statement 36)} \mathbin{;} \text{(instruction 9)} \mathbin{;} \text{(instruction 10)}\\ =\ &\text{'by Table 1 and Laws 3 and 4'}\\ &[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, m, c, \tau :=\\ &\ A_{i-1}, (A_i - A_{i-1}) * 20, 1, 1, \mathsf{P}^{t'},\\ &\ m \oplus \{600 \mapsto A_i\}, (A_i - A_{i-1}) * 20, 85\mu\text{s} \mid\\ &\ 15\mu\text{s} < \mathsf{P}^{t'} \leqslant 20\mu\text{s}]\end{aligned} \tag{37}$$

Finally, the last instruction makes a further update to data memory.

$$\begin{array}{l}\quad \text{(statement 37)} \mathbin{;} \text{(instruction 11)}\\ = \text{`by Table 1 and Law 3'}\\ \quad [\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{P}^v, \mathsf{P}^t, m, c, \tau := \\ \quad A_{i-1}, (A_i - A_{i-1}) * 20, 1, 1, \mathsf{P}^{t\prime}, \\ \quad m \oplus \{600 \mapsto A_i, 601 \mapsto (A_i - A_{i-1}) * 20\}, \\ \quad (A_i - A_{i-1}) * 20, 95\mu\text{s} \mid \\ \quad 15\mu\text{s} < \mathsf{P}^{t\prime} \leqslant 20\mu\text{s}]\end{array} \tag{38}$$

Statement 38 thus defines the semantics of the legacy code in Fig. 3. In particular, it tells us that the Rate task stores sampled altitude A_i in memory location 600, and the calculated rate of ascent in location 601. This rate is the difference between the current A_i and previous A_{i-1} altitudes multiplied by 20. Finally, the legacy task's end-to-end execution time is 95 microseconds.

6.3 Semantics of the Emulated Task

In the emulated system, the Rate task consists of a sequence of (interpreted) legacy instructions, emulator actions, and API subroutines, as shown in Fig. 8. Having defined the semantics of all of these operations above, we can determine the semantics of the emulated Rate task in much the same way as we did for the legacy task in Section 6.2.

Again, we model the external input value Z in Table 2 symbolically. Let the altitude sampled in the i^{th} invocation of the 'emulated' task be represented by symbolic constant $\overline{A}_i$. (This value does not necessarily bear any relation to constant A_i in Section 6.2.) As we will see, the emulated system multiplies altitude samples by 3.28, to convert them from metres to feet, so the emulated task's assumption about the value stored in memory location 600 by its previous invocation is as follows.

$$[m(600) = \overline{A}_{i-1} * 3.28] \tag{39}$$

Our goal is thus to calculate the semantics of the sequence of 'mixed language' operations in Fig. 8. We begin by combining the semantics of the first two assembly instructions with that of our overall assumptions. Keep in mind throughout this section that we assume 'emulated' instructions are interpreted faster than instructions executed on the legacy processor, and that all 5-microsecond execution times in Table 1 are thus replaced by 2 microseconds and all 10-microsecond execution times are replaced by 4 microseconds.

$$\begin{array}{l}\quad \text{(assumption 39)} \mathbin{;} \text{(assumption 27)} \mathbin{;}\\ \quad \text{(emulated instruction 1)} \mathbin{;} \text{(emulated instruction 2)}\\ = \text{`by Table 1 and Law 1'}\\ \quad [m(600) = \overline{A}_{i-1} * 3.28 \wedge \tau = 0] \mathbin{;}\\ \quad [\mathsf{A}, \mathsf{C}, c, \tau := \overline{A}_{i-1} * 3.28, 1, 1, 6\mu\text{s}]\end{array} \tag{40}$$

We then encounter a call to the subroutine in Fig. 6, which contains a loop, nested within a local variable block. To calculate the semantics of the whole subroutine we begin at the most deeply nested construct and work outwards. We therefore begin with the loop body and determine the semantics of a single occurrence of i/o statement 14 followed by the overhead of evaluating the loop condition and branching. We assume that it takes the emulator 2 microseconds to evaluate expression 'Busy == 1' and branch accordingly. As mentioned above, this duration could be established through static analysis techniques [30].

