Specifying mode requirements of embedded systems

Graeme Smith

Software Verification Research Centre
University of Queensland, Australia
Email: smith@svrc.uq.edu.au

Abstract

This paper presents a formal notation for specifying requirements of embedded systems which exhibit continuous, real-time behaviour and move through various *modes* under digital control. It does this by extending an existing formal notation supporting continuous, real-time behaviour with an explicit concept of modes. The resulting notation avoids the subtleties which would otherwise arise when specifying mode-related requirements. It is therefore ideal as a means of specifying and communicating requirements of embedded systems.

Keywords: formal specification, requirements engineering, embedded systems

1 Introduction

Embedded systems are becoming ubiquitous in modern society. Such systems comprise a digital controller which interacts in real time with a continuously changing physical environment. To cater for the growing complexity of applications, such controllers are increasingly implemented in software. Many embedded systems are safety-critical, being capable of causing harm (i.e., loss of life or injury) by failure to correctly interact with their environment. Examples include fly-by-wire aircraft, heart pacemakers and process control systems of chemical and electrical power plants. Hence, there has been growing interest in using formal methods to specify (functional and performance) requirements of embedded systems, or more specifically, requirements involving relationships between continuous values and time.

Research in this area has produced a number of different approaches, notable among which are those based on *timed traces*, i.e., where systems are modelled by functions whose domains range over all values of time. Examples include the Duration Calculus [Hansen and Zhou Chaochen, 1997], the timed stream model [Broy, 1993] and the timed refinement calculus [Mahony and Hayes, 1992, Fidge et al., 1998b]. These techniques provide a precise and intuitive approach to modelling time-dependent behaviour, essentially the same as that used in the physical sciences.

However, when it comes to modelling systems which, through digital control, move between various *modes* in which the time-dependent behaviour is different, the techniques tend to be much less intuitive. Since they model the value of variables as functions over all time, they do not explicitly support notions of time-bounded modes and mode changes. These need to be encoded indirectly via the interval concatenation, or 'chop', operator common to these notations. From a requirements engineering point of view, this is problematic because such specifications are difficult to understand and reason about.

In this paper, we present a formal specification language which builds on the timed trace paradigm, specifically on the notation of the timed refinement calculus, by introducing a notion of modes and operators for combining modes. We begin, in Section 2, by providing an overview of the notation of the timed refinement calculus. In Section 3, we introduce the concept of modes and the syntax of our notation. We illustrate the notation using the well-known Steam Boiler case study [Abrial et al., 1996]. A specification of the entire system, as well as another which separates the software controller from the boiler hardware, are provided. In Section 4, we provide the semantics of our notation and discuss briefly how the formal basis it provides can be used to gain confidence in the requirements engineering process.

2 Timed Trace Notation

The specification notation of the timed refinement calculus [Mahony and Hayes, 1992, Mahony, 1992] is set-theory based and uses the syntax of the well-known Z specification language [Spivey, 1992]. Since its initial definition, it has been extended with a simple set-theoretic notation for concisely expressing time *intervals*, i.e., non-empty contiguous sets of times, and operators for accessing interval endpoints. In this section, we present a simplified subset of the extended notation based on that of Fidge et al. [Fidge et al., 1998b] which provides the syntax and semantics of a minimal set of operators in addition to those of standard set theory. This notation has been successfully employed to specify the requirements of a sizeable case study [Smith and Fidge, 2000] as well as those of the Nulka Active Missile Decoy, a hovering rocket developed for the Australian and US Departments of Defence. However, it remains difficult for non-experts to read and write specifications in the notation.

Absolute time, $\mathbb{T}$, is modelled by the real numbers $\mathbb{R}$.

$$\mathbb{T} == \mathbb{R}$$

For the purposes of this paper, we assume the units of $\mathbb{T}$ is seconds.

