

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

**Queensland 4072
Australia**

TECHNICAL REPORT

No. 93-10

**Formally Verified Real-Time Software:
an Integrated Development Strategy (Extended Version)**

C. Fidge and P. Kearney and J. Staples

June 1993

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Formally Verified Real-Time Software: an Integrated Development Strategy*

C. Fidge P. Kearney J. Staples
Software Verification Research Centre
The University of Queensland
Queensland 4072 Australia

Abstract

The demands placed on programmers by the needs of safety-critical systems mandate the use of formal methods, supported by an integrated toolkit, throughout software development. Many formal specification, design and verification techniques have been proposed in the past. As the next step in the maturation of these techniques we are aiming to integrate previously disparate methods. Our overall methodology is illustrated with formal methods developed by the University of Queensland's Software Verification Research Centre, using the "boiler water content monitor and control system" as a case study. Practical problems posed by this application are discussed.

1 Introduction

Formal methods are essential for safety-critical software development. No amount of testing and debugging has the potential to offer as much assurance of correctness as can be achieved using formal specification, design and verification methods. Formal methods also have the potential for greater efficiency, by making it possible to detect errors as early as possible in the software development process and thus avoid costly redevelopment.

Despite these promises however, formal methods remain largely inaccessible to programmers "at the coalface". This is not surprising. Existing methods are not comprehensive; they treat the various problems of software development in a piecemeal fashion. For instance, although many specification and programming

*Extended version. A shorter version of this paper was presented at the *International Symposium on Design and Review of Software Controlled Safety-Related Systems*, Waterloo, Canada, 28-29 June 1993.

language notations have been proposed for real-time systems, few development approaches link the two. Also most contemporary methods are still experimental: they are not robust or well documented, nor are they supported by the development tools needed to make them suitable for immediate practical application. A large number of development methodologies have been proposed for real-time systems, typically by upgrading existing approaches [14, 10], but few achieve a high degree of formality at all stages of software development.

Recognising these deficiencies, we propose to integrate formal methods for real-time systems development, in a way which has good potential for tool support. This paper briefly outlines the overall approach and then describes its application to a substantial case study, the boiler water content monitor and control system.

We advocate using a formal description technique for specification, rigorous design rules for software development and formal verification methods for confirming real-time behaviour. As a practical instantiation of the methodology we show how it can be achieved by building on established techniques and how this instantiation can be applied to the boiler water monitor and control system considered in the Generic Problem Exercise. The language Z is used for formal specification, with appropriate conventions used for representing timing requirements. A refinement calculus is used for developing C code from the specification, with extra side-conditions introduced for expressing timing obligations. Finally, an induction-based verification technique is used for discharging these timing obligations, using timing prediction rules based on the particular target architecture and compiler. Integration of these techniques is being achieved by using the specified timing requirements to direct development into code and by using the timing obligations generated during development as the basis for timing verification. We wish to make clear, however, that the integration of these techniques is work in progress, by ourselves and others.

2 Methodology

This section briefly outlines the key features of the methodology.

2.1 Formal description techniques

A formal specification can unambiguously express both the intended behaviour of a proposed system and also its timing characteristics. The value of formal description techniques for real-time systems is being increasingly recognised but, as yet, there is no consensus on which notation is best. Numerous alternative approaches are possible [7]. Time may be discrete or continuous. Events may be considered as time-consuming or instantaneous. Timing requirements may be expressed relative to previous events or in absolute terms. Alternative behaviours

may be expressed via a set of linear histories or a single branching history. There may be a single frame of reference for expressing time or numerous, possibly unsynchronised, reference points. Time values may be totally or partially ordered.

For the moment, therefore, we suggest using a general-purpose specification language, with rigorously-applied conventions for modelling time as appropriate for the particular application. Whichever model is adopted, however, it must also be suitable for use during the development phase. The specified timing requirements must be precise enough for use in determining whether a development step is valid or not. They must also provide an unambiguous definition against which to verify the final code.

2.2 Formal top-down development

There is presently a disturbing shortage of development techniques for real-time software [6]. However, recent developments in rigorous software development laws offer ways of formally turning abstract specifications into concrete code.

We are therefore building on these techniques, so that they account for the timing, as well as functional, behaviour of the proposed system. The timing requirements in the formal specification are used to determine which refinement laws may be applied. Timing requirements that remain unproven in the design are retained as obligations that must later be discharged for each development step to be considered valid. These obligations must accurately reflect the timing requirements in the specification and must be expressed in a way usable by the verification methodology.

2.3 Formal verification of generated timing requirements

Formal verification techniques for real-time behaviour of assembly-level code have already been successfully developed [11]. However greater efficiency is possible by verifying at the source code level. This allows reasoning during software development to be undertaken using the same methods as that used during verification. On the other hand it introduces the problem of dealing effectively with the range of compilers that may be applied to higher-level source code. Since different compilations of high-level source code may have different real-time behaviours, high-level reasoning inevitably is dependent to some extent on compiler characteristics. Our aim however is to factor out this dependence so that it is confined to a separate secondary stage of reasoning which has good potential for being achieved automatically by an appropriate integration of compilation and inference capabilities.

In brief, timing obligations developed during system development are discharged during a later verification phase. This, however, brings with it the chal-

length of effectively dealing with the cumulative timing uncertainty that results from attempting to predict the execution time of high-level language statements, without full knowledge of the compiler optimisation strategies and target architecture. Verification of code execution times takes place against the timing obligations generated during development (which are themselves developed from the specified timing requirements).

3 Case study

This section shows how the above methodology can be followed using software development techniques currently being developed for real-time systems work. The “boiler water content monitor and control system” [1] is used as a case study.

3.1 A core example

The full requirements document for the boiler system is quite detailed. There are many types of incoming message that must be considered, and numerous types of failure that must be handled. Although these issues are important in a complete system, the methods used to treat them are largely similar. To achieve a comprehensive coverage of the range of issues involved, within the constraints of available time, we concentrated on a “core” problem which removes repetitive features while retaining those aspects essential for establishing timing and safety. Specifically, our core requirements specification differs from the full specification [1] in the following ways.

- Incoming transmissions from the instrumentation system are defined to each consist of a single message. This message always includes the current water level as measured by the water content measuring device. It is assumed that if the beginning of a transmission is seen (i.e., the STX character is received) then the remainder of the transmission will always be successfully received.
- Outgoing transmissions from the program are defined to consist of a single message. This message always includes an indication of whether the feedpumps should be switched on or off.
- Access to the steaming rate measuring device [1, §2.3.5] and the feedpump monitors [1, §2.3.3] is omitted.
- Interactions with the host operating system [1, §4.2.2a] and the operator [1, §4.9] are omitted, apart from the audio warning device.

However all timing requirements are retained, as are some troublesome features that might compromise safety, in particular the ability of transmissions to be

delayed [1, §C-1.7], corrupted or lost [1, §C-1.6]. We do not claim that every potential hazard that occurs in the full specification has been retained, but we believe that we have retained a representative sample. Working with this subset specification helped us to visualise the overall structure of the problem, without being distracted by repetitive detail. The remainder of this paper presents the work undertaken on the core system; however the techniques are applicable to the full requirements specification – the difference is one of scale rather than kind.

As a running example to illustrate how our methodology contributes towards the satisfaction of safety requirements, the sections below focus on a particular requirement that can be seen as an important pre-requisite for safety. We consider the case when the program is in normal operating mode and a single transmission from the instrumentation system is anticipated and successfully received, which reports an unsafe boiler water level. We show how verified code may be produced which conforms with the requirement that normal operating mode will be exited [1, §4.4.2a] and the audio warning device will be activated within 5 seconds [1, §4.7.1].

3.2 Specification

A formal specification of the core system was written as a 19 page (handwritten, uncommented) Z document. It was developed over time, involving three major revisions. Considerable effort was devoted to finding a format that offered a good compromise between the needs of readability, traceability back to the requirements document, traceability of the forthcoming design process, and amenability to the verification steps. Such an expenditure of effort in this early phase of system development is justified easily in light of the way it simplifies later steps.

Indeed, the mere act of writing a formal specification was beneficial in forcing us to consider problems that may not be evident from merely reading the natural language requirements specification. Some of the questions that were raised during preparation of the formal specification are listed in Appendix A.

Z [21] is a powerful specification language based on set theory, with a modularisation mechanism expressed using named boxes, called schemata. It is a general-purpose language; although it does not have any predefined notation for representing timing requirements, these can be declared.

Z users often strive to take advantage of the power of the notation by specifying systems at as high a level of abstraction as possible. In this instance, however, we elected to let the structure of the natural language requirements specification [1] dictate the structure of the formal specification (and ultimately the implementation). Although this approach can be criticised for restricting the range of possible implementations, it has an important advantage for safety-critical work: traceability. Features in the specification can be readily traced back to their origin in

the requirements specification.

Also, because the boiler requirements document frequently spelled out the order in which actions were to occur (e.g., [1, §4.2.1]) and the way code was to be written (e.g., [1, §4.8.3]), it was considered efficient to follow these directions.

This section presents those parts of the Z specification relevant to establishing the property described above.

3.2.1 Types

Data types declared included

$$\begin{aligned} \textit{Transmissions} &== \textit{seq Char} \\ \textit{Modes} &::= \textit{SelfTestMode} \mid \textit{SystemTestMode} \mid \textit{NormalMode} \mid \\ &\quad \textit{DegradedMode} \mid \textit{EmergencyMode} \mid \textit{ShutDownMode} \\ \textit{Status} &::= \textit{On} \mid \textit{Off} \\ \textit{Times} &== \mathbb{R} \\ \textit{Intervals} &== \mathbb{N} \end{aligned}$$

where incoming and outgoing *Transmissions* are declared as a sequence of characters [1, Annex A], the various system *Modes* [1, §4.1] and the *Status* of devices such as the pumps and the siren are given as enumerated types, absolute *Times* are expressed using real numbers (representing seconds, as measured by the “master” timer) [1, §4.8.1], and the periodic *Intervals* used by the instrumentation system are represented by natural numbers [1, §4.8.1].

3.2.2 Variables

The following specification variables are used to describe aspects of the interface to the environment.

Each incoming transmission from the instrumentation system is associated with a particular interval [1, §4.8.1] and for each such transmission there is an associated arrival time.

$\begin{aligned} \textit{Inputs} & \\ \textit{incoming} &: \textit{Intervals} \rightarrow \textit{Transmissions} \\ \textit{whenseen} &: \textit{Intervals} \rightarrow \textit{Times} \end{aligned}$

The symbol \rightarrow states that these are partial functions only; it is possible that no transmission will be received in some intervals [1, §C-1.6.2]. Similarly, for transmissions from the program to the instrumentation system:

Outputs

outgoing : *Intervals* \leftrightarrow *Transmissions*
whensent : *Intervals* \leftrightarrow *Times*

Messages to activate or deactivate the audio shutdown warning [1, §4.7.1] are associated with the time at which they are sent.

Audio

siren : *Times* \leftrightarrow *Status*

The incoming sequence of transmissions from the instrumentation system is assumed to be supplied by the environment and cannot be modified, whereas the outgoing transmissions from the program are to be directly controlled by the program. The specification can be viewed as defining the way in which *outgoing*, *whensent* and *siren* are constructed, given particular values for *incoming* and *whenseen*.

Important characteristics of the incoming transmissions, such as whether the transmission was delayed [1, §C-1.7], entirely absent [1, §C-1.6.2], syntactically invalid [1, §C-1.6.3], or violated the message protocol [1, §B-3.1.1.3], are modelled by variables representing the set of intervals in which this characteristic is true. Also modelled explicitly are those intervals in which a shutdown is necessary, due either to there being no communication received for a number of intervals [1, §C-1.6.2], or to the received transmissions being too frequently corrupted [1, §C-1.6.2], or to the water content reaching unsafe limits [1, §4.4.2].

Properties

delayed, failed, badsyntax, badprotocol : \mathbb{P} *Intervals*
nocomm, toocorrupt, unsafe : \mathbb{P} *Intervals*

\mathbb{P} *Intervals* is the set of subsets of *Intervals*.

Specification variables describing the operation of the control program indicate which mode the program is in [1, §4.9.1a], the current interval number and the number of the interval in which the first transmission was received.

LocalVars

mode : *Modes*
interval, first : *Intervals*
...

The passage of time is represented by a specification variable holding the current moment of absolute (master clock [1, §4.8.1]) time.

<i>AbsTime</i> <i>now : Times</i>

In the specification below this variable represents the absolute moment at which each of the state changes defined by operation schemata “occurs” (i.e., the time at which the effects of a state change become visible).