$$\begin{array}{l}\quad \text{(statement 14)} \mathbin{;} [\tau := \tau + 2\mu\text{s}]\\ = \text{`by Table 2 and Law 3'}\\ \quad (\{\mathsf{T}^v = 1\} \mathbin{;}\\ \quad [\text{Busy}, \tau := \text{Busy}', \tau + 4\mu\text{s} \mid \\ \quad ((\tau' - \mathsf{T}^t < 14\mu\text{s}) \Rightarrow \text{Busy}' = 1) \wedge \\ \quad ((\tau' - \mathsf{T}^t \geqslant 14\mu\text{s}) \Rightarrow \text{Busy}' = 0)])\end{array} \tag{41}$$

The 'ReadIOPort(V)' operation has a preceding assertion which we must retain until it can proven to hold. For clarity below we have also reexpressed the predicate in terms of the finishing time τ', rather than the starting time τ.

From the definition of **do-while** loops in Table 2 we therefore have the following semantics for iterative statement 15. (We use mathematical, rather than C++, notation for expressions in the semantics.)

$$\begin{array}{l}\textbf{rec}\ X \bullet (\text{(statement 41)} \mathbin{;} \\ \qquad (([\text{Busy} = 1] \mathbin{;} X) \sqcap [\text{Busy} \neq 1]))\end{array}$$

In this case we propose the following statement for the loop's fixed point. Let $\mathbb{N}_1$ denote the positive natural numbers (excluding zero).

$$\begin{array}{l}(\{\mathsf{T}^v = 1\} \mathbin{;}\\ [\text{Busy}, \tau := 0, \min\{u \mid u - \mathsf{T}^t \geqslant 14\mu\text{s} \wedge \\ \qquad (\exists n : \mathbb{N}_1 \bullet u = \tau + n * 4\mu\text{s})\}])\end{array} \tag{42}$$

The whole loop retains the preceding assertion stating that control register T has previously been assigned value 1. The loop finishes only when variable 'Busy' equals zero. The time τ is increased by 4-microsecond increments (i.e., the execution time of the loop body, statement 41) until it exceeds timestamp T^t by 14 microseconds (rather than 10 microseconds because the 4-microsecond loop body will be executed even if the loop is reached after the end of the ADC's 10-microsecond busy interval).

We can confirm that this statement is indeed a suitable fixed point of the recursive construct above as follows.

$$\begin{array}{l}\quad \text{(statement 41)} \mathbin{;}\\ \quad (([\text{Busy} = 1] \mathbin{;} \text{(statement 42)}) \sqcap [\text{Busy} \neq 1])\\ = \text{`by removing the redundant assertion and Law 1'}\\ \quad \text{(statement 41)} \mathbin{;}\\ \quad ([\text{Busy}, \tau := 0, \min\{u \mid \cdots\} \mid \text{Busy} = 1] \sqcap \\ \quad [\text{Busy} \neq 1])\\ = \text{`by Law 5'}\\ \quad \text{(statement 41)} \mathbin{;}\\ \quad [\text{Busy}, \tau := \text{Busy}', \tau' \mid \\ \quad (\text{Busy} = 1 \wedge \text{Busy}' = 0 \wedge \tau' = \min\{u \mid \cdots\}) \vee \\ \quad (\text{Busy} \neq 1 \wedge \text{Busy}' = \text{Busy} \wedge \tau' = \tau)]\end{array}$$

$=$ 'by Law 3'

(statement 42)

In the first step we eliminated assertion $\{\mathsf{T}^v = 1\}$ from statement 42 because this assertion already appears in statement 41 and none of the intervening statements updates T^v. (Although we have not presented a law for handling this specific situation, it is an application of the principles of propagation of assertions through compound statements [1, §28.3].) In the second step we treated assumption $[\text{Busy} \neq 1]$ as a vacuous assignment.

The loop semantics can then be combined with that of the first statement in the subroutine which writes to the ADC's control register at port T. In particular, this knowledge allows us to elimate the assertion concerning T^v and to determine the number of iterations.