Observable variables of a systems are modelled as timed traces, i.e., total functions from the time domain to a type representing the set of all values the variable may assume. For example, a variable indicating that a boiler system is operating may be declared using the Boolean type $\mathbb{B}$ as follows.

$$OP : \mathbb{T} \rightarrow \mathbb{B}$$

Copyright ©2001, Australian Computer Society, Inc. This paper appeared at the Twenty-Fifth Australasian Computer Science Conference (ACSC2002), Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Functions modelling physical quantities generally map from the time domain to some contiguous subset of the real numbers. In most cases, such functions are *differentiable* (i.e., continuous and smooth). To facilitate specifying this, whenever X is a contiguous set of real numbers, we use the notation $\mathbb{T} \rightsquigarrow X$ to represent the set of all differentiable functions from the time domain to X [Fidge et al., 1998a]. For example, the steam demand SD and steam output SO of the boiler system may be declared, using a type $RATE == \{r : \mathbb{R} \mid r \geqslant 0\}$, as follows.

$$SD, SO : \mathbb{T} \rightsquigarrow RATE$$

Given this declaration, we let the derivatives of SD and SO be denoted by $\underline{d}\, SD$ and $\underline{d}\, SO$ respectively.

System requirements are specified using time intervals over which properties hold. Sets of such intervals can be specified using the interval brackets $\langle\,\rangle$. For example, the set of intervals where the boiler is operational for the whole interval is specified as:

$$\langle OP \rangle$$

An interval I is in the set of intervals $\langle OP \rangle$ if, for all times t in I, $OP(t)$ is true.

In general, the property in brackets is any first-order predicate in which total functions from the time domain to some type X may be treated as values of type X. The elision of explicit references to the time domain of these functions results in specifications which are more concise and readable.

The starting point, end point and duration of intervals can also be accessed using the reserved symbols α, ω and δ respectively. For example, given duration $N : \mathbb{T}$, the set of intervals where the steam boiler has been operating for at least N seconds is specified as:

$$\langle OP \wedge \delta \geqslant N \rangle$$

Requirements are specified by predicates formed by combining sets of intervals using operators from set theory such as $\cap$, $\cup$ and $\subseteq$, and the interval concatenation operator ';'. The latter operator forms a set of intervals by joining intervals from one set to those of another whenever their end points meet [Fidge et al., 1998b]. For example, given $err : RATE$, the requirement that the steam output must be within err of the steam demand whenever the boiler has been operating for at least N seconds can be specified as:

$$\langle OP \wedge \delta \geqslant N \rangle \subseteq (\langle true \rangle \,;\, \langle SO \in SD \pm err \rangle)$$

Note that $\langle true \rangle$ is the set of all possible intervals, and hence the right-hand side above is the set of all intervals that end with the predicate $SO \in SD \pm err$ true.

The above specification is arguably understandable by someone familiar with set theory. However, it is not without subtlety. The fact that $SO \in SD \pm err$ is always true after N seconds can only be gleaned from the realisation that, given an interval in the left-hand set of duration greater than N seconds, all prefixes of this interval of duration at least N seconds are also in the set. For each of these prefixes, $SO \in SD \pm err$ must be true at the end of the prefix. Hence, it follows that it must be true after N seconds in the original interval.

3 Modes

A *mode* is a set of intervals in which similar behaviours can be observed. In the steam boiler example of Section 2, there are two obvious modes: when the boiler is operating and when it is not. Often we want to specify requirements on the timing properties of a mode, e.g., its duration, or to specify requirements on the occurrence of a mode with respect to other modes. In a timed-trace notation such as that of the timed refinement calculus, this can be difficult to specify resulting in specifications which are difficult to understand [Smith, 1999].

As an example, consider adding a requirement to those in Section 2, that whenever the steam boiler is not operational, it must remain so for a duration of at least $MinStop : \mathbb{T}$ seconds. This requirement may be necessary for the boiler to undergo initialisation tests before starting again. The easiest way to specify this is as follows.

$$\langle OP \rangle \,;\, \langle \neg\, OP \rangle \,;\, \langle OP \rangle \subseteq \langle \delta \geqslant MinStop \rangle$$

The intuition behind this specification is that the intervals $\langle OP \rangle$ are not restricted in duration and so may be arbitrarily small (and hence of zero duration)[1]. Hence, the above predicate states that all *maximal* intervals in the set $\langle \neg\, OP \rangle$, i.e., those that are not strict prefixes or postfixes of any other interval in the set, have to satisfy the constraint on the duration.