3.2.3 Invariants

The specification expresses a number of important properties about the program and its environment as invariants on the above variables. Firstly they are gathered together using Z schema “inclusion”, which allows us to textually include a previously defined schema by merely including its name:

<i>State</i> <i>Inputs</i> <i>Outputs</i> <i>Audio</i> <i>Properties</i> <i>LocalVars</i> <i>AbsTime</i> <hr/> ...

The body of this schema consists of several invariants, some of which are illustrated below.

The program has no control over transmission failures, and can merely record those intervals in which no transmission was received, i.e., those intervals not in the domain of *incoming*.

$$\mid \text{failed} = \text{Intervals} \setminus \text{dom } \textit{incoming}$$

Path failures due to two successive transmission failures [1, §C-1.6.2c] are then described by

$$\mid \text{nocomm} = \{i : \text{Intervals} \mid i \geq 2 \wedge (i - 1) \in \text{failed} \wedge (i - 2) \in \text{failed}\}$$

The transmission failure is not diagnosed until the interval *after* the second transmission failure, to allow for the possibility that the second anticipated transmission was delayed, rather than missing.

Path failures due to two successive corrupted messages [1, §C-1.6.2a] are described by

$$\left| \begin{array}{l} toocorrupt = \{i : Intervals \mid i \geq 1 \wedge \\ \quad i \in (badsyntax \cup badprotocol) \wedge \\ \quad (i - 1) \in (badsyntax \cup badprotocol \cup failed)\} \end{array} \right.$$

In this case the failure can be diagnosed as soon as the second transmission is received.

Some incoming transmissions may be delayed,

$$\left| \quad delayed \subset \text{dom } incoming \right.$$

and this is “evidenced” by the appearance of the transmission at the same time as the following (non-delayed, non-failed) transmission [1, §C-1.7].

$$\left| \begin{array}{l} \forall i : Intervals \mid (i \geq 1) \wedge (i \in delayed) \bullet \\ \quad (i - 1) \notin delayed \wedge (i + 1) \in \text{dom } incoming \end{array} \right.$$

This invariant shows how the Z notation allows the range of a variable bound by a quantifier to be restricted both by a type and by a predicate. The symbol ‘•’ separates the binding from the predicate bound.

Those intervals in which the boiler is reported to be unsafe due to the water level being out of range are defined by

$$\left| \begin{array}{l} unsafe = \{i : \text{dom } incoming \mid \\ \quad (incoming(i)(level) < MinLevel) \vee \\ \quad (incoming(i)(level) > MaxLevel)\} \end{array} \right.$$

where the levels defined by constants $MinLevel$ and $MaxLevel$ are those given for safe operation of the boiler by the manufacturer [1, §2.3.1]. Constant $level$ indexes the part of the incoming transmission that holds the current reported water level.

To simplify invariants relating to absolute times, the following function returns the time at which an interval x is defined to begin. This is done by relating it to the number of five second intervals since the first non-delayed transmission j was sent. All time values are expressed using “master” timing. (Interval j denotes the first successful transmission sent from the instrumentation system. This is not necessarily the first transmission *received*, however, to allow for the possibility that the control program is not ready when the instrumentation system is activated.)

$$\left| \begin{array}{l} \overline{startint : Intervals \leftrightarrow Times} \\ \forall x : Interval \mid x \geq j \bullet \\ \quad startint(x) = (x - j) * 5 + whenseen(j) \\ \quad \mathbf{where} \quad j = \min((\text{dom } incoming) \setminus delayed) \end{array} \right.$$

The environmental assumptions that “the instrumentation system will transmit at the beginning of every five second interval” [1, §4.8.1], except for delayed transmissions, which appear five seconds late, are captured as follows.

$$\left| \begin{array}{l} \forall i : (\text{dom } incoming) \setminus delayed \bullet whenseen(i) = startint(i) \\ \forall i : delayed \bullet whenseen(i) = startint(i) + 5 \end{array} \right.$$

At this level of abstraction events are assumed to happen at exactly the specified times. Timing uncertainties are added later as the design progresses.

Invariants such as this describe the anticipated behaviour of the environment, an important consideration in safety-critical work, since they define the limitations under which the system can be expected to behave properly [5]. In the full specification a more extensive specification of the environment may be needed. In particular, a model of the boiler itself (rather than just our interface to it through the instrumentation system) may be necessary to capture boiler dynamics [1, §2.3.1] and thus enable accurate extrapolation of possible boiler behaviour when transmissions fail. Similarly, other hardware components may also need explicit modelling [1, §2.3].

Similar invariants are used to constrain the visible behaviour of the program. Transmissions to the instrumentation system may be sent only within a given window in each interval [1, §4.8.2].

$$\left| \begin{array}{l} \forall i : \text{dom } outgoing \bullet \\ \quad whensent(i) \in ((startint(i) + 1.25) .. (startint(i) + 4.5)) \end{array} \right.$$

When defining timing requirements relating to external events it has been assumed that $whenseen(i)$ and $whensent(i)$ represent the current time (i.e., the value of *now*) when the transmissions for interval i (if any) occurred. Each value of *interval* is associated with a five second period of time. This assumption is formally captured by relating these variables to one another, e.g., for *interval* and *now*,

$$\left| \quad now \in (startint(interval) .. (startint(interval) + 5)) \right.$$

The invariants above express those timing requirements relating to external events. Those based on internal events are expressed directly, using the *now* variable (e.g., see schema *ShutDown* below).

3.2.4 Operations

The need to recognise, and respond to, potentially catastrophic system failures occurs in all system modes [1, §4.7.3]. It is therefore convenient to define a predicate (on variables “included” from schema *State*) that alerts us to the fact that the current *interval* is one in which such a failure can be recognised.

<i>Catastrophe</i>
<i>State</i>
$ \begin{aligned} & (interval \geq first \wedge \\ & \quad ((interval \notin delayed \wedge interval \in unsafe) \vee \\ & \quad \quad ((interval - 1) \in delayed \wedge (interval - 1) \in unsafe))) \vee \\ & (interval \geq (first + 1) \wedge interval \in toocorrupt) \vee \\ & (interval \geq (first + 2) \wedge interval \in nocomm) \end{aligned} $

The specification variable *first* is intended to record the first interval in which a valid transmission is received. It is set by the *SystemTest* operation and is used to model the minimum time following the initialisation tests at which a catastrophe can be recognised.

Motivations for this formalisation of *Catastrophe* are as follows. An incoming transmission that indicates the boiler water level is unsafe must trigger an immediate response, unless the transmission was delayed, in which case the error is to be diagnosed in the next interval. If there are too many corrupted transmissions, this cannot be diagnosed until two such transmissions have been received [1, §C-1.6.2a]. A failure caused by loss of communication cannot be diagnosed until three intervals have passed (to allow for the possibility that a transmission was lost in the first interval, but the transmission expected in the second interval was merely delayed, not lost).

This predicate then appears in the definitions of distinct operations for each mode, i.e., *SelfTest*, *SystemTest*, *NormalOp*, *DegradedOp* and *EmergencyOp*. For instance, the system behaviour for a given interval in “normal” operating mode [1, §4.4] is defined using logical schema operators as

$$\begin{aligned}
NormalOp \cong & (\neg Catastrophe \wedge Respond) \vee \\
& (Catastrophe \wedge ShutDown)
\end{aligned}$$

In other words, if the current *interval* is not one defined as a *Catastrophe* the program should behave as defined by schema *Respond*, otherwise it should behave like schema *ShutDown*.

Respond defines the action of the program during a particular interval in normal operating mode. Typically this involves sending a transmission, part of which describes whether the pumps should be switched on or off, depending on the current boiler content.

<i>Respond</i>
$\Delta State$
$\Xi Inputs$
$\Xi Audio$
$\Xi Properties$
$mode = NormalMode$
$dom\ outgoing' = dom\ outgoing \cup \{interval\}$
$\{interval\} \triangleleft outgoing' = outgoing$
$content \in TooLow \Rightarrow outgoing'(interval)(pumps) = On$
$content \in TooHigh \Rightarrow outgoing'(interval)(pumps) = Off$
\dots
$mode' = mode$
$now' = now + 5$

Variables in schemata preceded by Δ are, by Z convention, capable of being modified by the operation. However those variables listed in schemata preceded by Ξ are assumed to remain unchanged. Primed variables (e.g., “*outgoing'*”) represent components of the “after” state. Thus, *outgoing* is left unchanged except a new value is added corresponding to this interval; the domain subtraction operator \triangleleft is used to remove the entry corresponding to *interval* from the function *outgoing'*, leaving a function identical to the original *outgoing*. The part of the transmission associated with water pump action depends on the current boiler *content*. (In the core example *content* is merely the last value successfully reported by the instrumentation system; in the full system it should be computed based on the latest information received and an extrapolation of anticipated boiler behaviour during the interval.) These actions should ideally occupy exactly 5 seconds, as measured by *now*.

When a catastrophic failure occurs during normal operation, however, the boiler must be shut down [1, §4.4.2a]. In this situation there are three actions to be performed. Within 5 seconds the operator must be alerted [1, §4.7.1], which includes activating an audio output device. This audio warning can be switched off after a further 20 seconds. The system can then revert to self-test mode, after at least 30 seconds [1, §4.7.2]. These requirements are captured as follows, using the Z schema composition operator ‘ \circledast ’ which specifies the composition of the behaviours specified by its arguments.

$$ShutDown \cong (SirenOn \circledast SirenOff \circledast Recover) \mid (now' \geq now + 30)$$

The 30 second requirement has been modelled by using the Z operator ‘ \mid ’ to conjoin a constraint on *now* to the overall composition of *SirenOn*, *SirenOff* and *Recover*. These components are shown below.

<i>SirenOn</i>
$\Delta State$ $\Xi Inputs$ $\Xi Outputs$ $\Xi Properties$
$now' \in (now .. (now + 5))$ $siren' = siren \cup \{now' \mapsto On\}$ \dots $mode' = ShutDownMode$

<i>SirenOff</i>
$\Delta State$ $\Xi Inputs$ $\Xi Outputs$ $\Xi Properties$
$mode = ShutDownMode$ $now' \geq now + 20$ $siren' = siren \cup \{now' \mapsto Off\}$ \dots $mode' = mode$

<i>Recover</i>
$\Delta State$ $\Xi Inputs$ $\Xi Outputs$ $\Xi Properties$
$mode = ShutDownMode$ \dots $now' \geq now$ $mode' = SelfTestMode$

SirenOn and *SirenOff* illustrate how timing requirements can be directly modelled using the *now* variable.

3.3 Design

A skeletal program design was undertaken using a top-down development methodology. Care was taken to ensure that timing obligations were recorded throughout the design process.

The approach used was based on a real-time extension of a “refinement calculus” [4], with some notational modification to reflect recently published work on algorithm design using the Z notation [21] and our use of C as the target language.

A refinement calculus [18] is a formal methodology for program development from formal specifications. It is based on a number of rules that allow statements in the specification to be re-expressed in a lower-level specification or in the programming language. The rules, denoted

$$Spec \sqsubseteq Ref$$

guarantee that at each step the refinement *Ref* is correct relative to the specified functional behaviour *Spec*; thus program verification is performed simultaneously with program development. Furthermore, application of the rules provides a trace of software development that can aid in later system maintenance. During intermediate phases of program development the “code” *Ref* may be a mixture of specification and implementation statements [18].

Each of the rules is accompanied by a number of proof obligations (sometimes implicit). For the rule to be validly applied, these obligations must be discharged. For purely functional (procedural) requirements this can be done as each rule is applied. Our “real-time” rules [4] account for time by extending these proof obligations to encompass requirements relating to code execution times. These extensions make use of variables in the specification which, by convention, denote the passage of real time. In the boiler specification above it is assumed that the value of *now* represents the time (in terms of the “master” clock [1, §4.8.1]) at which each state change is expected to take place. It is also assumed that *whenseen* and *whensent* record the times at which transmissions are received and sent and that *siren* records the times at which the audio device was switched on and off. Variables relating solely to timing requirements do not always contribute directly to the code produced; their rôle is often to define proof obligations only.

It may be impossible to immediately discharge proof obligations relating to time because this usually requires an intimate knowledge of the target compiler, architecture and operating system. The obligations are therefore recorded during program development, and discharged in a separate verification phase when such information becomes available (see section 3.4).

The following sections show the key steps in producing the code relevant to the safety property described in section 3.1.