(statement 13) **;** (statement 42)

$=$ 'by Table 2 and Law 2'

$[\mathsf{T}^v, \mathsf{T}^t, \tau := 1, \mathsf{T}^{t'}, \tau + 2\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau'] \,;$
$[\text{Busy}, \tau := 0, \min\{u \mid u - \mathsf{T}^t \geqslant 14\mu\text{s} \wedge (\exists n : \mathbb{N}_1 \bullet u = \tau + n * 4\mu\text{s})\}]$

$=$ 'by Laws 3 and 4'

$[\text{Busy}, \mathsf{T}^v, \mathsf{T}^t, \tau := 0, 1, \mathsf{T}^{t'}, \tau' \mid \tau < \mathsf{T}^{t'} \leqslant \tau + 2\mu\text{s} \wedge \tau' = \min\{u \mid u - \mathsf{T}^t \geqslant 14\mu\text{s} \wedge (\exists n : \mathbb{N}_1 \bullet u = \tau + 2\mu\text{s} + n * 4\mu\text{s})\}]$

$= [\text{Busy}, \mathsf{T}^v, \mathsf{T}^t, \tau := 0, 1, \mathsf{T}^{t'}, \tau + 18\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau + 2\mu\text{s}]$ (43)

The final step recognises that the only possible value for the number of times n that the loop body is performed is 4. Thus we have determined the number of iterations from the duration of the loop body and the ADC's busy signal.

This result can be combined with the next statement in the subroutine. Statement 16 reads from the ADC's data port U. In this case we assume the application-specific input value Z in Table 2 denotes the altitude sampled from the new radar altimeter hardware, here represented by symbolic constant $\overline{A}_i$. Table 2 also tells us that this statement has an initial assertion requiring '1' to have been written to control register T at least 10 microseconds ago. Fortunately, preceding statement 43 immediately satisfies this.

(statement 43) **;** (statement 16)

$=$ 'by Table 2 and Law 2'

$[\text{Busy}, \mathsf{T}^v, \mathsf{T}^t, \tau := 0, 1, \mathsf{T}^{t'}, \tau + 18\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau + 2\mu\text{s}] \,;$
$[\text{Altitude}, \tau := \overline{A}_i, \tau + 2\mu\text{s}]$

$=$ 'by Laws 3 and 4'

$[\text{Altitude}, \text{Busy}, \mathsf{T}^v, \mathsf{T}^t, \tau := \overline{A}_i, 0, 1, \mathsf{T}^{t'}, \tau + 20\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau + 2\mu\text{s}]$ (44)

The remaining four statements within the subroutine are semantically all simple state updates.

(statement 44) **;** (statements 17 to 20)

$=$ 'by Table 2 and Laws 3 and 4'

$[\text{Altitude}, \text{Busy}, \mathsf{B}, \mathsf{T}^v, \mathsf{T}^t, c, \tau := \overline{A}_i * 3.28, 0, \overline{A}_i * 3.28, 1, \mathsf{T}^{t'}, \overline{A}_i * 3.28, \tau + 28\mu\text{s} \mid \tau < \mathsf{T}^{t'} \leqslant \tau + 2\mu\text{s}]$ (45)

This defines the behaviour of statements 13 to 20 in Fig. 6. We now incorporate this result into the definition of declarative block 12. With respect to the semantics in Table 2, we assume that it takes 2 microseconds for the emulator to allocate or deallocate space for each integer-valued variable.

var Altitude, Busy : $\mathbb{Z} \bullet$
$([\tau := \tau + 4\mu\text{s}] \,;$ (statement 45) $\,;\, [\tau := \tau + 4\mu\text{s}])$

$=$ 'by Law 3'

var Altitude, Busy : $\mathbb{Z} \bullet$
$[\text{Altitude}, \text{Busy}, \mathsf{B}, \mathsf{T}^v, \mathsf{T}^t, c, \tau := \overline{A}_i * 3.28, 0, \overline{A}_i * 3.28, 1, \mathsf{T}^{t'}, \overline{A}_i * 3.28, \tau + 36\mu\text{s} \mid \tau + 4\mu\text{s} < \mathsf{T}^{t'} \leqslant \tau + 6\mu\text{s}]$

$=$ 'by Law 4'

var Altitude, Busy : $\mathbb{Z} \bullet$
$([\text{Altitude}, \text{Busy} := \overline{A}_i * 3.28, 0] \,;$
$[\mathsf{B}, \mathsf{T}^v, \mathsf{T}^t, c, \tau := \overline{A}_i * 3.28, 1, \mathsf{T}^{t'}, \overline{A}_i * 3.28, \tau + 36\mu\text{s} \mid \tau + 4\mu\text{s} < \mathsf{T}^{t'} \leqslant \tau + 6\mu\text{s}])$

$=$ 'by Law 6'

$[\mathsf{B}, \mathsf{T}^v, \mathsf{T}^t, c, \tau := \overline{A}_i * 3.28, 1, \mathsf{T}^{t'}, \overline{A}_i * 3.28, \tau + 36\mu\text{s} \mid \tau + 4\mu\text{s} < \mathsf{T}^{t'} \leqslant \tau + 6\mu\text{s}]$ (46)

In the final step the locally-scoped high-level language variables are eliminated.