3.1 Syntax

The notation presented in the remainder of this paper aims at overcoming the subtleties that arise when specifying requirements using the interval concatenation operator. In particular, it makes the notion of modes explicit and introduces operators for combining modes. The latter are not only used for describing system requirements, but also for describing requirements of complex modes in which a number of (sub) modes are composed sequentially or concurrently. Semantically, modes and systems are not distinguished and hence, from this point on, we discuss modes only.

A simple mode is defined by a statement $M \mathrel{\widehat{=}} \langle P \rangle$. The right-hand side of the definition is the definition of a set of intervals using the syntax of the timed refinement calculus. These intervals are those in which the mode may be operating. All variables in P are regarded as *output* (i.e., system controlled) variables unless they underlined in which case they are regarded as *input* (i.e., environment controlled) variables. For example, the mode corresponding to when the boiler is operational and the steam output is an acceptable approximation of the steam demand is specified as:

$$Running \mathrel{\widehat{=}} \langle OP \wedge SO \in \underline{SD} \pm err \rangle$$

A complex mode is defined by a statement $M \mathrel{\widehat{=}} Q$ where Q is either a sequence of modes of the following form:

$M_1 {}^\frown M_2$ – mode M_1 until some arbitrary instant then mode M_2

$M_1 \downarrow M_2$ – mode M_1 for as long as possible then mode M_2

$\widehat{M}$ – mode M until some arbitrary instant then mode M again

$\downarrow M$ – mode M for as long as possible then mode M again

[1] An interval cannot be empty [Fidge et al., 1998b]. However, it may comprise only a single point of time in which case its duration is zero.

$$SteamBoiler \mathrel{\widehat=} Init ^\frown \widehat{SB}$$

$$Init \mathrel{\widehat=} \langle \neg\ OP \wedge \delta \geqslant MinInit \rangle$$

$$SB \mathrel{\widehat=} (Operate ^\frown Stop) \square Operate$$

$$Operate \mathrel{\widehat=} (RunUp ^\frown Running) \square RunUp$$

$$RunUp \mathrel{\widehat=} \langle OP \wedge \delta \leqslant MaxRunUp \rangle$$

$$Running \mathrel{\widehat=} \langle OP \wedge (\underline{SD} \leqslant MaxSteam \Rightarrow SO \in \underline{SD} \pm err) \rangle$$

$$Stop \mathrel{\widehat=} \langle \neg\ OP \wedge \delta \geqslant MinStop \rangle$$

Figure 2: System specification of the steam boiler.

or constructed using the following operators:

$M_1 \square M_2$ – mode M_1 or mode M_2

$M_1 \parallel M_2$ – mode M_1 running concurrently with mode M_2

The difference between the binary operators $^\frown$ and $\downarrow$ is that the latter only allows a mode change when the property of the first mode is no longer true, whereas the former allows a mode change at any time which respects the constraints on the durations, and start and end times of the modes. The $^\frown$ operator is therefore useful for modelling mode changes which occur due to influences outside the scope of the requirements. No particular reason for the change of mode need be specified. The $\downarrow$ operator, on the other hand, is useful for specifying mode changes which occur for defined reasons. The unary forms of these operators facilitate the definitions of behaviours which repeat over time.

The use of these operators, as well as the operators $\square$ and $\parallel$, is illustrated in the following case study. The semantics of the operators is defined in Section 4.

3.2 Steam Boiler Case Study

One of the challenges of requirements engineering for embedded systems is that the requirements must be specified at two levels. One level is required to capture the requirements of the entire system (both the hardware and the software). This level is required for communication with end users of the system and/or customers for which the system is being implemented. The second level separates the hardware and software components of the system. While such a separation of a system into components often occurs after the requirements phase, during design, it is essential that it occurs as part of the requirements phase for embedded systems. This is so that requirements can be communicated to the implementors who will generally be involved with only the software or only the hardware.