3.3.1 Top-level iteration

A Z specification is conventionally interpreted as specifying a computation which starts from a state satisfying a special *Init* schema and then repeatedly performs specified operations, chosen so that each operation occurs in a state which satisfies

the operation’s pre-condition, until no such operation exists. The pre-condition of an operation is inferred from the predicate part of the schema defining the operation [21, §5.4.4]. Let Ops be a choice between all “top-level” operations in the boiler specification, i.e., those not used in defining any other schemata.

$$Ops \hat{=} SelfTest \vee SystemTest \vee \\ NormalOp \vee DegradedOp \vee EmergencyOp$$

$ShutDown$ is not included since it appears in the definition of each of these schemata. Any such Z specification can then be refined into skeletal code [4] as follows. (The reason for using an indirect form of loop is made apparent in section 3.3.2.)

```
“core specification”  $\sqsubseteq$ 
main() {
  Init;
  do {
    if (pre  $Ops_1$ )
       $Ops_2$ 
    else
      break;
  } while(1);
}
```

with proof obligations

$$Ops \sqsubseteq Ops_1 \circ WhileTime \\ Ops_1 \sqsubseteq IfTime \circ Ops_2 \circ ElseTime$$

where

$$WhileTime \hat{=} [\Delta State; \exists Proc \mid now' - now \in T_c(1)] \\ IfTime \hat{=} [\Delta State; \exists Proc \mid now' - now \in T_c(pre\ Ops_1)] \\ ElseTime \hat{=} [\Delta State; \exists Proc \mid now' - now \in T(else)]$$

and

$$Proc \hat{=} Inputs \wedge Outputs \wedge Audio \wedge Properties \wedge LocalVars$$

$Proc$ consists of all specification variables that are not used to effect the passage of time. $WhileTime$, $IfTime$ and $ElseTime$ are schemata expressed using the concise Z horizontal notation – semi-colons serve to separate “lines”. Thus

‘ $\Delta State; \Xi Proc$ ’ states that all variables remain unchanged except those in schema $AbsTime$, i.e., *now*.

We now discuss some aspects of this refinement step. Ops is re-expressed twice, as Ops_1 and Ops_2 , firstly to account for the small amount of time required to iterate, and then to account for the time required to evaluate the condition. These are examples of how subscripted numbers are used throughout this presentation to distinguish schemata which define identical functional behaviour, but with different execution time requirements. Because Ops_1 and Ops_2 differ from Ops only in their requirement for updating *now* we omit their full description. The schemata $WhileTime$ and $IfTime$ change only the *now* variable. Thus, in showing that Ops is refined by $Ops_1 \circledast WhileTime$, we must establish that Ops_1 performs the same functions as Ops , but accounting for the overhead of iterating. Similarly, Ops_2 must be shown to be functionally the same as Ops_1 , and it must account for the overheads of evaluating the *if* expression and branching.

Functions T and T_c return sets of possible execution times for the given construct in seconds of “master” time. (“Set additions”, denoted \boxplus , operate pointwise on each of the two sets of times [4].) For instance, $T(\mathbf{else})$ is the set of possible times required to branch at the end of the alternative. The expression $T_c(\text{pre } Ops_1)$ is the set of possible times required to evaluate the boolean condition associated with the pre-condition of Ops_1 and branch to the correct location. Expression $T_c(1)$ represents the set of possible times required to evaluate the constant expression 1 and branch. Further detail about the issues involved in assessing these functions is given in section 3.4.

In the general case, Wordsworth [21, §7.3] notes that to show that a refinement Ref is a correct implementation of a specification $Spec$ we must prove that any circumstance acceptable to $Spec$ is acceptable to Ref , i.e., in Wordsworth’s notation,

$$\text{pre } Spec \vdash \text{pre } Ref$$

and that in any circumstance acceptable to $Spec$, the behaviour of Ref is allowed by $Spec$, i.e.,

$$(\text{pre } Spec) \wedge Ref \vdash Spec$$

Thus, to satisfy the first of the two proof obligations above, we must show that

$$\begin{aligned} &\text{pre } Ops \vdash \text{pre}(Ops_1 \circledast WhileTime) \\ &(\text{pre } Ops) \wedge (Ops_1 \circledast WhileTime) \vdash Ops \end{aligned}$$

and similarly for the second. We cannot do this completely, however, without knowing what values are returned by functions T and T_c . These are dependent upon the target environment. We therefore record the timing obligations here, and leave their satisfaction to the verification phase described in section 3.4.

3.3.2 Evaluate top-level condition

The condition in the `if` statement above, `pre Ops`, is quite complex, consisting of the disjunction of the pre-conditions of several operations. Furthermore, when expanded, it can be seen that the condition includes sub-expressions such as `incoming(interval)` which is intended to access a new value from the *incoming* data stream. Such a complicated condition would lead to unreadable code in the target language. In these circumstances it is better to re-express the `if` statement so that it is preceded by sequential code that evaluates the necessary data [21, 15], i.e.,

```
A;
if (E)
    Ops2
else
    break;
```

where *A* is code that performs work necessary to make the condition easily evaluable in the target language, and *E* is the simplified condition equivalent to `pre Ops1`.

In the *Z* specification, each of the top-level operations increments *interval*, either directly, or indirectly by incrementing *now* by 5. This indicates that the operation is intended to occupy one master timing interval, of duration 5 seconds. The next time a top-level operation is performed the expression `incoming(interval)`, which occurs frequently in the pre-condition, therefore refers to a “new” value in *incoming*. By convention, we assume that this is equivalent to performing a “read” from the concrete data stream, i.e., *A* must first attempt to read a transmission from the instrumentation system, if the expression is to be evaluated. This involves allowing for failed or delayed transmissions, as defined by the invariants in section 3.2.3.

Furthermore, `pre Ops1` uses the *Catastrophe* predicate (for example, see the definition of *NormalOp* in section 3.2.4), and hence must know not only the status of the transmission intended for the current interval, but also that of the previous and penultimate intervals. The definition of *SelfTest* also requires that special action be taken to synchronise with the first incoming transmission in that mode.

Finally, the read operations must be performed at the correct time. As shown by, e.g., *Respond*, this is in intervals of five seconds.

The following skeletal code is presented as an embodiment of these requirements. In a complete development a data refinement step would be undertaken and appropriate variable and type declarations made [21, §7.6.5]. Here these include `trans`, the transmission received, `this`, `previous` and `penultimate`, the status of the last three transmissions received, `catastrophe`, a boolean variable

which is true if a shutdown is required, `dataready`, a boolean variable which is true if an incoming character is available on the RS232 port, and `interval`, the current interval number as indicated in synchronisation messages [1, §B-3.1.1.3]. The full process for formally developing this code would consist of iteratively introducing each of the new variables, postulating code for evaluating them, and proving the validity of the new code [21, §7.6.6].

```

A ⊆
if (first != 0) {
  penultimate = previous;
  previous = this;
  CheckDataReady;
  if (dataready) {
    ReadTransmission;
    if (previous == FAILEDORDELAYED) {
      ...
    }
  }
  else {
    if (trans[SYNC] != interval)
      this = CORRUPT;
    else
      this = OK;
    catastrophe = (((this == CORRUPT) &&
                      ((previous == CORRUPT) ||
                       (previous == FAILED))) ||
                    ((this != CORRUPT) &&
                     ((trans[LEVEL] < MINLEVEL) ||
                      (trans[LEVEL] > MAXLEVEL))));
  }
  else
    this = FAILEDORDELAYED
}
else {
  ...
}

```

When a single transmission is received it is first checked for corruption (as required by the invariants in section 3.2.3). In the core example this merely involves checking that the synchronisation number is correct. Then the boolean variable `catastrophe` is set to true if the transmission received is corrupt and the preceding one was corrupt or failed, or the boiler water level is reported to be out

of range. Again, this test is simpler than would be the case for the full example.

Action *CheckDataReady* determines whether or not an STX character [1, §A-5] is available on the RS232 datalink. If so, action *ReadTransmission* reads the incoming transmission into variable `trans`. These low-level actions are described fully in section 3.3.9.

Observe that, in this implementation, there are several actions to be undertaken before the STX character is read (in *CheckDataReady*). However, by definition, the arrival of the STX character denotes the beginning of a master timing interval [1, §4.8.1]. Care must be taken to recognise that the body of the main `do-while` loop does not correspond to a master interval. Each interval begins in *CheckDataReady* and involves the code up to the next occurrence of *CheckDataReady* in the *next* iteration. (In the formal specification, operations such as *NormalOp* are specified to occur in a single master interval, and are thus implicitly synchronised with the master clock. The refined code needs to take action to achieve synchronisation with the master clock.)

Expression E can now be easily expressed in C. After simplification it becomes merely

```
mode != SHUTDOWNMODE
```

since all possible transmission eventualities are treated in each “top-level” mode. Variable `mode` is initially set by the *Init* operation and is maintained by each operation to reflect the current system status. Ironically, simplification has removed all mention of the other variables set by A (this outcome is not obvious during application of the refinement rules, however). Nevertheless, the effort was not in vain as these variables are all needed in the steps below which treat the disjuncts in pre *Ops* separately.

It is important to note that the execution time for the `if` condition, i.e., $T_c(\text{pre } Ops_1)$, is given by $T(A) \boxplus T_c(E)$ because the overhead associated with evaluating all of the variables is due to the need to evaluate the original condition `pre Ops1`, and hence this expression will be used in *IfTime₁* above.

The execution time requirements for concrete code fragments can be expressed using the rules in section 3.4. For instance,

$$\begin{aligned} & T_c(E) \\ &= T_c(\text{mode} \neq \text{SHUTDOWNMODE}) \\ &= T(\text{mode}) \boxplus T_c(\neq, \text{type}(\text{mode})) \boxplus T(\text{SHUTDOWNMODE}) \end{aligned}$$

Similarly, for an assignment statement, e.g.,

$$\begin{aligned} & T(\text{penultimate} = \text{previous}) \\ &= T(=, \text{type}(\text{penultimate}), \text{extent}(\text{penultimate})) \boxplus T(\text{previous}) \end{aligned}$$

Together with the rules for sequential composition and choice we can express $T(A)$ entirely in terms of such primitives.

3.3.3 Refine operations

As defined above, Ops consists of a disjunction of several operation schemata. This commonly-occurring specification structure is routinely refinable to a set of nested `if` statements [21, §7.5.4]. Here and below we show only the overall structure of new code generated. Expansions of operations with subscripts, denoting operations which are functionally unchanged but progressively more constrained in their timing behaviour, are omitted.

```

Ops2 ⊆
if (pre SelfTest2)
  SelfTest3;
else
  if (pre SystemTest3)
    SystemTest4;
  else
    if (pre NormalOp3)
      NormalOp4;
    else
      if (pre DegradedOp3)
        DegradedOp4;
      else
        if (pre EmergencyOp3)
          EmergencyOp4;

```

where, for each component, we must show that it accounts for the time taken to evaluate all preceding conditions. For example, we are obliged to prove that

$$\begin{aligned}
NormalOp_2 &\sqsubseteq IfTime_2 \circ NormalOp_3 \\
NormalOp_3 &\sqsubseteq IfTime_3 \circ NormalOp_4 \circ ElseTime
\end{aligned}$$

where

$$\begin{aligned}
IfTime_2 &\hat{=} [\Delta State; \Xi Proc \mid \\
&\quad now' - now \in (\top_c(\text{pre } SelfTest_2) \boxplus \top_c(\text{pre } SystemTest_3))] \\
IfTime_3 &\hat{=} [\Delta State; \Xi Proc \mid \\
&\quad now' - now \in \top_c(\text{pre } NormalOp_3)]
\end{aligned}$$

3.3.4 Normal operation condition

The conditions appearing in the `if` statements above are expressed easily given the pre-conditions already established by A . For instance, with simplification,

$$\text{pre } NormalOp_3 \sqsubseteq (\text{mode} == \text{NORMALMODE})$$

where

$$\mathbb{T}_c(\text{pre } NormalOp_3) = \mathbb{T}(\text{mode}) \boxplus \mathbb{T}_c(==, \text{type}(\text{mode})) \boxplus \mathbb{T}(\text{NORMALMODE})$$

The other conditions in section 3.3.3 are equally simple.