By preceding this statement with the overhead of the '**call**' statement we obtain the complete semantics of the first subroutine call in Fig. 8.

call ; (statement 46)

$=$ 'by Table 2 and Law 3'

$[\mathsf{B}, \mathsf{T}^v, \mathsf{T}^t, c, \tau := \overline{A}_i * 3.28, 1, \mathsf{T}^{t'}, \overline{A}_i * 3.28, \tau + 40\mu\text{s} \mid \tau + 8\mu\text{s} < \mathsf{T}^{t'} \leqslant \tau + 10\mu\text{s}]$ (47)

This block can then be placed in the context of the preceding and succeeding assembler instructions. (Again, keep in mind the faster execution times for emulated instructions.)

(statement 40) **;** (statement 47) **;**
(emulated instructions 8 to 10)

$=$ 'by Table 1 and Laws 3 and 4'

$[\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{T}^v, \mathsf{T}^t, m, c, \tau := \overline{A}_{i-1} * 3.28, (\overline{A}_i - \overline{A}_{i-1}) * 3.28 * 20, 1, 1, \mathsf{T}^{t'}, m \oplus \{600 \mapsto \overline{A}_i * 3.28\}, (\overline{A}_i - \overline{A}_{i-1}) * 3.28 * 20, 56\mu\text{s} \mid 14\mu\text{s} < \mathsf{T}^{t'} \leqslant 16\mu\text{s}]$ (48)

We then encounter the second subroutine call in Fig. 8 and again calculate its semantics 'inside out.' The four state-

ments inside the variable block are simple state updates.

$$\begin{aligned}
&\text{(statements 22 to 25)}\\
&= \text{‘by Table 2 and Laws 3 and 4’}\\
&\quad [\text{AscentRate}, \mathsf{B}, c, \tau := \mathsf{B} * 1.25, \mathsf{B} * 1.25, \\
&\qquad\qquad \mathsf{B} * 1.25, \tau + 8\mu\text{s}] \qquad (49)
\end{aligned}$$

As we did for the first subroutine, we can then add the overheads of allocating and deallocating space for variable ‘AscentRate’, and then hide the local variable entirely.

$$\begin{aligned}
&\textbf{var}\ \text{AscentRate} : \mathbb{Z} \bullet\\
&\quad ([\tau := \tau + 2\mu\text{s}]\ \textbf{;}\ \text{(statement 49)}\ \textbf{;}\ [\tau := \tau + 2\mu\text{s}])\\
&= \text{‘by Laws 3, 4 and 6’}\\
&\quad [\mathsf{B}, c, \tau := \mathsf{B} * 1.25, \mathsf{B} * 1.25, \tau + 12\mu\text{s}] \qquad (50)
\end{aligned}$$

Adding the overhead of the **call** statement then completes the semantics of the second subroutine call in Fig. 8. The result clearly matches the intention of this simple subroutine.

$$\begin{aligned}
&\textbf{call}\ \textbf{;}\ \text{(statement 50)}\\
&= \text{‘by Table 2 and Law 3’}\\
&\quad [\mathsf{B}, c, \tau := \mathsf{B} * 1.25, \mathsf{B} * 1.25, \tau + 16\mu\text{s}] \qquad (51)
\end{aligned}$$

Finally, we can put this statement in the context of the preceding code and the remaining emulated instruction to complete the semantics of the emulated task.

$$\begin{aligned}
&\text{(statement 48)}\ \textbf{;}\ \text{(statement 51)}\ \textbf{;}\\
&\text{(emulated instruction 11)}\\
&= \text{‘by Table 1 and Law 3’}\\
&\quad [\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{T}^v, \mathsf{T}^t, m, c, \tau := \qquad (52)\\
&\quad \overline{A}_{i-1} * 3.28, (\overline{A}_i - \overline{A}_{i-1}) * 3.28 * 20 * 1.25,\\
&\quad 1, 1, \mathsf{T}^{t\prime},\\
&\quad m \oplus \{600 \mapsto \overline{A}_i * 3.28,\\
&\qquad 601 \mapsto (\overline{A}_i - \overline{A}_{i-1}) * 3.28 * 20 * 1.25\},\\
&\quad (\overline{A}_i - \overline{A}_{i-1}) * 3.28 * 20 * 1.25, 76\mu\text{s} \mid\\
&\quad 14\mu\text{s} < \mathsf{T}^{t\prime} \leqslant 16\mu\text{s}]
\end{aligned}$$

In particular, this result tells us that memory location 600 contains the sampled altitude (converted from metres to feet) and location 601 contains the difference between the last two altitude samples (converted to feet) multiplied by 25. The overall execution time of the emulated task is 76 microseconds.

6.4 Comparison of the Legacy and Emulated Semantics

Statements 38 and 52 confirm that the legacy and emulated Rate tasks do *not* have precisely the same behaviour. Therefore, the change impact analysis must be completed by justifying the differences in the light of the Mission Computer System’s hardware upgrade. In particular, the major concerns for code in a multi-tasking Operational Flight Program are: the task’s functional behaviour; code that is dependent on instruction execution speeds; and code that depends on the frequency of task invocations.

Overall, the Rate task’s functional behaviour is preserved. The semantics show that both versions update memory locations 600 and 601 (and general-purpose registers A, B and C, and the comparison register c). The legacy task writes to port P, whereas the emulated one uses port T, but this is explained by the replacement of the Analog-to-Digital Converter. Also, recalling that the old altimeter was calibrated in feet, while the new one produces readings in metres, explains the way that the emulated task scales all altimeter readings $\overline{A}_i$ by 3.28, to maintain consistency with the legacy code. (Notably, the altitude stored by the emulated task in location 600 is measured in feet, so that other tasks accessing this value are not impacted by the changes to the Rate task. Similarly, for the ascent rate stored by the emulated task in location 601.)

With regard to instruction execution speeds, the most obvious difference is that the legacy Rate task takes 95 microseconds where the emulated one takes only 76. However, a *faster* task execution time is (usually) acceptable in cyclic-executive scheduling because it makes it easier for the task invocations to fit into their frame. (Curiously, this ‘improvement’ *can* change the behaviour of some systems by allowing tasks that previously always overran the frame to run to completion.) More importantly, we must beware of task code that relies for its correctness on instruction execution speeds. The busy-wait loop at label ADCDELAY in Fig. 3 has this characteristic but, in this case, the programmer has correctly patched the code with the subroutine in Fig. 6. The semantics show that both versions of the task successfully sample altitudes from their respective ADCs, when the specific timing characteristics of the hardware interface are included. (The timestamps associated with output ports P and T also reveal that the two versions of the task access their ADCs at different times, relative to the task’s starting time, but the absolute timing of i/o events *within* a task invocation is usually unimportant in a cyclic executive design.)

Finally, we must consider code that depends on the frequency of task invocations. Typically, the programmer of a periodic task relies on the task being invoked regularly (with as little ‘jitter’, i.e., variation between successive invocations, as possible) but the *absolute* timing of frames is unimportant. Generally speaking, there is no definable relationship between an altitude A_i sampled by the legacy task and the corresponding altitude $\overline{A}_i$ sampled by the emulated task. (Even if the two systems were started at the same time, the different task frequencies would mean that these two values are unrelated.) What *is* meaningful in this application, however, is the difference between *consecutive* samples. Thus, expression ‘$(A_i - A_{i-1}) * 20$’ in the legacy task’s semantics denotes the change in altitude in one second. The corresponding expression in the emulated task’s semantics, ‘$(\overline{A}_i - \overline{A}_{i-1}) * 20 * 1.25$’, is meant to denote the same value. To allow for the fact that minor cycles are 50 milliseconds long in the legacy system but only 40 milliseconds long in the emulated system, the emulated task correctly scales the measurement by $\frac{50}{40} = 1.25$.

We have now accounted for all the significant differences between the legacy and emulated semantics and can thus con-

clude that the emulated system is indeed a satisfactory replacement for the legacy one.

7 Conclusion

Changes to safety and mission-critical control systems must be made with utmost care. Ideally, therefore, formal techniques should be used for change impact analyses of control systems reimplemented using processor emulation. We have shown how this can be done by pragmatically combining semantic modelling with an intuitive understanding of the old and new software architectures.

In particular, we identified an elegant modelling notation based on relational assignments, defined a common semantics for actions of both the legacy and emulated systems in terms of the legacy processor's state, modelled essential timing properties via a special time variable, captured important features of external hardware devices in the semantics of the statements that access them, and formally calculated the semantics of both the old and new software. The impact of the software upgrade on the system's behaviour was thus clearly revealed, potentially alerting the programmer to unanticipated side effects, and providing an opportunity to justify any differences in terms of the specific application.

In current research we are investigating ways in which the theory described in this paper can be made more accessible to practising programmers. For instance, a significant issue is how the program code that needs to be modified in response to hardware changes can be identified. We have already achieved promising results through dimensional analysis, using compiler-like type inference, to allow statements whose units of measurement have been invalidated by the hardware change to be identified automatically.

Acknowledgements. I wish to thank Geoffrey Watson for his insights into the airworthiness certification issues associated with processor emulation and Luke Wildman for advice on relational program semantics. This research was funded by Australian Research Council Large Grant A00104650, *Verified Compilation Strategies for Critical Computer Programs*.

References

1. R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
2. R.-J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (REX Workshop 1989)*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1989.
3. T. P. Baker and A. Shaw. The cyclic executive model and Ada. *Journal of Real-Time Systems*, 1(1):7–26, June 1989.
4. L. Beus-Dukic. COTS real-time operating systems in space. *Safety Systems: The Safety-Critical Systems Club Newsletter*, 10(3):11–14, May 2001.
5. E. Börger and I. Durdanović. Correctness of compiling occam to transputer code. *The Computer Journal*, 39(1):52–92, 1996.
6. C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237. IEEE Computer Society Press, 1998.
7. R. A. Comfort. The economics of microprocessor obsolescence. *COTS Journal*, pages 21–23, July/August 1998.
8. D. Corman, P. Goertzen, J. Luke, and M. Mills. Incremental Upgrade of Legacy Systems (IULS): A fundamental software technology for aging aircraft. In *Fourth Joint DOD/FAA/NASA Conference on Aging Aircraft*, 2000.
9. D. Culpin. Overcoming technology lag in mission computers. *Australian Defence Science*, 11(1):4–5, 2003.
10. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
11. J. D. G. Falardeau. Schedulability analysis in rate monotonic based systems with application to the CF-188. Master's thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada, May 1994.
12. C. J. Fidge. Timing analysis of assembler code control-flow paths. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 370–389. Springer-Verlag, 2002.
13. C. J. Fidge. Verifying emulation of legacy mission computer systems. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 187–207. Springer-Verlag, 2003.
14. P. Gust. *Introduction to Machine and Assembly Language Programming*. Prentice-Hall, 1986.
15. I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385–448, 2001.
16. J. Hooman. Extending Hoare logic to real-time. *Formal Aspects of Computing*, 6(6A):801–825, 1994.
17. D. Kalinsky. Context switch. *Embedded Systems Programming*, 14(2):94–105, February 2001.
18. K. Kleiner. I ♡ Space Invaders. *New Scientist*, (2313):46–48, October 2001.
19. K. Lermer and C. J. Fidge. A formal model of real-time program compilation. *Theoretical Computer Science*, 282(1):151–190, July 2002.
20. L. A. Leventhal. *Introduction to Microprocessors: Software, Hardware and Programming*. Prentice-Hall, 1978.
21. C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *The Journal of Real-Time Systems*, 4:37–53, 1992.
22. C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, Software Engineering Insititute, Carnegie Mellon University, December 1990.
23. J. A. Luke, D. G. Haldeman, and W. J. Cannon. A COTS-based replacement strategy for aging avionics computers. *CrossTalk—The Journal of Defense Software Engineering*, pages 14–17, December 2001.
24. C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
25. M. Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