The Steam Boiler case study [Abrial et al., 1996] describes a system comprising a steam boiler and associated hardware such as water pumps and valves together with various sensors which report the state of the boiler to a monitor computer. Initially, the boiler undergoes a number of tests and possibly adjusts its water level to make sure that it can operate correctly. When these tests and adjustments are successfully completed, the monitor informs the system's environment which demands steam at a rate within the system's capabilities. If the boiler fails, either to complete its tests or during operation, the monitor informs the environment and stops the operation of the boiler. This action is critical because if the boiler continues to run, it can be damaged.

Our specifications are based on those developed by Mahony et al. [Mahony et al., 1994], but extended, as by Smith [Smith, 1999], to model system start-up, an acceptable error between steam demand and steam output, and a run-up time for the steam output to meet the steam demand within this error.

We begin by defining the input and output variables of our system specification (see Figure 1).

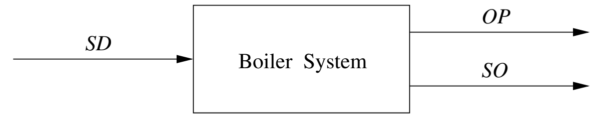

Figure 1: System view of the steam boiler.

These are identical to the definitions in Section 2.

$OP : \mathbb{T} \to \mathbb{B}$
$SD, SO : \mathbb{T} \rightsquigarrow \mathbb{R}$

We then define the constants necessary to formalise the requirements on the performance of the system. The system need only be able to supply steam up to a certain maximum level, $MaxSteam : RATE$, where $RATE$ is defined as in Section 2. Also, the error, $err : RATE$, between the steam output and steam demand is strictly positive, i.e., $err > 0$.

The steam boiler is then specified as in Figure 2 where $MinInit : \mathbb{T}$, $MinStop : \mathbb{T}$ and $MaxRunUp : \mathbb{T}$ are the minimum time required for initialisation, the minimum time required before the system can begin operating again after stopping, and the maximum time for the the system to reach *MaxSteam* output from a zero steam rate, respectively.

The specification states that the steam boiler starts in a mode *Init* and then, at some arbitrary instant, repeatedly behaves according to the complex mode *SB*.

- The mode *Init* is defined as one in which *OP* is not true and which lasts at least *MinInit* seconds.
- The mode *SB* either behaves as mode *Operate* followed, at some arbitrary instant, by mode *Stop*, or simply as mode *Operate*.

$$Boiler \mathrel{\widehat=} \widehat{B}$$

$$B \mathrel{\widehat=} Init_B \Box Operate_B \Box Stop_B$$

$$Init_B \mathrel{\widehat=} DetectInit_B {}^\frown Initialise_B$$

$$DetectInit_B \mathrel{\widehat=} \langle \underline{CD} = init \wedge \delta \leqslant MaxDelay_B \rangle$$

$$Initialise_B \mathrel{\widehat=} \langle \neg\ OR \wedge \delta \geqslant MinInit \rangle$$

$$Operate_B \mathrel{\widehat=} (RunUp_B {}^\frown Running_B) \Box RunUp_B$$

$$RunUp_B \mathrel{\widehat=} \langle \underline{CD} = operate \wedge \delta \leqslant MaxRunUp_B \rangle$$

$$Running_B \mathrel{\widehat=} \langle \underline{CD} = operate \wedge (\underline{SD} \leqslant MaxSteam \Rightarrow SO \in \underline{SD} \pm err_B) \rangle$$

$$Stop_B = DetectStop_B {}^\frown Stopped_B$$

$$DetectStop_B \mathrel{\widehat=} \langle \underline{CD} = stop \wedge \delta \leqslant MaxDelay_B \rangle$$

$$Stopped_B \mathrel{\widehat=} \langle \underline{CD} = stop \wedge \neg\ OR \rangle$$

Figure 4: Specification of the boiler component.