3.3.5 Refine normal operation

Using the specification of the normal operating mode from section 3.2.4, we can again re-express disjunction as follows,

$$\begin{aligned} NormalOp_4 &\sqsubseteq \\ &\text{if } (\text{pre}(\neg Catastrophe \wedge Respond)_4) \\ &\quad (\neg Catastrophe \wedge Respond)_5; \\ &\text{else} \\ &\quad \text{if } (\text{pre}(Catastrophe \wedge ShutDown)_5) \\ &\quad \quad (Catastrophe \wedge ShutDown)_6; \end{aligned}$$

providing that, for the second alternative

$$\begin{aligned} (Catastrophe \wedge ShutDown)_4 &\sqsubseteq \text{IfTime}_4 \circledast (Catastrophe \wedge ShutDown)_5 \\ (Catastrophe \wedge ShutDown)_5 &\sqsubseteq \text{IfTime}_5 \circledast (Catastrophe \wedge ShutDown)_6 \circledast \\ &\quad \text{ElseTime} \end{aligned}$$

where

$$\begin{aligned} \text{IfTime}_4 &\hat{=} [\Delta State; \Xi Proc \mid \\ &\quad \text{now}' - \text{now} \in \mathbb{T}_c(\text{pre}(\neg Catastrophe \wedge Respond)_4)] \\ \text{IfTime}_5 &\hat{=} [\Delta State; \Xi Proc \mid \\ &\quad \text{now}' - \text{now} \in \mathbb{T}_c(\text{pre}(Catastrophe \wedge ShutDown)_5)] \end{aligned}$$

and similarly for the first alternative.

3.3.6 Normal operation conditions

The two conditions in section 3.3.5 are refined easily, given the pre-conditions established by A .

$$\begin{aligned} \text{pre}(\neg Catastrophe \wedge Respond)_4 &\sqsubseteq \\ &\quad \text{mode} == \text{NORMALMODE} \ \&\& \ !catastrophe \\ \text{pre}(Catastrophe \wedge ShutDown)_5 &\sqsubseteq \text{catastrophe} \end{aligned}$$

where the execution times can be expressed as before.

3.3.7 Refine shutdown operation

A development rule that allows time-dependent Z schema composition operations to be re-expressed as sequential compositions [4] allows us to refine the shutdown actions as follows.

$$\begin{aligned} & (Catastrophe \wedge ShutDown)_6 \sqsubseteq \\ & \quad SirenOn_6; \\ & \quad SirenOff_6; \\ & \quad Recover_6; \end{aligned}$$

where, among other requirements [21, §7.6.1], there is a timing requirement which can be expressed as follows.

$$T(SirenOn_6) \boxplus T(SirenOff_6) \boxplus T(Recover_6) \subseteq \{a : Times \mid a \geq X\}$$

The value X is the original timing requirement of 30 seconds, less the execution time for all of the preceding code from sections 3.3.1 through to 3.3.6 (due to the repeated re-expression of $(Catastrophe \wedge ShutDown)$ at each preceding step).

3.3.8 Refine siren operation

The function of switching on the siren can be refined to code using the refinement laws for introducing sequential composition and assignment.

$$\begin{aligned} & SirenOn_6 \sqsubseteq \\ & \quad \text{mode} = \text{SHUTDOWNMODE}; \\ & \quad *((\text{char } *) \text{SIREN}) = \text{ON}; \\ & \quad \dots \end{aligned}$$

where

$$\begin{aligned} T(SirenOn_6) &= T(\text{mode} = \text{SHUTDOWNMODE}) \boxplus \\ & \quad T(*((\text{char } *) \text{SIREN}) = \text{ON}) \boxplus \dots \\ T(\text{mode} = \text{SHUTDOWNMODE}) &= T(=, \text{type}(\text{mode}), \text{extent}(\text{mode})) \boxplus \\ & \quad T(\text{SHUTDOWNMODE}) \end{aligned}$$

and so on. The hardware location denoted by constant `SIREN` maps onto the audio output device, and transmitting the character denoted by constant `ON` is assumed to switch the audio device on.

The safety pre-requisite outlined in section 3.1 can now be seen to be that the execution time for all of the code outlined in sections 3.3.2 to 3.3.8 is less than 5 seconds. See Appendix B.

3.3.9 Local clock

The requirements specification introduces the pragmatic need for the control system to use a source of timing information distinct from the (possibly unreliable) external environment [1, §4.8.4]. A decision was made early in our design to use this “local clock” to time instrumentation system intervals. We assume that the local clock can be assigned a time by writing an appropriate value to the hardware-specific location `CLOCKIN`, and that the current time (as an integer representing the elapsed number of milliseconds) can be accessed by reading from the location `CLOCKOUT`.

We can then express the need to look for each STX character at five second intervals as follows.

```
CheckDataReady  $\sqsubseteq$ 
waiting = 1;
while(waiting) {
    now = *((int *) CLOCKOUT);
    waiting = now < FiveSeconds;
};
ExpectFirstChar
```

Unfortunately, we cannot assume that the hardware clock runs at precisely the same rate as the master clock. Inevitably we must account for some deviation. We assume that constant `ERROR` is the maximum number of milliseconds deviation of our local clock in five seconds of master clock time. Then let

```
FiveSeconds = 5000-ERROR
```

and we must continue looking until we are certain that the deadline has expired, i.e.,

```
ExpectFirstChar  $\sqsubseteq$ 
looking = 1;
while (looking) {
    temp = *((char *) LSR);
    dataready = (temp & DRMASK);
    now = *((int *) CLOCKOUT);
    looking = !dataready && (now <= (5000+ERROR));
}
```

where `LSR` is the location of the hardware interface to the RS-232C line status register, and constant `DRMASK` masks off all bits other than the data ready bit. The temporary variable `temp` is used when reading incoming data to reduce the complexities of dealing with side-effecting expressions.

The act of reading the transmitted characters is then defined as follows, with an appropriate clock reset to allow for the next interval.

```

ReadTransmission  $\sqsubseteq$ 
*((int *) CLOCKIN) = 0;
temp = *((char *) RBR);
trans[0] = temp & DATAMASK;
for (i=1; i<TRANSSIZE; i++) {
    dataready = 0;
    while (!dataready) {
        temp = *((char *) LSR);
        dataready = temp & DRMASK;
    };
    temp = *((char *) RBR);
    trans[i] = temp & DATAMASK;
}

```

Constant RBR is the address of the RS-232C receive buffer register. (Together, LSR and RBR are the concrete representation of specification variable *incoming*.) The core example assumes that transmissions are all of a fixed size, represented here by constant TRANSSIZE, and that if an STX character arrives safely then the full transmission will be successfully received.

The execution time for all this code is accounted for in $T(A)$.

3.3.10 Discussion

Notice that the nested if statement in section 3.3.3 makes the evaluation of condition E in section 3.3.2 redundant. Similarly, the first condition in section 3.3.6 duplicates the test for the current mode already made in section 3.3.4. Naïve applications of refinement laws often lead to such redundancy, which can then be manually optimised. When developing programs with timing requirements, however, great care must be taken to reflect these optimisations in the timing obligations. Appendix B lists those statements for the situation of interest in this case study, but optimisations have been avoided (apart from the expression optimisations already noted above) so that the timing obligations do not have to be rewritten.

It is obvious that manual application of a refinement calculus requires a great deal of discipline from the programmer. The work of expressing and checking proof obligations is repetitive and tedious, and hence error-prone. In this case study we did not have time to formally perform all refinement steps, particularly those involved with input processing. For instance, the code in section 3.3.9 was entirely handwritten, with no formal refinement. We believe that all of the code

generation steps *can* be formalised, but clearly tools are required to make the task manageable. Some such tools already exist [8], but none yet incorporate timing considerations.

3.4 Verification of generated timing requirements

This section outlines an approach to formally verifying the timing obligations generated during the development steps of the previous sections.

The basis for the timing verification is an extension of previous work [12] which formally specified and verified a program for a simplified version of the RS232 software repeater problem [13, 11]. The program was written in assembly code for a Motorola MC6809 8-bit microprocessor. Similar methods are currently being used to formally specify aspects of a MIPS RISC processor and a scheduler for it and to formally verify the scheduler implementation (written in MIPS assembler) against these specifications [20].

The extension proposed is to use a theory which permits verification of software at a higher level, using a subset of C. To demonstrate how this can be done, we outline how a theory can be defined which specifies the real-time behaviours of a C subset for a specific compiler and a specific target processor. To build directly on previous work, we have used the MC6809 as the target processor. This is an 8-bit processor which can run at a maximum clock rate of 4 MHz. The compiler used is a C compiler for the MC6809, run in non-optimising mode. (This compiler traces its ancestry to the Unix portable C compiler).

The theory sketched can support verification of both functional and timing properties, and provides a possible foundation for the refinement rules used in the top-down development. However, as much as possible of the functional verification is intended to be encapsulated in the use of the refinement rules. The final timing verification phase discharges the timing requirements generated in the top-down development. Note that because this timing verification phase works on the complete code generated, more precise timing bounds are obtainable than during top-down development.

The theory we sketch is suitable for verification of non-interruptible C code which may interact with external processes through shared memory locations. The theory does not support interrupt-driven code, or software time-slicing. Our description of the real-time code behaviour is by no means complete. We have considered essentially only those C language features required to demonstrate the feasibility of verifying the timing properties of the code in Appendix B, under the assumed conditions. However, we believe the approach outlined here is extendable to a much larger C subset than we have considered.

3.4.1 The theory formalism

The formalism used is functional logic and set theory [19, 9], as used in the RS232 study. Functional logic extends classical logic to support reasoning with implicitly parameterised expressions and to support modelling of partial functions. Functional set theory is a theory of implicitly parameterised sets, based on functional logic; it parallels classical set theory.

In effect, functional logic provides an adaptable modal reasoning facility. An assertion, or other term, that depends on an implicit parameter is interpreted as a function with domain and range a set of implicit parameter values. Such a function is called an action.

The logic includes forward function composition, denoted ‘;’. The expression $x; y$ is evaluated with respect to an implicit parameter i as follows. Let \bar{x} be the function on implicit parameters which x denotes and similarly let \bar{y} be the function which y denotes. Then for each implicit parameter i , $(\overline{x; y})(i) = \bar{y}(\bar{x}(i))$. Intuitively, y is here evaluated with respect to an implicit parameter delivered by x .

The set of possible implicit parameters includes a value $?$, which represents undefinedness. All other implicit parameters are termed ‘normal’. An action which yields the same value at all normal implicit parameters is termed ‘absolute’. The predicate $is_abs(A)$ is true if and only if A denotes an absolute action. The predicate $defined(A)$ is true at an implicit parameter if A at that implicit parameter is not equal to $?$.

Functional set theory considers interpretations of functional logic in which the implicit parameters are the objects of some model of set theory, plus a separate object $?$. The only non-logical primitive is a two-place membership operator \in , whose semantics is as follows: for all normal implicit parameters i ,

$$(\overline{X \in Y})(i) = \overline{X}(i) \in \overline{Y}(i).$$

Note that the occurrence of \in on the left is functional set membership, defined on actions, while the \in on the right is classical set membership on implicit parameter values.

In effect, much of functional set theory can be understood as performing classical set theory in an indexed fashion for each implicit parameter. Many notations in functional set theory carry over from classical set theory. We do not attempt to enumerate them all here.

3.4.2 Modelling real-time behaviour

We use a descriptive approach which considers complete traces representing the activities (over all time) of a run of the modelled system. Reasoning is about a typical run. It utilises constraints on possible runs imposed by the code executed and by the target hardware, including devices with which the processor interacts.

Assertions about system behaviour are implicitly parameterised by ‘configurations’, where a configuration consists of a complete trace of a computation and the current time.

Times are modelled by natural numbers, and the unit of time is the clock cycle of the target processor. A trace is a function from all times to states. Thus, a trace represents a state for each processor clock cycle.

A state consists of an environment, a store and an activity-state.

The environment is intended to model certain aspects of a C program state. To handle nested scoping, an environment consists of a sequence of level-environments, where a level-environment consists of a mapping from program identifiers to locations, a mapping from program identifiers to their extents (for example, static, auto), and a mapping from program identifiers to types. Locations are classified as either program locations or shared locations. Program locations are those locations entirely under the control of the program and shared locations are those that other processes may affect, for example device registers.

The store models the current state of the locations, including both locations of program variables and non-program locations that the program may access. The store is a mapping from locations to values.

The activity state records current activity which can affect non-program locations. The motivation for the activity state is to support modelling of low-level interaction with hardware devices, as occurs in embedded programs. This is discussed further in section 3.4.3.

The activity state consists of two relations, the read relation and the write relation. Each element of the read relation is a pair consisting of a process identifier P and a location L . Such an element represents the activity of process P reading location L . Each element of the write relation is a triple consisting of a process P , a value V and a location L . Such an element represents the activity of process P writing the value V to the location L . We will use the process name *cpu* to represent activity of the CPU. We require that the CPU can be involved in at most one activity in each activity-state.

Table 1 lists a number of formal notations and their informal semantics. Formal definitions are given in [13]. In addition, note that the function l maps identifiers to locations and the postfix operator $\hat{}$ maps a location to its content.

3.4.3 Modelling hardware devices

In the present case study there are a number of devices that the processor interacts with: a RS232 communications chip, a local clock and a siren.

Here we axiomatise some properties of these devices that are relevant to the example code in Appendix B.

First, the RS232 communications chip. Again, to build on previous work, we assume the use of the Asynchronous Communications Element (ACE) [2], as in

Function	Description
$is_time(V)$	V is a valid time (i.e., V in $times$).
$time$	The current time.
$at(T)$	A function that changes the current time to be T .
$after(T)$	Equals $at(time + T)$.
$during(T1, T2, P)$	true if $T1$ and $T2$ are times and the predicate P is true at all times from $T1$ to $T2$, inclusive; false otherwise.
$until(T, P)$	Equals $during(time, T, P)$.
$read(P, L)$	true if current activity state records that process P reads from location L .
$write(P, V, L)$	true if current activity state records that process P writes V to location L .
$write_to(L)$	true if current activity state records that some process writes to location L .
$write_by(P, L)$	true if current activity state records that process P writes to location L .

Table 1: Some formal notations used with configurations

the RS232 study. The ACE has a number of CPU accessible registers, amongst them the line status register and the receiver buffer register. We model them as single byte locations lsr and rbr . The ACE processes an incoming bit stream and assembles the data into bytes, accessible to the CPU in the receiver buffer register. We model the event of a byte becoming available by a write by the ACE to rbr . When a byte is available, the data ready bit of lsr is set to 1. The same bit is reset to 0 when the CPU reads the receiver buffer register, and can only be reset in that way.

The following are formal statements of relevant properties of the ACE. The formalisations use the predicate hw ; intuitively, this predicate is true of all implicit parameters that are configurations whose trace component represents a feasible run of the modelled hardware.

We use the predicate dr_set which is true if the first bit of the line status register is set. To define this, first define a right shift operator as follows

$$\vdash V \gg N = V \text{ div } (2 ** N).$$

Now define a predicate $bitset$ such that $bitset(V, N)$ is true if the N -th bit of V is set.

$$\vdash bitset(V, N) = ((V \gg N) \text{ mod } 2 = 1).$$

Now dr_set can be defined by

$$\vdash dr_set = bitset(lsr, 1).$$

The relevant properties may be axiomatised as follows.

$$\begin{aligned} &\vdash (hw \wedge \neg dr_set \wedge (at(time + 1); dr_set)) \\ &\quad \Rightarrow write_by(ace, rbr) \\ &\vdash (hw \wedge write_by(ace, rbr)) \\ &\quad \Rightarrow (at(time + 1); dr_set) \\ &\vdash (hw \wedge dr_set \wedge (at(time + 1); \neg dr_set)) \\ &\quad \Rightarrow read(cpu, rbr) \\ &\vdash (hw \wedge read(cpu, rbr)) \\ &\quad \Rightarrow (at(time + 1); \neg dr_set) \\ &\vdash (hw \wedge (at(T_1); write_by(ace, rbr)) \wedge (at(T_2); write_by(ace, rbr))) \\ &\quad \wedge (T_1 > T_2)) \\ &\quad \Rightarrow ((T_1 - T_2) \geq m1) \wedge (T_1 - T_2) \leq m2 \end{aligned}$$

where in the last axiom $m1$ and $m2$ are respectively the minimum and maximum inter-arrival time of incoming bytes, measured in machine cycles.

Concerning the local clock, we assume that the clock can be read by reading from the (16-bit) location $clockout$ and can be loaded with a value by writing to $clockin$. We also assume that if the clock is not reloaded, the clock increments at the rate of 1 per d machine cycles. This implies a tight coupling of the CPU cycle and the local clock, which can be achieved by deriving the local clock time from CPU cycles.

$$\begin{aligned} &\vdash (hw \wedge write(cpu, V, clockin) \wedge (T > time) \\ &\quad \wedge during(time + 1, T - 1, \neg write_by(cpu, clockin))) \\ &\quad \Rightarrow ((at(T); clockout) = (V + (T - time) \text{ div } d)) \end{aligned}$$

3.4.4 Describing real-time code behaviour

Real-time code behaviour is described by constraining the possible traces that are compatible with running the code. This is done by writing assertions of the form

$$\vdash exec(CODE) \Rightarrow PROPERTIES$$

where the predicate $exec(CODE)$ is true intuitively if the trace component of the implicit parameter is compatible with running the designated code from the current time. Note that $exec$ includes the hardware constraints; that is,

$$\vdash exec(CODE) \Rightarrow hw.$$

Code is represented in abstract syntax tree form, using abstract syntax tree constructors. We use typewriter font for such constructors. For example the term $X + Y$ intuitively denotes the abstract syntax tree constructed from subtrees denoted by X and Y and the operator symbol $+$. We also use a constant $+$ to denote the operator symbol itself. (Thus there are two $+$ operators, one binary and the other with zero arity.) For readability, we also sometimes use a distributed (‘mixfix’) notation for operations; for example the term

$$\text{if } E \text{ then } C_1 \text{ else } C_2$$

denotes the syntax tree for the if-then-else construct with subtrees E , C_1 and C_2 .

3.4.5 Expression Evaluation

For simplicity, our description of expression evaluation is restricted to expressions without side effects. We postulate the functional predicate *side_effect_free* which is true of those expressions which are side effect free. We say that expressions without assignments and function calls are ‘simple expressions’. As seen above, accessing certain external devices can have side effects. To ensure absence of side-effects, a method is required for deciding whether an expression causes a reference to such a device location. An approach to achieving this is to write axioms which define the set of locations which a simple expression accesses. The predicate *side_effect_free* is intended to exclude access to side-effecting locations.

In a general, syntax-directed description of expressions, it is not possible to give exact times for expression evaluations. Thus our descriptions give sets of possible times. The actual time taken will be one of the possible times. However, it is possible to augment the general rules with more specific ones for particular expressions, if this is required, as illustrated below.

Table 2 shows the notation used to describe expression timing.

We have distinguished the set of possible times to evaluate an expression E in a control context, denoted $T_c(E)$, from the set of possible times taken to evaluate an expression in a general context, denoted $T(E)$. This distinction is made since the times taken to evaluate the expressions in these different contexts are distinct in C. An example of a ‘control context’ is in the guard of an if or while statement, where only the truth or falsity of an expression is required, rather than a value. We also use the notation $t_c(E)$ ($t(E)$) to denote the actual time taken to evaluate the expression E in a control (respectively, general) context. This value is in general underdetermined, but is always an element of the corresponding set of possible times. That is: $\text{defined}(T(E)) \Rightarrow t(E) \in T(E)$ and $\text{defined}(T_c(E)) \Rightarrow t_c(E) \in T_c(E)$.

The timing functions return times as natural numbers representing numbers of machine cycles. If we assume that each CPU cycle takes *cycle_time* master

Notation	Description
$T(E)$	The set of possible times to evaluate the expression E .
$T_c(E)$	The set of possible times to evaluate the expression E in a control context.
$t(E)$	The actual time to evaluate the expression E in a general context.
$t_c(E)$	The actual time to evaluate the expression E in a control context.
$T_c(Op, Type)$	The set of possible times to evaluate the operator Op on arguments of type $Type$ in a control context.
$T(=, Type, Extent)$	The set of possible times to assign to a variable of type $Type$, and extent $Extent$.
$Tarray_access(Type, Extent)$	The set of possible times to access an array of type $Type$ and extent $Extent$.
$Tconst(Type)$	The set of possible times to evaluate a constant of type $Type$.
$Tconst_c(Type)$	The set of possible times to evaluate a constant of type $Type$ in a control context.
$Tvar(Type, Extent)$	The set of possible times to evaluate a variable of type $Type$ and extent $Extent$.
$Tvar_c(Type, Extent)$	The set of possible times to evaluate a variable of type $Type$ and extent $Extent$ in a control context.

Table 2: Timing notations

clock seconds with an error of plus or minus err , then the master clock timing set $\mathbb{T}(E)$ may be defined in terms of the cycle-time timing set $T(E)$ as follows:

$$\mathbb{T}(E) = \{t \mid \exists t'((t' \in T(E)) \wedge (t' * (cycle_time - err) \leq t) \wedge (t \leq t' * (cycle_time + err)))\}$$

A similar relation exists between the sets $\mathbb{T}_c(E)$ and $T_c(E)$. The description of timing for expression evaluation is guided by the code produced by the MC6809 compiler used. Quantitative timing details for the MC6809 are encapsulated in a number of parameters whose numerical values are given in Appendix C.

The value yielded by an expression E will be denoted $v(E)$.

If X is an expression in the class of integer constant expressions, then its value and timing may be described as follows, where $int(X)$ yields the integer value corresponding to the constant expression X :

$$\begin{aligned}
&\vdash \text{int_const}(X) \Rightarrow (v(X) = \text{int}(X)) \\
&\vdash \text{int_const}(X) \Rightarrow (T(X) = T\text{const}(\text{integer})) \\
&\vdash \text{int_const}(X) \Rightarrow (T_c(X) = T\text{const}_c(\text{integer}))
\end{aligned}$$

If X is a program identifier in the current state then:

$$\begin{aligned}
&\vdash (X \in \text{program_ids}) \Rightarrow (v(X) = l(X) \wedge) \\
&\vdash (X \in \text{program_ids}) \\
&\quad \Rightarrow T(X) = T\text{var}(\text{type}(X), \text{extent}(X))
\end{aligned}$$

Note that a complete description of a C subset would describe the effects of declarations on the environment, which would result in identifiers being added to *program_ids* and corresponding locations to *program_locs*. We do not give such axioms here.

To illustrate how arithmetic expressions may be handled, consider the $+$ operator.

$$\begin{aligned}
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) = \text{integer}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) = \text{integer}) \\
&\quad \wedge ((v(X) + v(Y)) \leq \text{max_int}) \wedge ((v(X) + v(Y)) \geq \text{min_int})) \\
&\quad \Rightarrow v(X + Y) = v(X) + v(Y) \\
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) = \text{integer}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) = \text{integer})) \\
&\quad \Rightarrow T(X + Y) = T(X) \boxplus T(Y) \boxplus T(+, \text{integer})
\end{aligned}$$

To indicate how more specific rules may be used, we give a much more specific timing estimate for a particular case of an addition expression:

$$\begin{aligned}
&\vdash (\text{int_const}(X) \wedge \text{type}(Y) = \text{integer}) \\
&\quad \Rightarrow t(X + Y) \in T(Y) \boxplus \{4\}
\end{aligned}$$

Note that the ‘4’ here is a 6809 specific constant.

To show how logical operators may be handled, consider the $\&\&$ operator. In the axioms below $=_s$ denotes strict equality which returns the exceptional value when either argument is itself undefined, i.e., returns the exceptional value. We also use a strict function if_s : $if_s(A, B, C)$ is undefined if A is undefined, yields B if A is true and C if A is false.

$$\begin{aligned}
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge (\text{type}(Y) \in \text{scalar_types}) \wedge (v(X) = 0)) \\
&\quad \Rightarrow (v(X \ \&\& \ Y) = 0) \\
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) \in \text{scalar_types}) \\
&\quad \wedge (v(X) \neq 0)) \\
&\quad \Rightarrow (v(X \ \&\& \ Y) = \text{if}_s(v(Y) =_s 0, 0, 1)) \\
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge (\text{type}(Y) \in \text{scalar_types}) \wedge (v(X) = 0)) \\
&\quad \Rightarrow (T_c(X \ \&\& \ Y) = T_c(X)) \\
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) \in \text{scalar_types}) \\
&\quad \wedge (v(X) \neq 0)) \\
&\quad \Rightarrow T_c(X \ \&\& \ Y) = T_c(X) \boxplus T_c(Y)
\end{aligned}$$

Note that the timings $T_c(E)$ here include the time for branching out of the expression. When a logical expression is used in a context where its value is required, for example on the right hand side of an assignment statement, a different timing is required. Effectively, the expression must be evaluated as before, but with additional branching and other code to yield the correct result (0 or 1). An appropriate timing axiom for the $\&\&$ operator in such a context is as follows.

$$\begin{aligned}
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge (\text{type}(Y) \in \text{scalar_types}) \wedge (v(X) = 0)) \\
&\quad \Rightarrow (T(X \ \&\& \ Y) = T_c(X \ \&\& \ Y) \boxplus T_{\text{const}}(\text{integer})) \\
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{scalar_types}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) \in \text{scalar_types}) \\
&\quad \wedge (v(X) \neq 0)) \\
&\quad \Rightarrow T(X \ \&\& \ Y) = T_c(X \ \&\& \ Y) \boxplus T_{\text{const}}(\text{integer}) \boxplus T(\text{else})
\end{aligned}$$

To illustrate how relational operators can be described, the following gives axioms for the $!=$ operator.

$$\begin{aligned}
&\vdash (\text{side_effect_free}(X) \wedge (\text{type}(X) \in \text{arith_types}) \\
&\quad \wedge \text{side_effect_free}(Y) \wedge (\text{type}(Y) \in \text{arith_types}) \\
&\quad \wedge (\text{type}(X) = \text{type}(Y))) \\
&\quad \Rightarrow v(X \ != \ Y) = \text{if}_s(v(X) \neq v(Y), 1, 0) \\
&\vdash (\text{side_effect_free}(X) \wedge \text{side_effect_free}(Y) \\
&\quad \wedge (\text{type}(X) = \text{type}(Y))) \\
&\quad \Rightarrow T_c(X \ != \ Y) = T(X) \boxplus T(Y) \boxplus T_c(!=, \text{type}(X))
\end{aligned}$$

3.4.6 Commands

For each type of command, we describe the constraints on the trace component of the implicit parameter which result from executing the command from the current time. The approach followed here in describing the effects of commands is similar in a number of respects to that of [3], particularly in the treatment of while commands.