26. RTCA, Inc. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992. Special Committee 167 Document No. RTCA/DO-178B.
27. A. Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *AMAST Series in Computing*. World Scientific, 1997.
28. D. Scholefield. Real-time refinement in Manna and Pnueli's temporal logic. *Formal Aspects of Computing*, 8(4):408–427, 1996.
29. E. Sekerinski. A calculus for predicative programming. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction (MPC'93)*, volume 669 of *Lecture Notes in Computer Science*, pages 302–322. Springer-Verlag, 1993.
30. A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.
31. J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, January 1988.
32. S. Stepney. *High Integrity Compilation: A Case Study*. Prentice-Hall, 1993.
33. D. B. Stewart. 30 pitfalls for real-time software developers, part 1. *Embedded Systems Programming*, 12(1):32–41, October 1999.
34. TRW Inc. Emulator Application Programming Interface (API) for the 1750A Virtual Component Environment (VCE1750A). Technical Report HML-API-001, TRW Dayton Engineering Laboratory, March 2001. Revision D.
35. U.S. Department of Transportation Federal Aviation Administration. *Guidelines for the Oversight of Software Change Impact Analyses Used to Classify Software Changes as Major or Minor*, 2000. FAA Notice N8110.85.
36. U.S. Department of Transportation Federal Aviation Administration. *Guidelines for the Approval of Software Changes in Legacy Systems Using RTCA DO-178B*, 2001. FAA Notice N8110.89.

A Semantics of Modelling Language Primitives

Section 5.1 above introduced several well-known primitives for modelling programming language code. For completeness, this appendix presents their weakest precondition semantics. In particular, these definitions are sufficient to prove the correctness of the laws in Section 6.1.

A *weakest precondition*, wp.$S.R$, is a predicate characterising those initial states from which statement S is guaranteed to terminate in a state satisfying a given postcondition predicate R [10, p. 128]. The full stops denote left-associative function application, so wp.$S.R$ is equivalent to $(\text{wp}(S))(R)$ and wp.S is thus a higher-order function on predicates, called a *predicate transformer*.

Table 3 gives the weakest precondition semantics for our modelling language primitives. Let P and R be predicates on the program state; v be a program variable; T be a type; G be a predicate on the program state which may also refer to v'; S be a statement in our modelling language; X be a statement-valued variable; and $S(X)$ be a compound statement that may contain 'X' where a statement is expected.

Table 3. Weakest precondition semantics for modelling language primitives

Statement S	Semantics wp.$S.R$
$[P]$	$P \Rightarrow R$
$\{P\}$	$P \wedge R$
$[v$ **:=** $v' \mid G]$	$(\exists v' \bullet G) \wedge (\forall v' \bullet G \Rightarrow R[v'/v])$
S_1 **;** S_2	wp.S_1.(wp.$S_2.R$)
var $v : T \bullet S$	$\bigwedge_{v \in T}$ wp.$S.R$, for v not free in R
$S_1 \sqcap S_2$	wp.$S_1.R \wedge$ wp.$S_2.R$
rec $X \bullet S(X)$	$(\mu X \bullet S(X)).R$

Also let $\mu X \bullet S(X)$ denote the least fixed point of the higher-order function wp.$S(X)$ [1, §20.1]. In other words, it is a statement X such that $X = S(X)$. Where there is more that one such statement we choose the 'worst' one in the sense that it is the least deterministic and is guaranteed to terminate from the smallest set of initial states.

The semantics for most of the primitives in Table 3 are conventional. For an assertion $\{P\}$ to achieve postcondition R it must be the case that R already holds and that the program has already made P true [1, p. 189]. Similarly, an assumption $[P]$ can make R true only if R already holds, but the statement is obliged to be well-behaved only if the program's environment has established P [1, p. 189]. Sequential composition 'S_1 **;** S_2' is modelled as functional composition of the predicate transformers corresponding to the two statements [1, p. 189]. A variable declaration '**var** $v : T \bullet S$' will make R true provided that statement S does so for any initial value of freshly-declared variable v. The semantics has a proviso that says variable v may not occur free in postcondition predicate R, thus forcing 'v' to be a previously unused name [1, p. 227]. To guarantee that a demonic nondeterministic choice '$S_1 \sqcap S_2$' makes R true, both alternatives must establish R [1, p. 189]. For a recursive construct to achieve R, the least fixed point of the statement must do so.

The only uncommon statement in Table 3 is the relational assignment, $[v$ **:=** $v' \mid G]$. Here we use the 'strict' form [2, p. 58] which is well-behaved only if there exists some final value v' that makes G true. This is appropriate for our case study since we model executable programming language constructs only, and avoid operators such as division that may be undefined for certain values of their operands. The second conjunct in the semantics says that such an assignment will achieve postcondition R, reexpressed using variable v's final value v', provided that R is established by any final value v' that satisfies G.