- Mode *Stop* is similar to *Init*. During *Stop*, *OP* is not true and it lasts at least *MinStop* seconds.
- Mode *Operate* begins as mode *RunUp*, during which *OP* is true and which lasts for no longer than *MaxRunUp* seconds, and then, at some arbitrary instant, becomes mode *Running*. During the latter mode, *OP* is also true and $SO \in SD \pm err$ provided that *SD* is less than or equal to *MaxSteam*. In general, requirements on the environments under which a system is supposed to behave in a certain way can be specified using implication as in the definition of *Running*.

  The choice operator $\Box$ is used to allow for the possibility that the mode may change before run-up has completed.

The inputs and outputs of the component specification, in terms of a software monitor and a hardware boiler, are shown in Figure 3.

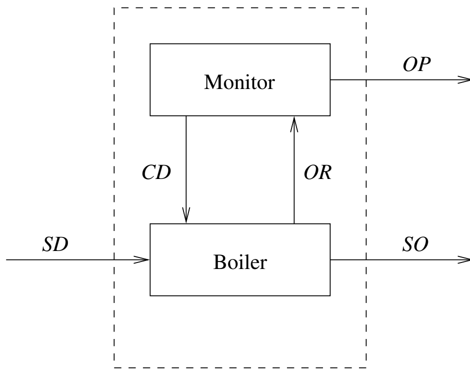

Figure 3: Component view of the steam boiler.

The boiler has an input $CD ::= init \mid operate \mid stop$ which provides it with the commands to initialise, operate or stop, and an output $OR : \mathbb{B}$ which models the indication from the sensors that the boiler is operating correctly.

The steam boiler can be specified as the boiler and monitor components operating in parallel.

$$SteamBoiler_{MB} \mathrel{\widehat=} Monitor \parallel Boiler$$

The specifications of the boiler and monitor components are given in Figure 4 and Figure 5 respectively.

The boiler simply waits for commands from the monitor and behaves accordingly. It is allowed to take up to $MaxDelay_B : \mathbb{T}$ seconds before detecting a command. As shown by Smith [Smith, 1999], the values for the allowable error in the steam output $err_B : RATE$ and the maximum run-up time $MaxRunUp_B : \mathbb{T}$ of the boiler are smaller than those of the system. This is necessary to account for this time delay, as well as that which may occur between the boiler failing and the monitor reacting.

The boiler is in one of three modes, $Init_B$, $Operate_B$ or $Stop_B$, depending on the input command *CD*.

- If the command is *init* then the boiler is in $Init_B$ mode. After a time less than $MaxDelay_B$ seconds, the boiler must indicate that it is not operational for at least *MinInit* seconds (while it performs its initialisation tests).
- If the command is *operate* then the boiler is in $Operate_B$ mode. After a run-up of less than $MaxRunUp_B$ seconds, if the input command has not changed and the steam demand is less than or equal to *MaxSteam*, the boiler must provide steam output within err_B of the demanded rate.
- If the command is *stop*, the boiler is required to indicate that it is not operational after a period of $MaxDelay_B$ seconds.

The monitor's behaviour is more sophisticated than that of the boiler. It cycles through its modes in a particular order, only breaking this order when a failure occurs. Failures are detected via the boiler output *OR*. They must be detected within $MaxDelay_M : \mathbb{T}$ seconds.

The monitor is modelled as having a *Normal* and *Failed* mode of operation.

- In the *Normal* mode, the monitor sends the *init* command to the boiler and waits for up to

$$Monitor \mathrel{\widehat=} \widehat{M}$$

$$M \mathrel{\widehat=} Normal \,\Box\, Failed$$

$$Normal \mathrel{\widehat=} Init_M {}^\frown (((Operate_M \downarrow Fail_M) {}^\frown Stop_M) \,\Box\, Operate_M)$$

$$Init_M \mathrel{\widehat=} \langle CD = init \wedge \neg\ OP \wedge \delta \leqslant MaxWait_M \rangle$$

$$Operate_M \mathrel{\widehat=} \langle \underline{OR} \wedge CD = operate \wedge OP \rangle$$

$$Fail_M \mathrel{\widehat=} \langle \neg\ \underline{OR} \wedge CD = operate \wedge OP \wedge \delta \leqslant MaxDelay_M \rangle$$

$$Stop_M \mathrel{\widehat=} \langle CD = stop \wedge \neg\ OP \rangle$$

$$Failed \mathrel{\widehat=} TimeOut_M {}^\frown Stop_M$$

$$TimeOut_M \mathrel{\widehat=} \langle \neg\ \underline{OR} \wedge CD = init \wedge \neg\ OP \wedge \delta = MaxWait_M \rangle$$