First we describe the following predicates which are used in the command descriptions.

The predicate $invariant(L, T)$ asserts that the value of the location L is unchanged during the next T time units.

$$\begin{aligned} &\vdash invariant(L, T) \\ &= \forall T_1(((T_1 \geq time) \wedge (T_1 \leq (time + T))) \Rightarrow ((at(T_1); L^\wedge) = L^\wedge)) \end{aligned}$$

Next, the predicate $stable(T)$ asserts that the values of all program locations remain unchanged during the next T time units.

$$\begin{aligned} &\vdash stable(T) \\ &= \\ &\forall L((is_abs(L) \wedge (L \in program_locs)) \Rightarrow invariant(L, T)) \end{aligned}$$

The predicate $update(A, V)$ describes the effect of updating the location A to the value V . The values of other program locations remain unchanged.

$$\begin{aligned} &\vdash update(A, V) \\ &= \\ &(A^\wedge = V) \\ &\wedge \forall L((is_abs(L) \wedge (L \in program_locs) \wedge (L \neq A)) \\ &\quad \Rightarrow (L^\wedge) = (at(time - 1); L^\wedge)) \end{aligned}$$

The next predicate $env_stable(T)$ asserts that the program environment is unchanged during the next T time units. Let env return the environment in the current state. Then

$$\begin{aligned} &\vdash env_stable(T) \\ &= \\ &\forall T_1(((T_1 \geq time) \wedge (T_1 \leq (time + T))) \Rightarrow ((at(T_1); env) = env)) \end{aligned}$$

The assertion $env_stable(T)$ implies, for example, that the sets $program_ids$ and $program_locs$ are unchanged, that the type of each current identifier is unchanged, and so on. We do not formalise all these assertions here.

It is convenient to have an abbreviation for the assertion that the CPU is not engaged in any external activity.

$$\vdash \text{no_cpu_activity} = (\neg \text{write}(\text{cpu}, V, L) \wedge \neg \text{read}(\text{cpu}, L))$$

The *delay* predicate states that nothing happens for a certain time interval.

$$\vdash \text{delay}(T) = (\text{stable}(T) \wedge \text{env_stable}(T) \wedge \text{until}(T, \text{no_cpu_activity}))$$

It is useful to have a concise notation for a time chosen arbitrarily from a set of times. For this we use the function *sm_t*, defined by:

$$\text{sm}_t(T) = \text{sm}_i T_1 (T_1 \in T)$$

where *sm_i* (see [19]) is a functional version of Hilbert's ϵ -symbol.

Finally, we define the predicate *following*:

$$\text{following}(T, P) = (\text{defined}(T) \Rightarrow (\text{after}(T); P))$$

Simple assignment

$$\begin{aligned} &\vdash (\text{exec}(A = B) \wedge (A \in \text{program_ids}) \wedge (\text{type}(A) = \text{type}(B)) \\ &\quad \wedge \text{side_effect_free}(B) \\ &\quad \wedge (T_1 = (T(B) \boxplus T(=, \text{type}(A), \text{extent}(A)))) \\ &\quad \wedge (T_2 = \text{sm}_t(T_1))) \\ &\Rightarrow \\ &\quad ((T(A = B) = T_1) \wedge (t(A = B) = T_2) \\ &\quad \wedge \text{stable}(T_2 - 1) \wedge \text{until}(\text{time} + T_2, \text{no_cpu_activity}) \\ &\quad \wedge \text{env_stable}(T_2) \wedge \text{following}(T_2, \text{update}(l(A), v(B)))) \end{aligned}$$

Note that this axiom describes both the set of possible execution times, that is, $T(A = B)$, and the actual time taken, $t(A = B)$. For conciseness, the remaining example command axioms define only the actual timing. Extending them to define the possible timings is not difficult.

Array assignment

The axiom for assigning to an array is as follows.

$$\begin{aligned} &\vdash (\text{exec}(X[Y] = B) \wedge (X \in \text{program_ids}) \wedge (\text{type}(B) = \text{Ty}) \\ &\quad \wedge (\text{type}(X) = \text{array}(\text{Ty})) \wedge \text{side_effect_free}(B) \wedge \text{side_effect_free}(Y) \\ &\quad \wedge (T_1 = \text{sm}_t(T(B) \boxplus T(Y) \boxplus T(=, \text{array}(\text{Ty}), \text{static}))) \\ &\quad \wedge (v(Y) \geq 0) \wedge (v(Y) < \text{size}(X))) \\ &\Rightarrow \\ &\quad \text{stable}(T_1 - 1) \wedge \text{until}(\text{time} + T_1, \text{no_cpu_activity}) \\ &\quad \wedge \text{following}(T_1, \text{update}(\text{app}(l(X)^\wedge, v(Y)), v(B))) \\ &\quad \wedge (t(X[Y] = B) = T_1) \end{aligned}$$

Note that *app* here is the function which applies a function to its argument. Array values are modelled as functions from indices to locations. The *update* updates the required location.

Device access

To access a device register, the relevant location is cast to a pointer type and then immediately de-referenced. Since accessing such a register may have side-effects, such expressions are not covered by our general rules above. We give axioms which describe the behaviour of single device accesses in a simple assignment.

For example, the behaviour defined by reading an 8-bit location, such as the receive buffer register, is described by the following.

$$\begin{aligned}
& \vdash \text{exec}(A = * \text{cast}(\text{char } *, B)) \wedge (A \in \text{program_ids}) \\
& \quad \wedge \text{integral_const}(B) \wedge (\text{type}(A) = \text{char}) \\
& \quad \wedge T_1 = \text{sm_t}(T\text{var}(\text{char}, \text{static})) \wedge T_2 = \text{sm_t}(T(=, \text{char}, \text{static})) \\
& \quad \wedge (\text{at}(T_1); v(B)^\wedge = V) \\
& \quad \Rightarrow \\
& \quad (\text{until}(\text{time} + T_1 - 1, \text{no_cpu_activity}) \\
& \quad \wedge \text{stable}(T_1 + T_2 - 1) \wedge \text{env_stable}(T_1 + T_2) \\
& \quad \wedge \text{after}(T_1); \text{read}(\text{cpu}, v(B)) \wedge \text{after}(T_1 + T_2); \text{update}(l(A), V) \\
& \quad \wedge \text{during}(\text{time} + T_1 + 1, \text{time} + T_1 + T_2, \text{no_cpu_activity}) \\
& \quad \wedge t(A = * \text{cast}(\text{char } *, B)) = (T_1 + T_2))
\end{aligned}$$

Command sequencing

The following axiom describes the behaviour of a sequence of two commands.

$$\begin{aligned}
& \vdash \text{exec}(C1; C2) \\
& \quad \Rightarrow \\
& \quad \text{exec}(C1) \wedge \text{following}(t(C1), \text{exec}(C2)) \\
& \quad \wedge t(C1; C2) = t(C1) + \text{after}(t(C1)); t(C2))
\end{aligned}$$

Selection commands

The following describes the behaviour of an if command.

$$\begin{aligned}
& \vdash (\text{exec}(\text{if } E \text{ then } C1) \wedge \text{side_effect_free}(E)) \\
& \quad \Rightarrow \\
& \quad \text{stable}(t_c(E)) \wedge \text{until}(t_c(E), \text{no_cpu_activity}) \\
& \quad \wedge ((v(E) \neq 0) \Rightarrow \text{following}(t_c(E), \text{exec}(C1))) \\
& \quad \wedge t(\text{if } E \text{ then } C1) \\
& \quad = \text{if}_s(v(E) \neq 0, t_c(E) + (\text{after}(t_c(E)); t(C1)), \\
& \quad \quad \quad t_c(E))
\end{aligned}$$

The following describes the behaviour of an if-then-else command.

$$\begin{aligned}
& \vdash \text{exec}(\text{if } E \text{ then } C1 \text{ else } C2) \wedge \text{side_effect_free}(E) \\
& \Rightarrow \\
& (\text{stable}(t_c(E)) \wedge \text{until}(t_c(E), \text{no_cpu_activity}) \\
& \wedge ((v(E) \neq 0) \\
& \Rightarrow \\
& \text{following}(t_c(E), \text{exec}(C1) \wedge \text{following}(t(C1), \text{delay}(t(\text{else})))))) \\
& \wedge ((v(E) = 0) \Rightarrow \text{following}(t_c(E), \text{exec}(C2))) \\
& \wedge t(\text{if } E \text{ then } C1 \text{ else } C2) \\
& = \text{if}_s(v(E) \neq 0, t_c(E) + (\text{after}(t_c(E)); t(C1)) + t(\text{else}), \\
& \quad t_c(E) + \text{after}(t_c(E)); t(C2))
\end{aligned}$$

While statements

To describe the effect of while statements, we first introduce a recursive function *iter*. Intuitively, *iter*(*N*, *E*, *C*) evaluates to the time till the *N*-th iteration of the while loop **while** *E* **do** *C* begins.

$$\begin{aligned}
& \vdash \text{iter}(0, E, C) = 0 \\
& \vdash \text{iter}(N + 1, E, C) \\
& \quad = \text{iter}(N, E, C) + (\text{after}(\text{iter}(N, E, C)); t(\text{if } E \text{ then } C)) + t(\text{while})
\end{aligned}$$

Now the function *end_iter* is defined which yields the number of the final iteration of a while loop, if the loop terminates, and gives undefined otherwise.

$$\begin{aligned}
& \vdash \text{end_iter}(E, C) \\
& = \\
& \text{sm_i } N((N \geq 0) \wedge \text{after}(\text{iter}(N, E, C)); (v(E) = 0) \\
& \quad \wedge \forall N'((N' < N) \wedge (0 \leq N') \\
& \quad \Rightarrow \text{after}(\text{iter}(N, E, C)); (v(E) \neq 0)))
\end{aligned}$$

Now if a while loop terminates, its behaviour is described as follows:

$$\begin{aligned}
& \vdash \text{exec}(\text{while } E \text{ do } C) \wedge \text{defined}(\text{end_iter}(E, C)) \\
& \quad \wedge \text{side_effect_free}(E) \\
& \Rightarrow \\
& \forall N(((0 \leq N) \wedge (N \leq \text{end_iter}(E, C)) \\
& \quad \Rightarrow \text{after}(\text{iter}(N, E, C)); \\
& \quad \quad (\text{exec}(\text{if } E \text{ then } C) \\
& \quad \quad \wedge \text{after}(t(\text{if } E \text{ then } C)); \text{delay}(t(\text{while})))) \\
& \wedge t(\text{while } E \text{ do } C) = \text{iter}(\text{end_iter}(E, C), E, C) \\
& \quad + (\text{after}(\text{iter}(\text{end_iter}(E, C), E, C); t_c(E))
\end{aligned}$$

The behaviour of a non-terminating while loop is described by:

$$\begin{aligned}
& \vdash \text{exec}(\text{while } E \text{ do } C) \wedge \text{side_effect_free}(E) \\
& \quad \wedge \neg \text{defined}(\text{end_iter}(E, C)) \\
& \quad \Rightarrow \\
& \quad \forall N((0 \leq N) \\
& \quad \quad \Rightarrow \text{after}(\text{iter}(N, E, C)); \\
& \quad \quad \quad \text{exec}(\text{if } E \text{ then } C) \\
& \quad \quad \quad \wedge \text{following}(t(\text{if } E \text{ then } C), \text{delay}(t(\text{while}))) \\
& \quad \quad \wedge \neg \text{defined}(t(\text{while } E \text{ do } C))
\end{aligned}$$

Other iterative constructs

The meaning of other iterative constructs can be defined in terms of the while command.

The behaviour of the do-while construct may be defined as follows.

$$\begin{aligned}
& \vdash \text{exec}(\text{do } C \text{ while } E) \\
& \quad \Rightarrow \\
& \quad \text{exec}(C; \text{while } E \text{ do } C) \\
& \quad \wedge t(\text{do } C \text{ while } E) = t(C ; \text{while } E \text{ do } C)
\end{aligned}$$

The behaviour of the for construct can be described as follows:

$$\begin{aligned}
& \vdash \text{exec}(\text{for}(C1, E, C2, \text{Body})) \\
& \quad \Rightarrow \\
& \quad \text{exec}(C1; \text{while } E \text{ do } (\text{Body}; C2)) \\
& \quad \wedge t(\text{for}(C1, E, C2, \text{Body})) = t(C1; \text{while } E \text{ do } (\text{Body}; C2))
\end{aligned}$$

3.4.7 Timing verification

The timing property to be verified is that of the requirement outlined in section 3.1: if the code in Appendix B is executing under the conditions given in section 3.1, then in less than 5 seconds from the arrival of the transmission (beginning with an STX), there will be a write to the siren. The conditions are essentially as follows.

1. $v(\text{mode}) = v(\text{NORMALMODE});$
2. $v(\text{first}) \neq 0;$
3. $v(\text{this}) \neq v(\text{FAILEDORDELAYED});$
4. Provided there is no write to the clock, when the next transmission arrives, the clock will read greater than or equal to $5000 - v(\text{ERROR})$ and strictly less than $5000 + v(\text{ERROR}).$

5. The current local clock reading is strictly less than $5000 - v(\text{ERROR})$.
6. The transmission will consist of $v(\text{TRANSSIZE})$ characters such that the $v(\text{SYNC})$ -th of these equals $v(\text{interval})$;
7. The $v(\text{LEVEL})$ -th of these is strictly less than $v(\text{MINLEVEL})$ or strictly greater than $v(\text{MAXLEVEL})$.

The rest of this section indicates how timing axioms such those outlined above could be used to verify the timing requirement. As noted at the beginning of section 3.4, as much as possible of the functional verification is carried out during top down development in the application of the refinement rules. A full refinement process would also give bounds on loop iterations, which can be used in the timing verification. Thus, while the verification axioms given above could be used directly for functional and timing verification, here we emphasise the purely timing aspects, and illustrate how the axioms can be used to obtain timing bounds on the code.

For the detailed timing, we assume that all variables have static extent.

To analyse the timing of the first `if` statement using the `if` rule, the time to compute the guard is first evaluated. The set of possible timings for that is given by

$$T_c(\text{first} \neq 0)$$

which is given by the `!=` rule to be

$$T(\text{first}) \boxplus T_c(\neq, \text{type}(\text{first})) \boxplus T(0).$$

Since `first` is an integer variable, this becomes

$$Tvar(\text{integer}, \text{static}) \boxplus T_c(\neq, \text{integer}) \boxplus Tconst(\text{integer})$$

which can be computed from the values given in Appendix C. The analysis here calls for a worst case upper bound on timing, which we can compute from the given values as 29 CPU cycles.

Now the timing for the body of the `if` is evaluated. This is a sequential composition, and the timing for the first command must be evaluated. The simple assignment rule gives the possible timings for `penultimate = previous` as

$$\begin{aligned} & T(\text{previous}) \boxplus T(=, \text{type}(\text{previous}), \text{extent}(\text{previous})) \\ & = Tvar(\text{integer}, \text{static}) \boxplus T(=, \text{integer}, \text{static}) \end{aligned}$$

From Appendix C, an upper bound is obtained of 12 cycles.

The maximum timing for the statement `previous = this` is also 12 cycles, and the timing for the statement `waiting = 1` is given by

$$\begin{aligned} & T(1) \boxplus T(=, \text{type}(\text{waiting}), \text{extent}(\text{waiting})) \\ & = T_{\text{const}}(\text{integer}) \boxplus T(=, \text{integer}, \text{static}) \end{aligned}$$

from which a maximum timing of 9 cycles is obtained.

To illustrate the timing analysis of a loop, we evaluate the maximum time for one iteration of the loop beginning at the line labelled A1. A single iteration takes a time drawn from the following set:

$$\begin{aligned} & T_c(\text{waiting}) \boxplus T(\text{now} = *((\text{int} *) \text{CLOCKOUT})) \\ & \boxplus T(\text{waiting} = \text{now} < (5000\text{-ERROR})) \boxplus T(\text{while}) \end{aligned}$$

Now

$$T_c(\text{waiting}) = T_{\text{var}_c}(\text{integer}, \text{static})$$

and thus the maximum time for the loop test is 12 cycles.

Next,

$$\begin{aligned} & T(\text{now} = *((\text{int} *) \text{CLOCKOUT})) \\ & = T_{\text{var}}(\text{integer}, \text{static}) \boxplus T(=, \text{integer}, \text{static}) \end{aligned}$$

giving a maximum time for this statement of 12 cycles.

The maximum time to execute `waiting = now < (5000-ERROR)` is the maximum of the set

$$\begin{aligned} & T(\text{waiting} = \text{now} < (5000\text{-ERROR})) \\ & = T(\text{now} < (5000\text{-ERROR})) \boxplus T(=, \text{integer}, \text{static}) \\ & = T_c(\text{now} < (5000\text{-ERROR})) \boxplus T_{\text{const}}(\text{integer}) \boxplus T(\text{else}) \\ & \quad \boxplus T(=, \text{integer}, \text{static}) \end{aligned}$$

Note that in this calculation, we have taken the worst case, when the boolean expression evaluates to true, introducing the timing term $T(\text{else})$.

Evaluating the first of the resulting terms,

$$\begin{aligned} & T_c(\text{now} < (5000\text{-ERROR})) \\ & = T(\text{now}) \boxplus T((5000\text{-ERROR})) \boxplus T_c(<, \text{integer}) \\ & = T_{\text{var}}(\text{integer}, \text{static}) \boxplus T_{\text{const}}(\text{integer}) \\ & \quad \boxplus T_c(<, \text{integer}) \end{aligned}$$

gives a maximum possible time of 29 cycles. Thus the maximum timing for the statement `waiting = now < (5000-ERROR)` is 44 cycles.

Thus a single iteration of the loop takes at worst 74 cycles.

To further illustrate application of the timing rules, consider:

$$\begin{aligned}
& T(\text{dataready} = (\text{temp} \& \text{DRMASK})) \\
&= T(\text{temp} \& \text{DRMASK}) \boxplus T(=, \text{char}, \text{static}) \\
&= T(\text{temp}) \boxplus T(\&, \text{char}) \boxplus T(\text{DRMASK}) \boxplus T(=, \text{char}, \text{static}) \\
&= T_{\text{var}}(\text{char}, \text{static}) \boxplus T(\&, \text{char}) \boxplus T_{\text{const}}(\text{char}) \boxplus T(=, \text{char}, \text{static})
\end{aligned}$$

giving a maximum possible timing for this statement of 29 cycles.

Consider now obtaining the maximum possible execution time of the statement `looking = !dataready && (now <= (5000+ERROR))`. This can be obtained by taking the maximum of the possible times in the case when the first conjunct evaluates to true:

$$\begin{aligned}
& T(\text{looking} = !\text{dataready} \&\& (\text{now} \leq (5000+\text{ERROR}))) \\
&= T(!\text{dataready} \&\& (\text{now} \leq (5000+\text{ERROR}))) \\
&\quad \boxplus T(=, \text{integer}, \text{static}) \\
&= T_c(!\text{dataready} \&\& (\text{now} \leq (5000+\text{ERROR}))) \boxplus T_{\text{const}}(\text{integer}) \\
&\quad \boxplus T(\text{else}) \boxplus T(=, \text{integer}, \text{static}) \\
&= T_c(!\text{dataready}) \boxplus T_c(\text{now} \leq (5000+\text{ERROR}))) \boxplus T_{\text{const}}(\text{integer}) \\
&\quad \boxplus T(\text{else}) \boxplus T(=, \text{integer}, \text{static}) \\
&= T_c(\text{dataready}) \boxplus T_c(\text{now} \leq (5000+\text{ERROR}))) \boxplus T_{\text{const}}(\text{integer}) \\
&\quad \boxplus T(\text{else}) \boxplus T(=, \text{integer}, \text{static}) \\
&= T_{\text{var}_c}(\text{char}, \text{static}) \boxplus T_c(\text{now} \leq (5000+\text{ERROR}))) \\
&\quad \boxplus T_{\text{const}}(\text{integer}) \boxplus T(\text{else}) \boxplus T(=, \text{integer}, \text{static})
\end{aligned}$$

where the second term may be evaluated as illustrated above. We obtain a maximum execution time of 55 cycles.

Since the inter-arrival time of incoming bytes is no more than $m2$, the last message of the transmission will arrive no later than $(v(\text{TRANSSIZE}) - 1) * m2$ cycles after the arrival of the STX. In the worst case this could occur immediately after LSR is read at the statement labelled A3. In this case the time to exit the polling loop beginning at the line labelled A2 is bounded by the maximum of the set

$$\begin{aligned}
& T(\text{temp} = *((\text{char} *) \text{LSR})) \boxplus T(\text{dataready} = \text{temp} \& \text{DRMASK}) \\
&\boxplus T(\text{while}) \boxplus T_c(!\text{dataready}) \\
&\boxplus T(\text{temp} = *((\text{char} *) \text{LSR})) \boxplus T(\text{dataready} = \text{temp} \& \text{DRMASK}) \\
&\boxplus T(\text{while}) \boxplus T_c(!\text{dataready})
\end{aligned}$$

which is computed to be 114 cycles.

Then the maximum time to terminate the enclosing for loop after exiting the while is given by the maximum of

$$\begin{aligned}
& T(\text{temp} = *((\text{char} *) \text{RBR})) \boxplus T(\text{trans}[\text{i}] = \text{temp} \& \text{DATAMASK}) \\
&\boxplus T(\text{while}) \boxplus T(\text{i}++) \boxplus T_c(\text{i} < \text{TRANSSIZE})
\end{aligned}$$

We illustrate the calculation of the second term here:

$$\begin{aligned}
& T(\text{trans}[i] = \text{temp} \ \& \ \text{DATAMASK}) \\
& = T(\text{temp} \ \& \ \text{DATAMASK}) \boxplus T(i) \boxplus T(=, \text{array}(\text{char}), \text{static}) \\
& = T(\text{temp} \ \& \ \text{DATAMASK}) \boxplus T\text{var}(\text{integer}, \text{static}) \\
& \quad \boxplus T(=, \text{array}(\text{char}), \text{static})
\end{aligned}$$

The maximum of this set is 44 cycles. We thus obtain a value of 151 cycles for the maximum time to complete the for loop after exiting the polling loop, and thus a maximum time to complete the for loop after the arrival of the last message of 265 cycles.

Similar application of the timing rules to the remaining code shows that under the assumed conditions a write to the siren occurs in no more than a further 461 cycles and thus within 726 cycles of the last message arriving. We can conclude then that a write to the siren occurs within

$$((v(\text{TRANSSIZE}) - 1) * m2) + 726$$

cycles of the arrival of the STX.

Choosing a typical value of 1 millisecond for $m2$ and assuming that the processor is run at 4 MHz, we see that the timing requirement is easily met for expected transmission lengths (in the core problem, each transmission contains 5 bytes). Of course, more processing is required in the full problem, but similar techniques could be applied to verify timing. Clearly, machine assistance for computation of maximum timings is necessary for practical application.

4 Conclusions

The boiler control problem provided a welcome opportunity to apply our evolving methods to a substantial case study. It was immediately apparent that there are several practical problems that require consideration. The real-time refinement calculus, for example, is very unwieldy when dealing with the large number of invariants in the specification (it was necessary to expand the invariants in each pre and post-condition), and it does not handle the requirements of embedded input/output as well as we would have liked (primarily because the unadorned refinement calculus itself does not cater explicitly for i/o operations [18, ch. 15]). The verification techniques, which had previously been defined at the assembler level, were upgraded to operate at the C programming level. However, it was found to be difficult to keep the predicted execution times sufficiently “narrow”, given the many possible compiler code optimisation strategies that may be applied to a particular C construct. Previously the verification techniques had used only a single time-frame, namely CPU cycles. The multiple frames of reference defined

during the course of this case study, i.e., master clock, local clock, CPU cycles and “real” time, increased the complexity of the task.

The scale of the problem, even in its “core” form, was a cause of difficulty at all stages of specification, design and verification. The current lack of tool support was sorely felt, particularly when changes that affected the whole system structure were required. For example, the way time was modelled in the specification was altered slightly on a number of occasions to improve the way the passage of time was related to the incoming and outgoing streams. Such a fundamental change in the specification inevitably led to the frustrating need to redo much of the refinement work that had already been completed. Tools to assist with the clerical aspects of managing these updates were keenly missed.

Foremost among other methodologies that emphasise a strong formal base is that of the “Mars” group [16, 17]. Their techniques cover all phases of software development for real-time systems, from specification to testing. They too stress the need for tool support and have developed a number of static analysis tools. However, faced with the considerable problems posed by the unpredictable timing behaviour of contemporary programming languages and architectures, they decided to sidestep these problems by defining their own language and architecture, with sufficient restrictions to ensure that all constructs were predictable.

Our techniques are still evolving. Future work will see a closer integration of the techniques and, ultimately, a set of integrated tools to assist with the entire development and verification process.

Acknowledgement

We wish to thank Mark Utting for his careful review of this report.

References

- [1] *Specification for a Software Program for a Boiler Water Content Monitor and Control System*, January 1992. Revision 1.
- [2] National Semiconductor. *INS8250A Asynchronous Communications Element*.
- [3] R. Cardell-Oliver. A Mechanized Theory for the Verification of Real-Time Program Code Using Higher Order Logic. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 375–392. Springer-Verlag, 1992.

- [4] C.J. Fidge. Real-time refinement. In J. Woodcock and P. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 314–331. Springer-Verlag, 1993.
- [5] C.J. Fidge and A.M. Lister. A disciplined approach to real-time systems design. *Information and Software Technology*, 34(9):603–610, September 1992.
- [6] C.J. Fidge and A.M. Lister. The challenges of non-functional computing requirements. In preparation, March 1993.
- [7] C.J. Fidge and M.J. Pilling. Specification languages for distributed real-time software. In *Proc. 5th Australian Software Engineering Conference*, pages 195–200, Sydney, May 1990.
- [8] L. Groves, R. Nickson, and M. Utting. A tactic driven refinement tool. In C.B. Jones, R.C. Shaw, and T. Denvir, editors, *Fifth Refinement Workshop*, pages 272–297. Springer-Verlag, 1992.
- [9] P. Henman and J. Staples. Introduction to functional set theory. Technical Report 144, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1990. Revised September 1991.
- [10] M.E.C. Hull, P.G. O'Donoghue, and B.J. Hagan. Development methods for real-time systems. *The Computer Journal*, 34(2):164–172, April 1991.
- [11] P. Kearney and A. Abbas. Functional verification of a simplified RS232 software repeater problem. Technical Report 91-3, Software Verification Research Centre, Department of Computer Science, University of Queensland, August 1991. Revised May 1992.
- [12] P. Kearney, J. Staples, and A. Abbas. Functional verification of hard real-time programs. In J. van Leeuwen, editor, *Algorithms, Software, Architecture, Information Processing 92 Vol.1*, pages 113–119. Elsevier Science Publishers B.V. (North Holland), 1992.
- [13] P. Kearney, J. Staples, A. Abbas, and Chuichang Liu. Functional verification of real-time procedural code: a simplified RS232 software repeater problem. Technical Report 91-2, Software Verification Research Centre, Department of Computer Science, University of Queensland, August 1991. Revised May 1992.
- [14] J.C. Kelly and Y.S. Sherif. Comparison of four design methods for real-time software development. *Information and Software Technology*, 34(2):74–82, February 1992.

- [15] S. King. *Z* and the refinement calculus. In D. Bjørner, C.A.R. Hoare, and H. Longmaack, editors, *Proc. VDM'90*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188. Springer-Verlag, April 1990.
- [16] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The design of real-time systems: From specification to implementation and verification. *Software Engineering Journal*, 6(3):72–82, May 1991.
- [17] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. An engineering approach to hard real-time system design. In A. van Lamswerde and A. Fugetta, editors, *Proc. Third European Software Engineering Conference*, volume 550 of *Lecture Notes in Computer Science*, pages 166–188. Springer-Verlag, 1991.
- [18] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [19] J. Staples, P. Robinson, and D. Hazel. A functional logic for higher level reasoning about computation. Technical Report 141, Key Centre for Software Technology, Department of Computer Science, University of Queensland, 1989. Revised September 1991; to appear in *Formal Aspects of Computing*.
- [20] Mark Utting and Peter Kearney. Pipeline specification of a MIPS R3000 CPU. Technical Report 92-6, Software Verification Research Centre, Department of Computer Science, University of Queensland, October 1992.
- [21] J.B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

A Requirements document anomalies

The following queries about the boiler requirements [1] were raised during formal specification.

- Sections C-1.6.2a and C-1.6.2c of the requirements document tell us that two *successive* “corrupt” or two *successive* “failed” (i.e., missing) transmissions will be diagnosed as an instrumentation system failure (and thus lead to a shutdown). However, there is no requirement that a failed transmission followed by a corrupt one, or vice versa, should cause a shutdown. Thus, an indefinite sequence of interleaved corrupt and failed transmissions may occur with no error being diagnosed! This was felt to be such a serious error in the requirements specification that we corrected it in our formal specification. The definition of *toocorrupt* in section 3.2.3 allows for failed transmissions.

- The definition of a delayed transmission in section C-1.7 means there can never be delayed messages in two successive intervals; it is unclear if this is the *intention* of the requirements specification, but a strict interpretation of the natural language text in section C-1.7 leads to this conclusion. Furthermore, section C-1.7 implies that a delayed transmission can never be followed by an interval in which the transmission for the second interval fails. The invariant on delayed transmissions in section 3.2.3 models both of these restrictions.
- It is difficult to see how “safety” [1, §5.1] can be *guaranteed* given the potential for data fields in incoming transmissions to be arbitrarily corrupted [1, §C-1.6.2d] in such a way that they appear reasonable but no longer reflect the actual state of the boiler. This problem would be alleviated by including error-detection or error-correction in all transmission protocols. Section C-1.7 implies that the instrumentation system can detect, and correct, transmission failures. It seems strange that the program cannot do likewise.
- There are two distinct forms of timing obligation in the requirements document. Some, such as those in section 4.8, are expressed relative to external events (and hence relative to the “master” clock). Others, such as those in section 4.7, are expressed relative to actions internal to the proposed program (i.e., state changes within the code). This leads to considerable ambiguity. For instance, should the requirement that the audio warning be issued “within 5 seconds of entering shutdown mode” [1, §4.7.1] be taken to mean that the warning be issued within
 1. 5 seconds of a catastrophic situation arising within the boiler itself,
 2. 5 seconds from the time the instrumentation system alerts the program to such a situation (which may involve a 5 second wait due to a delayed transmission), or
 3. 5 seconds from the time the program itself has recognised the potential catastrophe?

In the last case, how much time should be allowed to elapse between the potential catastrophe arising in the boiler, and this internal program state change occurring?¹ In the example used throughout this paper we have assumed case 2, but concede that other interpretations of the requirements document are possible. In any event, safety analysis would be simplified by specifying a maximum time interval (subject to transmission failures)

¹A possible answer to this question is that as much time is allowed to elapse as is compatible with the goal of safety [1, §5.1], but this makes specifying the 5 second delay entirely vacuous.

between transmission of hazard-warning data by the instrumentation system and receipt by the instrumentation system of hazard-avoidance commands.

- There is no “normal” way specified for shutting down the boiler system. (The only way to exit the program is to fail a “self test” [1, §4.2.2].)
- Section 4.7.3 states that “shutdown mode may be entered from any mode”. This is inconsistent with the requirements stated for computer self-test mode in section 4.2.2, however.
- The system test and initialisation mode requirements do not seem to allow for the possibility that the feed pumps are initially on when the program begins (in particular see section 4.3.1d).
- Section B-3.2.2 contradicts Section 4.9.2.1e and the other sections that describe shutdown behaviour.
- It is not clear whether the instrumentation system expects a transmission from the program in every interval or not. If not, then it is not clear what happens if the program does not transmit for a considerable period of time because no change in the boiler system is seen to be necessary.
- It is not clear how long the program should wait for the very first transmission [1, §4.2.1b] given that transmissions may fail or be delayed. Indeed, section 4.2.1b does not seem to ask for any special action if no usable transmissions are received!
- Although the purpose of the specified program [1] is to monitor boiler water levels only, it is surprising that the pressure gauge was not made accessible to the program, given the safety-critical nature of the system.
- The execution time requirements for the computer self-test [1, §4.2] and system test and initialisation [1, §4.3] modes are underspecified. There is no upper bound given for the amount of time that the system may spend in either mode.

B C code

This appendix gives the skeletal C code for the situation considered in the case study described in section 3.1. It shows each statement executed from the beginning of an interval until the audio warning is activated in the situation where a single transmission arrives, while the system is in normal operating mode, that indicates that the boiler water level is unsafe. No optimisations have been performed.

```

#define ERROR 1
#define TRANSSIZE 5
#define LSR ...
#define RBR ...
#define SIREN ...
...
do {
    if (first != 0) {
        penultimate = previous;
        previous = this;
        waiting = 1;
        while(waiting) {
            now = *((int *) CLOCKOUT);
            waiting = now < (5000-ERROR);
        };
        looking = 1;
        while (looking) {
            temp = *((char *) LSR);
            dataready = (temp & DRMASK);
            now = *((int *) CLOCKOUT);
            looking = !dataready && (now <= (5000+ERROR));
        };
        if (dataready) {
            *((char *) CLOCKIN) = 0;
            temp = *((char *) RBR);
            trans[0] = temp & DATAMASK;
            for (i=1; i<TRANSSIZE; i++) {
                dataready = 0;
                while (!dataready) {
                    temp = *((char *) LSR);
                    dataready = temp & DRMASK;
                };
                temp = *((char *) RBR);
                trans[i] = temp & DATAMASK;
            };
            if (previous == FAILEDORDELAYED) {
                ...
            }
            else {
                if (trans[SYNC] != interval)
                    ...
            }
        }
    }
}

```

```

else
    this = OK;
    catastrophe = (((this == CORRUPT) &&
                    ((previous == CORRUPT) ||
                     (previous == FAILED))) ||
                  ((this != CORRUPT) &&
                   ((trans[LEVEL] < MINLEVEL) ||
                    (trans[LEVEL] > MAXLEVEL))));
    }
else {
    ...
}
else {
    ...
};
if (mode != SHUTDOWNMODE) {
    if (mode == SELFTESTMODE)
        ...
    else
        if (mode == SYSTEMTESTMODE)
            ...
        else
            if (mode == NORMALMODE)
                if (mode == NORMALMODE && !catastrophe)
                    ...
                else
                    if (catastrophe) {
                        mode = SHUTDOWNMODE;
                        *((char *) SIREN) = ON;           /* B */
                        ...
                    }
            else
                ...
}
} while(1);

```

The timing requirement of interest asserts that it must be possible to execute all statements between the time when the incoming STX character is identified, and point B, when the siren is activated, inclusive in under five seconds.

C MC6809 Timing Parameters

The timing parameters here are given in clock cycles. Recall that the 6809 can be run at a maximum rate of 4MHz. Some of these timing estimates are fairly loose. As discussed in section 3.4.5, tighter timings can be obtained for particular sequences of code. However, it is likely that the best timing estimates would be obtained from a suitably instrumented compiler.

$$T_{const}(integer) = \{0, 3\}$$

$$T_{const}(char) = \{0, 2\}$$

$$T_{const_c}(char) = T_{const_c}(integer) = \{3, 5, 6\}$$

$$T_{var}(integer, static) = \{5, 6\}$$

$$T_{var_c}(integer, static) = \{8, 9, 10, 11, 12\}$$

$$T_{var}(char, static) = \{2, 5\}$$

$$T_{var_c}(char, static) = \{7, 8, 10, 11\}$$

$$T(+, integer) = T(-, integer) = \{4, 6, 7, 10, 14, 19\}$$

$$T(*, integer) = \{178\}$$

$$T_c(==, integer) = T_c(!=, integer) = \{8, 10, 11, 12, 13, 14, 17, 19, 20\}$$

$$T_c(==, char) = T_c(!=, char) = \{5, 7, 8, 9, 10, 11, 14, 16, 17\}$$

$$T_c(<, integer) = T_c(==, integer)$$

$$T_c(<, char) = T_c(==, char)$$

$$T(&, char) = \{2, 4, 5, 8, 12, 17\}$$

$$T(array_access, char, static) = \{6, 11, 14\}$$

$$T(=, integer, static) = \{5, 6\}$$

$$T(=, array(char), static) = \{11, 14\}$$

$$T(=, char, static) = \{4, 5\}$$

$$T(\text{while}) = \{3, 5, 6\}$$

$$T(\text{else}) = \{3, 5, 6\}$$