Figure 5: Specification of the monitor component.

$MaxWait_M$: $\mathbb{T}$ seconds for OR to become true. After this, it sends the *operate* command to the boiler and indicates to the environment, via OP, that the boiler is operating. It continues to do this as long as OR is true. The $\downarrow$ operator is used to model the fact that the monitor will send the *operate* command as long as possible. If OR becomes false, it responds by sending the *stop* command and indicating that the boiler is no longer operational within $MaxDelay_M$ seconds.

- In the *Failed* mode, the monitor, after waiting $MaxWait_M$ seconds for the boiler to successfully initialise, sends a *stop* command to the boiler.

The specifications of the boiler system and boiler and monitor components presented in this section are much clearer than similar specifications produced by the author using the timed trace notation of Section 2 alone [Smith, 1999]. The latter specifications required extensive use of the interval concatenation operator to specify durations of modes and mode changes. As well as the inherent subtleties with using this approach, the resulting specifications fail to show the partitioning of the overall behaviours into easily understood modes. A clear understanding of the system can only be inferred through some amount of analysis.

The use of modes for specifying embedded systems is in fact well-established. A number of automata-based approaches have been developed [Alur et al., 1995, Kesten et al., 1998, Lynch et al., 1996] in which vertices represent modes and edges represent instantaneous mode changes. The semantic identification of systems and modes in our approach, however, leads to an approach which is more structured (since modes can be specified in terms of sub-modes) and flexible (since traditionally system-level operators such as parallel composition can be used for modes and vice-versa). While this semantic identification is quite simple in a set-theoretic semantics such as ours, it would be more difficult in an automata-based approach.

4 Towards a Mode Calculus

One of the advantages of using a formal specification language is that analysis of descriptions is precise and potentially automatable. Such analysis is crucial for safety critical applications. In the context of requirements engineering of embedded systems, formal analysis can also be used to verify that a system specification is *refined* by its component specification. That is, that the component specification meets all requirements specified in the system specification. This relationship between specifications is essential when the different specifications are being used to communicate to different parties involved with the system.

For a specification language to be formal, it must have a precise mathematical semantics. In this section, we provide a semantics for the notation introduced in Section 3. We also explain briefly how this could be used as the basis for a set of rules for refinement. A complete set of rules, however, is beyond the scope of this paper.

We let $\mathbb{I}$ denote the set of all time intervals and, given a predicate P with free occurrences of timed trace variables $\vec{v}$, let $P[\vec{v}(t)/\vec{v}]$ denote the predicate with all occurrences of v from $\vec{v}$, which are not dereferenced, replaced by $v(t)$ [Fidge et al., 1998b]. The semantics of our notation is then as given in Figure 6.

The semantic function $\lceil _ \rceil$ returns the set of intervals in which a mode could be operating. The functions *inf* and *sup* return the start and end points (infimum and supremum) of an interval respectively.

- If the mode is a simple mode of the form $\langle P \rangle$, then its set of intervals comprises all those for which P holds.
- If the mode is formed using the binary $^\frown$ operator, then its set of intervals comprises all those which are the union of intervals from the constituent modes whose end points meet.
- If the mode is formed using the binary $\downarrow$ operator, then its set of intervals comprises all those which are the union of intervals from the constituent modes whose end points meet, and where all intervals which extend the first interval are not intervals of the first mode.
- If the mode is formed using the unary $^\frown$ operator, then its set of intervals comprises the concatenations of sequences of intervals of the constituent mode.
- If the mode is formed using the unary $\downarrow$ operator, then its set of intervals comprises the concatenations of sequences of intervals of the constituent mode which do not have extensions in the constituent mode.
- If the mode is formed by the $\Box$ operator, then its set of intervals comprises those intervals belonging to at least one of the constituent modes.

$$\lceil\langle P\rangle\rceil = \{i : \mathbb{I} \mid \exists\, \alpha, \omega, \delta : \mathbb{T} \bullet \alpha = \mathit{inf}(i) \wedge \omega = \mathit{sup}(i) \wedge \delta = \omega - \alpha \wedge (\forall\, t : i \bullet P[\vec{v}(t)/\vec{v}])\}$$
where $\vec{v}$ are the free occurrences of timed trace variables in P

$$\lceil M_1 \frown M_2\rceil = \{i : \mathbb{I} \mid \exists\, i_1 : \lceil M_1\rceil;\ i_2 : \lceil M_2\rceil \bullet \mathit{inf}(i_2) = \mathit{sup}(i_1) \wedge i = i_1 \cup i_2\}$$

$$\lceil M_1 \downarrow M_2\rceil = \{i : \mathbb{I} \mid \exists\, i_1 : \lceil M_1\rceil;\ i_2 : \lceil M_2\rceil \bullet \mathit{inf}(i_2) = \mathit{sup}(i_1) \wedge i = i_1 \cup i_2 \wedge (\forall\, i', i'' : \mathbb{I} \mid i' = i_1 \cup i'' \wedge \mathit{inf}(i'') = \mathit{sup}(i_1) \bullet i' \notin \lceil M_1\rceil)\}$$

$$\lceil\widehat{M}\rceil = \{i : \mathbb{I} \mid \exists\, s : \mathrm{seq}\lceil M\rceil \bullet i = \bigcup s \wedge (\forall\, n : 1 \mathinner{..} \#s - 1 \bullet \mathit{inf}(s(n+1)) = \mathit{sup}(s(n)))\}$$

$$\lceil\downarrow M\rceil = \{i : \mathbb{I} \mid \exists\, s : \mathrm{seq}\lceil M\rceil \bullet i = \bigcup s \wedge (\forall\, n : 1 \mathinner{..} \#s - 1 \bullet \mathit{inf}(s(n+1)) = \mathit{sup}(s(n)) \wedge (\forall\, i', i'' : \mathbb{I} \mid i' = s(n) \cup i'' \wedge \mathit{inf}(i'') = \mathit{sup}(s(n)) \bullet i' \notin \lceil M_1\rceil))\}$$

$$\lceil M_1 \Box M_2\rceil = \lceil M_1\rceil \cup \lceil M_2\rceil$$

$$\lceil M_1 \parallel M_2\rceil = \lceil M_1\rceil \cap \lceil M_2\rceil$$

Figure 6: Semantics of mode operators.

- If the mode is formed by the $\parallel$ operator, then its set of intervals comprises those belonging to both of the constituent modes.

Given this semantics, a system, or mode, M_1 is refined by a system, or mode, M_2 when the set of intervals of M_2 are a subset of those of M_1. That is, M_2 only operates over intervals which are intervals of M_1. Hence, any properties which are true for M_1 are also true for M_2.

Formally,

$$M_1 \sqsubseteq M_2 \quad \Leftrightarrow \quad \lceil M_2\rceil \subseteq \lceil M_1\rceil$$

where $\sqsubseteq$ denotes "is refined by".

Given this definition, it is now possible to check, under certain values of the constants such as *err* and err_B, whether $\mathit{SteamBoiler}_{MB}$ is a refinement of *SteamBoiler*, and hence whether the specifications in Section 3 are consistent. While this could be done by calculating the sets of intervals corresponding to each specification, this process, unless automated, is only really feasible for small specifications.

A preferable technique is to work at the level of the specification, rather than the semantics. To do this, we need to develop a set of rules for both reasoning and refinement which are sound with respect to the semantics. The following is an example of a possible refinement rule.

$$\langle P\rangle \sqsubseteq \langle Q\rangle \quad \Leftrightarrow \quad Q \Rightarrow P$$

It is obviously sound since the set of intervals over which Q hold will be a subset of those over which P holds only when Q is stronger than P, i.e., $Q \Rightarrow P$.

Similarly, refinement rules involving other operators can also be defined. For example, it is also possible to show the following is sound.

$$\langle P\rangle \sqsubseteq \langle Q\rangle \parallel \langle R\rangle \;\Leftrightarrow\; Q \wedge R \Rightarrow P$$

Derivation of an adequate set of such rules is an area of future work.

5 Conclusion

We have presented a notation for specifying the requirements of embedded systems based on a concept of modes. This notation has a formal semantics allowing us to precisely analyse specifications and prove one specification is a refinement of another. The latter is particularly useful for proving consistency between high-level specifications of entire systems (suitable for communication with customers) and component specifications in which hardware and software components are separated (suitable for communication with implementors). The use of an explicit concept of mode makes the notation more readable, and less prone to subtlety, than other formal notations capable of modelling real-time and continuous behaviour.

Acknowledgements

Thanks to Colin Fidge and Ian Hayes for helpful comments on this work. This work is funded by Australian Research Council grant number A49801500: *A Unified Formalism for Concurrent Real-Time Software Development.*

References

[Abrial et al., 1996] Abrial, J.-R., Börger, E., and Langmaack, H. (1996). *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Alur et al., 1995] Alur, R., Courcoubetis, C., Halbwachs, N., Helzinger, T., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34.

[Broy, 1993] Broy, M. (1993). Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46.

[Fidge et al., 1998a] Fidge, C., Hayes, I., and Mahony, B. (1998a). Defining differentiation and integration in Z. In Staples, J., Hinchey, M., and Liu, S., editors, *IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, pages 64–73. IEEE Computer Society Press.

[Fidge et al., 1998b] Fidge, C., Hayes, I., Martin, A., and Wabenhorst, A. (1998b). A set-theoretic model for real-time specification and reasoning. In Jeuring, J., editor, *Mathematics of Program Construction (MPC'98)*, volume 1422 of *Lecture Notes in Computer Science*, pages 188–206. Springer-Verlag.

[Hansen and Zhou Chaochen, 1997] Hansen, M. and Zhou Chaochen (1997). Duration calculus: Logical foundations. *Formal Aspects of Computing*, 9(3):283–330.

[Kesten et al., 1998] Kesten, Y., Manna, Z., and Pnueli, A. (1998). Verification of clocked and hybrid systems. In Rozenberg, G. and Vaandrager, F., editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 4–73. Springer-Verlag.

[Lynch et al., 1996] Lynch, N., Segala, R., Vaandrager, F., and Weinberg, H. (1996). Hybrid I/O automata. In Alur, R., Henzinger, T., and Sontag, E., editors, *Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer-Verlag.

[Mahony, 1992] Mahony, B. (1992). *The Specification and Refinement of Timed Processes*. PhD thesis, Department of Computer Science, University of Queensland.

[Mahony and Hayes, 1992] Mahony, B. and Hayes, I. (1992). A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826.

[Mahony et al., 1994] Mahony, B., Millerchip, C., and Hayes, I. (1994). A boiler control system: Overview of a case study in timed refinement. In Belluz, D. D. B. and Ratz, H., editors, *Software Safety: Everybody's Business, Proceedings of the 1993 International Invitational Workshop on Design and Review of Software-Controlled Safety-Related Systems, Ottawa*, pages 189–208. The Institute of Risk Research.

[Smith, 1999] Smith, G. (1999). Specification and refinement of a real-time control system. In Edwards, J., editor, *Australasian Computer Science Conference (ACSC 99)*, pages 360–371. Springer.

[Smith and Fidge, 2000] Smith, G. and Fidge, C. (2000). Incremental development of real-time requirements: The light control case study. *Journal of Universal Computer Science*, 6(7):704–730.

[Spivey, 1992] Spivey, J. (1992). *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition.