Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 214 (2008) 359–380
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Refactoring Real-time Specifications

Graeme Smith[1]

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

Tim McComb[2]

ARC Centre of Excellence in Bioinformatics, Institute for Molecular Biology,
The University of Queensland, Australia

Abstract

This paper presents an approach to refactoring real-time specifications written using Real-Time Object-Z. This allows implementation components such as clocks and sensors, not necessarily present in an initial abstract specification, to be introduced via a sequence of refinement steps. The approach, based on similar work for Object-Z, is enabled by a semantics of object instantiation and operation synchronisation introduced in this paper. Means of refining synchronising operations to reflect the timing and causality constraints of an implementation are also presented.

Keywords: Refactoring, formal development, real-time embedded systems, refinement, Real-Time Object-Z

1 Introduction

The way in which we structure an abstract specification of a system is often quite different to the way in which we structure its implementation. Generally, an implementation will have many more interacting components. Including such architectural details in an initial specification may obscure the essential functionality and complicate reasoning. Additionally, the final implementation architecture may not be able to be predicted until further into the development process.

[1] Email: smith@itee.uq.edu.au
[2] Email: t.mccomb@imb.uq.edu.au

doi:10.1016/j.entcs.2008.06.016

This is particularly true of object-oriented software systems where implementations require a range of classes addressing what are primarily implementation concerns, e.g., for user interface specific functionality, internals of data structures, and library interfaces. This has led to the development of structure-transforming refactoring approaches for a number of object-oriented specification languages [6,7,9,10,12,1,5]. In particular, McComb and Smith [12] have developed a way to introduce arbitrary numbers of objects into Object-Z specifications [15] using refinement.

The need for introducing design into specifications is also important in the domain of real-time embedded systems, i.e., systems where software components interact with a continuously changing non-software environment. Typically, components such as clocks, although central to any implementation, are ignored in abstract specifications of such systems. Also, sensor devices (distinct components in an implementation) are commonly modelled as simple inputs to another component at the specification level. Hence, a means of refactoring real-time specifications is desirable.

Real-Time Object-Z [17] is an extension of Object-Z which enables the specification of real-time embedded systems. It adds the notation of the Timed Interval Calculus (TIC) [4] to Object-Z enabling real-time constraints and interactions with a continuous environment to be specified. In this paper, we adapt the Object-Z refactoring approach of McComb and Smith to Real-Time Object-Z. To do so, we first need to define a semantics of objects in Real-Time Object-Z as the current definition of the formalism does not support object instantiation.

The paper is organised as follows. In Section 2, we introduce Real-Time Object-Z using a speedometer example. In Section 3, we introduce object instantiation to Real-Time Object-Z and, in particular, provide a means of specifying synchronisation between operations in different objects. In Section 4, we discuss the adaptation of the Object-Z refactoring approach and refactor the speedometer example to reflect a particular design. In Section 5, we examine the issues of refinement and compositionality that arise from our semantics of objects before concluding in Section 6.

2 Real-Time Object-Z

Real-Time Object-Z [17] is an integration of Object-Z [15] and the Timed Interval Calculus (TIC) [4] aimed at specifying systems in which both complex data structures and continuous real-time variables play a role. Components are specified using Object-Z's class construct extended with TIC predicates describing the component's environmental assumptions and effects. These predicates constrain the behaviour of the Object-Z class and define its interactions with its continuous environment. They refer exclusively to *timed trace* variables, i.e., variables whose types are total functions mapping times to the value the variable assumes at those times. For example, given that $\mathbb{T}$ denotes absolute time (in seconds), the following expresses that a variable $v : \mathbb{T} \rightarrow \mathbb{R}$ becomes equal to a continuous and differentiable (denoted by the function symbol $\rightsquigarrow$ [3]) variable $u : \mathbb{T} \rightsquigarrow \mathbb{R}$ within 0.1 seconds whenever $u > 10$.

$$\langle u > 10 \rangle \subseteq \langle \delta = 0.1 \rangle \; ; \langle v = u \rangle$$

The brackets $\langle \; \rangle$ are used to specify a set of time intervals. The left-hand side of the above predicate denotes the set of all time intervals where, for all times t in the intervals, $u(t)$ is greater than 10. The right-hand side of the above expression comprises two sets of intervals. The first uses the reserved symbol δ which denotes the duration of an interval. Hence, this set contains all those intervals with duration 0.1 seconds. The second set denotes all intervals in which (for all times in the intervals) v equals u. It is combined with the first set of intervals using the concatenation operator ';'. This operator forms a set of intervals by joining intervals from one set to those of another whenever their end points meet. (One endpoint must be closed and the other open [4]). Hence, the right-hand side of the predicate specifies all those intervals where after 0.1 seconds, v equals u. The entire predicate, therefore, states (using $\subseteq$) that all intervals where u is greater than 10, are also intervals where, after 0.1 seconds, v equals u.

As an example of Real-Time Object-Z, consider specifying a speedometer (based on that specified in [8]) which calculates the speed of a vehicle by detecting the rotation of one of its wheels: the speed is calculated by dividing the wheel circumference by the time taken for a single rotation.

Let the wheel circumference be 3 metres

$$\mathit{wheel\_circum} == 3 \qquad\qquad -\text{metres}$$

and assume a maximum speed of 60 metres per second (216 km/hr).

$$\mathit{MaxSpeed} == 60 \qquad\qquad -\text{metres per second}$$

The speed output by the speedometer is a natural number between 0 and *MaxSpeed*.

$$\mathit{Speed} == 0 \mathinner{..} \mathit{MaxSpeed} \qquad\qquad -\text{metres per second}$$

The complete specification of the speedometer is provided by the following class.

Speedometer

$$wheel\_angle? : \mathbb{T} \rightsquigarrow \mathbb{R}$$

$$\begin{array}{l} last\_calculation : \mathbb{T} \\ speed! : Speed \end{array}$$

INIT

$$\begin{array}{l} last\_calculation < \tau - 2 * wheel\_circum \\ speed! = 0 \end{array}$$

CalculateSpeed

$$\Delta(last\_calculation, speed!)$$

$$\begin{array}{l} wheel\_angle?(\tau) \bmod 2\pi = 0 \\ \forall\, t : [\tau \ldots \tau'] \bullet wheel\_angle?(t) \bmod 2\pi \neq 0 \\ last\_calculation' = \tau \\ speed!' = wheel\_circum/(\tau - last\_calculation) \pm 0.5 \end{array}$$

$$\langle |\, \underline{s}\ wheel\_angle? \,| \leqslant 2\pi * MaxSpeed/wheel\_circum \rangle = \langle \text{true} \rangle$$

$$\begin{array}{l} \langle wheel\_angle? \bmod 2\pi = 0 \rangle \mathbin{;} \langle wheel\_angle? \bmod 2\pi \neq 0 \rangle \subseteq \\ \qquad \langle \text{true} \rangle \mathbin{;} \langle CalculateSpeed \rangle \mathbin{;} \langle \text{true} \rangle \end{array}$$

The speedometer's environment includes a continuous variable representing the angle of the wheel in radians from some fixed position (*wheel_angle?*). The use of a constant giving the wheel's value over all time is consistent with the style of specification used in TIC. The "?" decoration on the name is used to indicate that it is an environmental variable that acts as an input to the specified system. Similarly, environmental variables decorated with "!" act as outputs from the system.

The speedometer calculates the speed (*speed!*) from the wheel circumference and the wheel angle. To do this it keeps track of the time of the last speed calculation in a state variable *last_calculation*. Initially, this variable is set to a time more than $2 * wheel\_circum$ seconds before the current time τ. This ensures that the first speed calculation, when the wheel starts rotating, will be zero (since the calculated speed is a natural number with units metres per second and a wheel rotation time of more than $2 * wheel\_circum$ corresponds to a speed of less than 0.5 metres per second). Ensuring the first speed calculation is zero is necessary because the wheel may not undergo a full rotation before it occurs.

The operation *CalculateSpeed* calculates the speed to the nearest natural number based on the wheel circumference and the time since the last calculation. As in Object-Z, operations include a delta-list (i.e., a list of the form $\Delta(\ldots)$) of variables which they are able to change (in this case both *last_calculation* and *speed*!), and denote post-state variables using primes, e.g., $last\_calculation'$.

CalculateSpeed is enabled each time the wheel passes the point corresponding to a multiple of 2π radians. The first two predicates of the operation ensure that the wheel angle mod 2π is 0 only for the first time instant of the operation. This prevents implementations where the wheel completes an entire rotation before *CalculateSpeed* has finished executing. (Note that intervals of real numbers can be specified using combinations of the brackets [ ] for closed intervals and ( ) for open intervals.)

The latter constraint is feasible since the class has an assumption predicate (above the short horizontal line at the bottom of the class) which limits the rate of change of *wheel_angle*? ($\underline{s}\ v$ denotes the derivative of a differentiable variable v [3]). This assumption also ensures that the speed calculated by the final predicate of *CalculateSpeed* is less than or equal to *MaxSpeed*. (Note that $\langle \text{true} \rangle$ denotes the set of all possible intervals.)

To ensure that *CalculateSpeed* occurs every time the wheel passes the point corresponding to 0 radians, the class also has an effect predicate (below the short horizontal line at the bottom of the class) which states that *CalculateSpeed* occurs in a sub-interval of any interval where the wheel angle mod 2π is 0, and then becomes non-zero.

2.1 Semantics

The semantics of Real-Time Object-Z [17] is given in terms of an extension to the history semantics of Object-Z [13, §2.3]. In the Object-Z semantics, a history of a class is a possible sequence of states an instance of the class can pass through, together with the associated sequence of operations that cause the state changes. A state is an assignment of values to a set of identifiers representing its variables and the constants it can refer to. The states S of a class are hence defined as

$$S \subseteq Id \twoheadrightarrow\!\!\!\!\!\!\shortmid\;\; Value$$

An operation comprises the operation's name and an assignment of values to the operations parameters. The operations O of a class are defined as

$$O \subseteq Id \times (Id \twoheadrightarrow\!\!\!\!\!+ \; Value)$$

Therefore, the set of histories of a class is represented by a set [3]

$$H \subseteq S^{\omega} \times O^{\omega}$$

such that

$$(s, o) \in H \Rightarrow s \neq \langle\,\rangle \tag{H1}$$
$$(s, o) \in H \wedge s \in S^{*} \Rightarrow \#s = \#o + 1 \tag{H2}$$
$$(s, o) \in H \wedge s \notin S^{*} \Rightarrow o \notin O^{*} \tag{H3}$$
$$(s_1 \frown s_2, o_1 \frown o_2) \in H \wedge \#s_1 = \#o_1 + 1 \Rightarrow (s_1, o_1) \in H \tag{H4}$$

These properties capture the fact that the sequence of states is non-empty (H1) and is one longer than the sequence of operations (H2) (except when both are infinite (H3)), and that the set of histories is prefix-closed (H4).

The semantics of Real-Time Object-Z models a class as a set of *real-time histories*. A real-time history extends a standard Object-Z history with

- start and end times of each operation,
- timed trace representations of all constants and variables, and
- a set of time intervals for each operation denoting the operation occurrences.

Since the latter can be derived from the start and end times of operations [17], we do not need to include them explicitly as part of the semantics. Similarly, since the timed trace representation of constants and variables can be derived from the sequence of states [17] [4], we do not need to explicitly include them either.

The start times are represented by a sequence of times equal in length to the number of operations (or infinite when the number of operations are infinite). Similarly, the end times are represented by a sequence of times. The first end time denotes the time at which initialisation occurred. Hence, the length of the sequence is one greater than the number of start times (or infinite when the number of start times is infinite).

Therefore, the real-time histories of a class can be represented by a set

$$R \subseteq S^{\omega} \times O^{\omega} \times \mathbb{T}^{\omega} \times \mathbb{T}^{\omega}$$

[3] S^{ω} and S^{*} denote the set of (possibly infinite) sequences and set of finite sequences, respectively, of elements from the set S.

[4] Note that since continuous variables are modelled as constants, their value (over all time) is available in any state.

such that

$$
\begin{aligned}
&(s, o, t_s, t_e) \in R \Rightarrow \\
&\qquad s \neq \langle\,\rangle \wedge (\forall\, i \in 1 \mathinner{..} \#t_s \bullet t_e(i) \leqslant t_s(i) \leqslant t_e(i+1)) && \text{(R1)}\\
&(s, o, t_s, t_e) \in R \wedge s \in S^* \Rightarrow \#s = \#o + 1 = \#t_s + 1 = \#t_e && \text{(R2)}\\
&(s, o, t_s, t_e) \in R \wedge s \notin S^* \Rightarrow o \notin O^* \wedge t_s \notin \mathbb{T}^* \wedge t_e \notin \mathbb{T}^* && \text{(R3)}\\
&(s_1 \frown s_2, o_1 \frown o_2, t_{s1} \frown t_{s2}, t_{e1} \frown t_{e2}) \in R \\
&\qquad \wedge \#s_1 = \#o_1 + 1 = \#t_{s1} + 1 = \#t_{e1} \Rightarrow (s_1, o_1, t_{s1}, t_{e1}) \in R && \text{(R4)}
\end{aligned}
$$

These properties extend those for standard Object-Z histories so that there is an appropriate ordering on start and end times of operations (R1), and the sequences of start and end times are of the same length as the sequence of operations, and one more than the sequence of operations, respectively (R2) (except when each of the sequences are infinite (R3)).

Given a function $\mathcal{R}_{OZ}$ mapping Object-Z classes to sets of real-time histories, and a function $\mathcal{R}_{TIC}$ mapping TIC predicates to real-time histories (see [17] for details), the set of real-time histories of a Real-Time Object-Z class C with Object-Z part O, assumption predicate A and effect predicate E is given by

$$\mathcal{R}(C) = \{h \mid h \in \mathcal{R}_{TIC}(A) \Rightarrow h \in \mathcal{R}_{TIC}(E) \cap \mathcal{R}_{OZ}(O))\}$$

2.1.1 Refinement

Refinement in Real-Time Object-Z can be performed by refining the Object-Z and timed trace parts of the class separately according to the rules of refinement of their respective notations. No new rules need to be developed.

For Object-Z, refinement is achieved by strengthening the initial condition and/or the postconditions of operations (modulo the retrieve relation). Preconditions of operations cannot be weakened, as in Z refinement [18], due to Object-Z's *blocking* interpretation of operations [15]. Under this interpretation operations cannot occur when their preconditions are not satisfied. In Z they can occur when their preconditions are not satisfied resulting in an undefined post-state.

Refinement in TIC consists of weakening of the assumptions and strengthening of the effects of specifications [4]. In Real-Time Object-Z, this strengthening and weakening of the TIC predicates is performed in the context of the class's operation definitions. This is necessary so that Boolean variables representing operations in the TIC predicates can be related to the environmental and local variables the operations access and modify.

Refining the Object-Z part O of a class C restricts the possible post-states

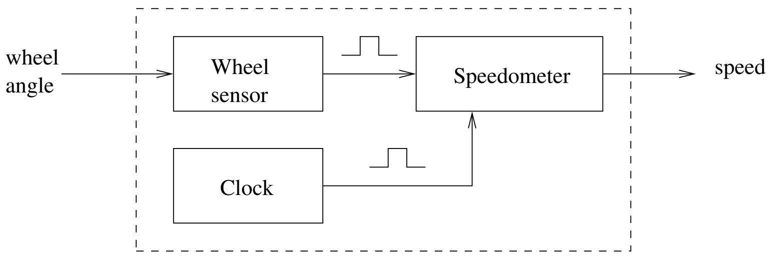

Fig. 1. Speedometer design

of operation occurrences (modulo the retrieve relation) and hence the possible real-time histories of the class, $\mathcal{R}_{OZ}(O)$. Similarly, refining the TIC part of a class restricts the real-time histories of the effect E, $\mathcal{R}_{TIC}(E)$, and/or increases the real-time histories of the assumption A, $\mathcal{R}_{TIC}(A)$. The overall effect of refining either part of a class, therefore, is to restrict the real-time histories given by $\mathcal{R}(C)$.

3 Objects in Real-Time Object-Z

Assume we want to implement the speedometer of Section 2 in terms of the three components in Figure 1. That is, a wheel sensor sends a pulse to the main speedometer component whenever the wheel completes a rotation. The latter component counts pulses from a clock component in order to calculate the speed.

Expressing such a design in an object-oriented fashion requires the use of object instantiation. Given a global constant

$$\mathit{freq} == 10^6 \qquad\qquad -\text{hertz}$$

denoting the clock frequency, the *Speedometer* class might, for example, be expressed in terms of instances of *Clock* and *WheelSensor* classes as follows.

Speedometer

$clock : Clock$
$sensor : WheelSensor$
$count : \mathbb{N}$
$speed! : Speed$

Init

$count > 2 * wheel\_circum * freq$
$clock.\mathit{Init}$

IncCount

$\Delta(count)$

$count' = count + 1$

$Count \mathrel{\widehat=} clock.ClockPulse \land IncCount$

NewSpeed

$\Delta(count, speed!)$

$count' = 0$
$speed!' = wheel\_circum/(count/freq) \pm 0.5$

$CalculateSpeed \mathrel{\widehat=} sensor.WheelPulse \land NewSpeed$

true

true

The class has an operation *Count* which conjoins two operations to increment the count (*IncCount*) whenever the object *clock* performs a *ClockPulse* operation (*clock*.*ClockPulse*). Similarly, *CalculateSpeed* conjoins an operation to calculate the new speed (*NewSpeed*) whenever the object *sensor* performs a *WheelPulse* operation (*sensor*.*WheelPulse*). Initially, the count is set to a large enough value so that the first speed calculation, when the wheel starts rotating, will be zero. Also, the object *clock* is initialised. The details of class *Clock* and *WheelSensor* will be seen in Section 4. Note that there are no TIC predicates in *Speedometer* as the timing constraints appear in the other classes.

Unlike Object-Z, however, the current definition of Real-Time Object-Z does not support object instantiation. In Object-Z the type of an object is the set of histories of its class. That is, the value of an object at any instant is a single history denoting the operations the object has undergone and the states it has passed through.

To reuse this existing semantics, we can also represent objects in Real-Time Object-Z by Object-Z histories. However, to also capture their real-time behaviour, we need to embed the operation start and end times within these histories. This can be done by adding state variables $\tau_s : \mathbb{T}$ and $\tau_e : \mathbb{T}$ to a class's histories denoting the times the last operation started and ended respectively. Initially, the variable τ_s is set to 0. That is, we define the set of histories associated with a Real-Time Object-Z class C as follows.

$$
\begin{aligned}
\mathcal{H}(C) = \{h \mid \exists\, r \in \mathcal{R}(C) \bullet\ & \\
& \mathit{states}(h)(1) = \mathit{states}(r)(1) \\
& \qquad \cup \{\tau_s \mapsto 0\} \\
& \qquad \cup \{\tau_e \mapsto \mathit{end\_times}(r)(1)\} \wedge \\
& (\forall\, n \in \operatorname{dom} \mathit{ops}(r) \bullet \\
& \quad \mathit{states}(h)(n+1) = \mathit{states}(r)(n+1) \\
& \qquad \cup \{\tau_s \mapsto \mathit{start\_times}(r)(n)\} \\
& \qquad \cup \{\tau_e \mapsto \mathit{end\_times}(r)(n+1)\}) \wedge \\
& \mathit{ops}(h) = \mathit{ops}(r)\}
\end{aligned}
$$

where given a history $h = s \times o$, $\mathit{states}(h) = s$ and $\mathit{ops}(h) = o$, and given a real-time history $r = s \times o \times t_s \times t_e$, $\mathit{states}(r) = s$, $\mathit{ops}(r) = o$, $\mathit{start\_times}(r) = t_s$ and $\mathit{end\_times}(r) = t_e$.

3.1 Operation synchronisation

Given the above semantics of objects, the issue arises of how to synchronise operations of an object with those in the class declaring the object. For example, in the *Speedometer* class above, we want the *IncCount* operation to occur whenever the *ClockPulse* operation of *clock* occurs. Simply conjoining the operations does not ensure this as the τ variables are local to their respective classes.

One approach to ensure synchronisation would be to equate the operations' start and end times, i.e., to equate the values of τ_s and τ_e embedded in the history representing *clock* to the values of τ and τ' of the operation *IncCount*. This is overly restrictive, however, forcing the operations from *Speedometer* and *Clock* to have identical start and end times. In general, synchronising operations may simply overlap.

To ensure operations overlap (but do not necessarily share start or end times) we introduce an implicit output variable $t! : \mathbb{T}$ to every operation and an implicit predicate $\tau \leqslant t! \leqslant \tau'$. Since any conjoined operations need to agree on $t!$ they will need to have at least one time in common between their start and end times.

In particular, it should be noted that an operation defined as

$$Op1 \mathrel{\widehat=} a.Op$$

where a is an object, is semantically equivalent to

$$Op1 \mathrel{\widehat=} [\, t! : \mathbb{T} \mid \tau \leqslant t! \leqslant \tau' \,] \wedge a.Op$$

Therefore, $Op1$ and $a.Op$ overlap in time only (through the shared output $t!$). They do not necessarily have the same start and end times.

Introducing such a shared output variable results in a known problem with compositionality and refinement in Object-Z [11]. A means of dealing with this is discussed in Section 5.

4 Refactoring

McComb and Smith [12] present a strategy for using the existing theory of class refinement in Object-Z [2] to introduce an arbitrary number of object instances into a specification. Since class refinement applies only to a single class, the key part of the strategy is to define a single class which is equivalent to a system of interacting classes. That is, the strategy creates one class which has all the features of every class in the system, and so can act as the class of every object. The system structure is then set up using self referencing to the class.

Consider the following refinement of the *Speedometer* class of Section 2 [5]. We have introduced details of the sensor and clock components we wish to introduce in order to refine to the design of Figure 1.

Speedometer

$wheel\_angle? : \mathbb{T} \leadsto \mathbb{R}$

$count : \mathbb{N}$
$speed! : Speed$

INIT
$count > 2 * wheel\_circum * freq$
$speed! = 0$

ClockPulse
$\tau' < \tau + 1/freq$

IncCount
$\Delta(count)$
$count' = count + 1$

$$Count \mathrel{\widehat=} ClockPulse \wedge IncCount$$

[5] We take the view that a class's interface can be widened under refinement [9].

WheelPulse

$$wheel\_angle?(\tau) \bmod 2\pi = 0$$
$$\forall t : [\tau \ldots \tau'] \bullet wheel\_angle?(t) \bmod 2\pi \neq 0$$

NewSpeed

$$\Delta(count, speed!)$$

$$count' = 0$$
$$speed!' = wheel\_circum/(count/freq) \pm 0.5$$

$$CalculateSpeed \mathrel{\widehat=} WheelPulse \land NewSpeed$$

$$\langle | \underline{s}\ wheel\_angle? | \leqslant 2\pi * MaxSpeed/wheel\_circum\rangle = \langle \mathrm{true}\rangle$$

$$\langle wheel\_angle? \bmod 2\pi = 0\rangle \mathbin{;} \langle wheel\_angle? \bmod 2\pi \neq 0\rangle \subseteq$$
$$\quad \langle \mathrm{true}\rangle \mathbin{;} \langle WheelPulse\rangle \mathbin{;} \langle \mathrm{true}\rangle$$
$$\langle \delta > 1/freq\rangle \subseteq \langle \mathrm{true}\rangle \mathbin{;} \langle ClockPulse\rangle \mathbin{;} \langle \mathrm{true}\rangle$$

The new operation *ClockPulse* has a duration of less than $1/freq$. Together with the new effect predicate that states that this operation occurs in every interval of length greater than $1/freq$, we can infer that the operation occurs with the desired frequency. Since the original class calculated the speed to the nearest 0.5 metres/second, the above class is a refinement provided that the clock frequency is sufficient to allow at least $MaxSpeed/0.5$ clock pulses in the minimum possible wheel rotation time $wheel\_circum/MaxSpeed$. The frequency of 1MHz is sufficient to ensure this.

Following the ideas in [12], we create objects of class *Speedometer* which will eventually be our clock and wheel sensor objects. A simple data refinement changes the state of *Speedometer* to the following (where the ellipses are placeholders for class features that have not changed since the previous definition of *Speedometer*).

Speedometer

...

$$clock : Speedometer_{©}$$
$$sensor : Speedometer_{©}$$
$$count : \mathbb{N}$$
$$speed! : Speed$$

...

The use of Object-Z's containment operator, denoted by a subscript ©, disallows any recursion [15], such that an object of class *Speedometer* cannot refer to itself via *sensor* or *clock*. In fact, the semantics of the containment operator is more general: it also prevents aliasing of objects of the type $\mathit{Speedometer}_{©}$ from all parts of the specification. We further assume aliasing with objects of type *Speedometer* is not possible when we apply this rule.

We then migrate some of the behaviour of our class to the new objects. In Object-Z, given an operation P and another operation S which uses P, e.g., $S \mathrel{\widehat=} P \wedge T$, we can migrate P to such an object a by changing S to

$$S \mathrel{\widehat=} a.P \wedge T$$

This will be a refinement under the retrieve relation R which maps each variable x which is free in P to $a.x$. This follows from the fact that the applicability and correctness simulation rules for Object-Z [2] hold:

Applicability: $R \Rightarrow (\mathrm{pre}\, P \Rightarrow \mathrm{pre}\, a.P)$
Correctness: $R \wedge a.P \Rightarrow (\exists\, s' \bullet R' \wedge P)$

where s' denotes the post-state abstract state variables.

In Real-Time Object-Z, the situation is not as straightforward. If the start or end time of the operation P, and hence S [6], are precisely defined, for example, then the migration to object a will remove this constraint from S. The constraint will be part of $a.P$ and, under the semantics of synchronisation of operations, S need only share a time with $a.P$ (not necessarily its start and end times). Hence, we will not have a refinement, since there will be more possible times that S can occur and hence more, rather than less, real-time histories.

A similar problem arises when the duration of P, and hence S, is precisely defined. In general, to determine whether an operation migration results in a refinement, we need to consider all timing constraints on the operation, including those in the operation's TIC predicates. We need to ensure that the possible start and end times for S after the migration are subsets of those before. Then the set of real-time histories of the class will also be a subset and we will have a refinement.

One way to ensure refinement is to duplicate timing constraints on the migrated operation. For example, returning to our case study, we migrate the operation *ClockPulse* to the object *Clock* as follows.

[6] Since P and S refer to the same τ and τ', they will necessarily have the same start and end times.

Speedometer

$\ldots$

IncCount

$\Delta(count)$

$count' = count + 1$
$\tau' < \tau + 1/freq$

$Count \mathrel{\widehat{=}} clock.ClockPulse \land IncCount$

$\ldots$

The constraint on the duration of *Count* (through *ClockPulse*) is maintained after the migration by its explicit duplication in *IncCount*. Note that there is no need to duplicate the effect predicate which states that *ClockPulse* occurs in every interval of $1/freq$ seconds. Since *clock*.*ClockPulse* is constrained to occur in all such intervals and its duration is less than $1/freq$ seconds, to ensure synchronisation *Count* must also occur in all such intervals.

Similarly, we migrate the operation *WheelPulse* to the *sensor* object.

Speedometer

$\ldots$

NewSpeed

$\Delta(count, speed!)$

$count' = 0$
$speed!' = wheel\_circum/(count/freq) \pm 0.5$
$\forall\, t : [\tau \ldots \tau'] \bullet sensor.wheel\_angle?(t) \bmod 2\pi \neq 0$

$CalculateSpeed \mathrel{\widehat{=}} sensor.WheelPulse \land NewSpeed$

$\ldots$

Again we have a refinement due to the explicit duplication of the predicate which constrains when *WheelPulse* can be occurring in *NewSpeed*. Note that the duplicated predicate refers to the *wheel_angle*? input of *sensor* (since this is the one that is used to constrain the timing of *sensor*.*WheelPulse*). The effect predicate on *sensor*.*WheelPulse* and the synchronisation of this operation with *CalculateSpeed* ensures the original constraints this effect predicate had on *CalculateSpeed* (through *WheelPulse*).

Following the approach of [12], we next introduce two new classes, *Clock* and *WheelSensor*, which are exact copies of *Speedometer*. We then change the types of the objects *clock* and *sensor* in *Speedometer* to be $Clock_{©}$ and $WheelSensor_{©}$ respectively. This change is semantics-preserving since the classes *Speedometer*, *Clock* and *WheelSensor* are syntactically, and hence semantically, identical. Furthermore, we do not have recursion of aliasing as discussed previously.

Speedometer

...

$clock : Clock_{\copyright}$
$sensor : WheelSensor_{\copyright}$
$count : \mathbb{N}$
$speed! : Speed$

...

Clock

...

$clock : Speedometer_{\copyright}$
$sensor : Speedometer_{\copyright}$
$count : \mathbb{N}$
$speed! : Speed$

...

WheelSensor

...

$clock : Speedometer_{\copyright}$
$sensor : Speedometer_{\copyright}$
$count : \mathbb{N}$
$speed! : Speed$

...

We are now in a position to significantly simplify all three class definitions. In *Clock* and *WheelSensor*, we can remove features which are not referenced. Since these classes were introduced by our process the only class referencing them will be *Speedometer*. Hence, the removal of the features will have no effect on the semantics of the specification. In *Speedometer*, we can remove operations which we introduced in the orignal refinement and which have been migrated to *Clock* and *WheelPulse*. Since the specification did not reference these before we began the refactoring process, their removal will again have no effect on its semantics. In addition, below we assume that the origial specification did not access the enviornmental input $wheel\_angle?$ of *Speedometer* allowing its removal as well.

Through these simplification we see how the classes reflect their intended purpose. The class *Speedometer* interacts with objects of classes *Clock* and *WheelSensor* to output a value for the current speed.

Speedometer

$$clock : Clock_{©}$$
$$sensor : WheelSensor_{©}$$
$$count : \mathbb{N}$$
$$speed! : Speed$$

Init

$$count > 2 * wheel\_circum * freq$$
$$speed! = 0$$

IncCount

$$\Delta(count)$$

$$count' = count + 1$$
$$\tau' < \tau + 1/freq$$

$$Count \mathrel{\widehat{=}} clock.ClockPulse \land IncCount$$

NewSpeed

$$\Delta(count, speed!)$$

$$count' = 0$$
$$speed!' = wheel\_circum/(count/freq) \pm 0.5$$
$$\forall\, t : [\tau \ldots \tau'] \bullet sensor.wheel\_angle?(t) \bmod 2\pi \neq 0$$

$$CalculateSpeed \mathrel{\widehat{=}} sensor.WheelPulse \land NewSpeed$$

$$true$$

$$true$$

The class *Clock* produces a pulse at a frequency of 1MHz

Clock

ClockPulse

$$\tau' < \tau + 1/\mathit{freq}$$

$$\text{true}$$

$$\langle \delta > 1/\mathit{freq} \rangle \subseteq \langle \text{true} \rangle \; ; \; \langle \mathit{ClockPulse} \rangle \; ; \; \langle \text{true} \rangle$$

and the class *WheelSensor* produces a pulse for each rotation of the wheel.

WheelSensor

$$\mathit{wheel\_angle}? : \mathbb{T} \rightsquigarrow \mathbb{R}$$

WheelPulse

$$\mathit{wheel\_angle}?(\tau) \bmod 2\pi = 0$$
$$\forall\, t : [\tau \ldots \tau'] \bullet \mathit{wheel\_angle}?(t) \bmod 2\pi \neq 0$$

$$\langle \left| \underline{s}\; \mathit{wheel\_angle}? \right| \leqslant 2\pi * \mathit{MaxSpeed} / \mathit{wheel\_circum} \rangle = \langle \text{true} \rangle$$

$$\langle \mathit{wheel\_angle}? \bmod 2\pi = 0 \rangle \; ; \; \langle \mathit{wheel\_angle}? \bmod 2\pi \neq 0 \rangle \subseteq \langle \text{true} \rangle \; ; \; \langle \mathit{WheelPulse} \rangle \; ; \; \langle \text{true} \rangle$$

5 Refinement

Given the semantics of operation synchronisation in Section 3, the specification of the previous section ensures that the operation *Count* of *Speedometer* overlaps in time with the operation *ClockPulse* of the object *clock*. Similarly, the operation *CalculateSpeed* of *Speedometer* overlaps in time with the operation *WheelPulse* of the object *sensor*. This notion of overlapping is very loose, however, and does not precisely capture the timing or causality of the synchronisations. For example, the *ClockPulse* operation of the object *clock* needs to occur first and trigger the *Count* operation of *Speedometer*.

We would like to be able to refine the specification, therefore, by placing more precise constraints in each class on the shared output variable $t!$. However, such refinements involving shared output varibles are potentially non-compositional. That is, refining one class in isolation does not guarantee the refinement of the whole specification. A method of dealing with this problem in Object-Z is presented in [11]. The method involves an equivalence transformation of the specification to one in which introduced constraints prohibit non-compositional refinements. We illustrate the method for the operation *Count*.

The first step of the process is to equate the shared output variables of the conjoined operations to fresh input variables. The only shared output variable of *Count* is the implicitly defined $t!$, which is implicitly constrained in each operation to be $\tau \leqslant t! \leqslant \tau'$ (refer to Section 3.1). Thus, we redefine *ClockPulse* and *IncCount* to include a fresh input variable $overlap? : \mathbb{T}$ and a constraint $t! = overlap?$:

Clock

. . .

ClockPulse

$overlap? : \mathbb{T}$

$\tau' < \tau + 1/freq$
$t! = overlap?$

. . .

Speedometer

. . .

$sensor : WheelSensor_{©}$
$clock : Clock_{1©}$
$count : \mathbb{N}$
$speed! : Speed$

IncCount

$\Delta(count)$
$overlap? : \mathbb{T}$

$\tau' < \tau + 1/freq$
$count' = count + 1$
$t! = overlap?$

$Count \mathrel{\widehat{=}} (clock.ClockPulse \wedge IncCount) \setminus \{overlap?\}$

. . .

The hiding of $overlap?$ in *Count* makes this step an equivalence transformation [11].

The next step is to introduce a *c*oupling operation that will allow us to perform refinements determining the choice of $overlap?$, i.e., a time at which the operations must overlap. This operation, *CountTime*, is defined as follows:

$$CountTime \mathrel{\widehat{=}} [\,\mathrm{pre}\; clock.ClockPulse \wedge \mathrm{pre}\; IncCount\,][overlap!/overlap?],$$

where [7]

$$\text{pre}\, clock.ClockPulse \equiv freq \neq 0 \wedge clock.\tau \leqslant overlap? < clock.\tau + 1/freq$$

and

$$\text{pre}\, IncCount \equiv \tau \leqslant overlap?.$$

assuming all pre-state variables are at least within their types. Given this definition, the following is semantically equivalent to our original *Speedometer* class [11].

Speedometer

$$\begin{array}{l}
\ldots \\
\quad CountTime \\
\quad\quad overlap! : \mathbb{T} \\
\quad\quad \hline \\
\quad\quad freq \neq 0 \\
\quad\quad clock.\tau \leqslant overlap! < clock.\tau + 1/freq \\
\quad\quad \tau \leqslant overlap! \\
Count \mathrel{\widehat=} CountTime \parallel (clock.ClockPulse \wedge IncCount) \\
\ldots
\end{array}$$

CountTime communicates the value of the variable *overlap*! to *ClockPulse* and *IncCount* through the Object-Z parallel composition operator. In Object-Z, the parallel operator ($\parallel$) unifies output variables with input variables where the basename is shared, and hides those variables. In this case *overlap*! is unified with *overlap*? in the latter operations and hidden.

The introduction of the operation *CountTime* into the specification of the *Speedometer* class does not complete the compositional decoupling of the *Count* operation, because in this case we have introduced a direct reference to the state variable τ in the *Clock* class from the *Speedometer* class. To circumvent this, as with all references to foreign state variables, we introduce an accessor operation (in this case *GetOpStartTime*):

Clock

$$\begin{array}{l}
\ldots \\
GetOpStartTime \mathrel{\widehat=} [\, \tau! : \mathbb{T} \mid \tau! = \tau \,] \\
\ldots
\end{array}$$

[7] In Object-Z the precondition is defined by existentially quantifying the post-state variables (outputs included) over the predicate of the operation [2].

Speedometer

$\ldots$

$sensor : WheelSensor_{©}$
$clock : Clock_{©}$
$count : \mathbb{N}$
$speed! : Speed$

CountTime

$overlap!, last\_clock?, \tau? : \mathbb{T}$

$freq \neq 0$
$\tau? \leqslant overlap! < \tau? + 1/freq$
$\tau \leqslant overlap!$

$$\begin{aligned}Count \mathrel{\widehat{=}} & \ clock.GetOpStartTime \\ & \parallel CountTime \\ & \parallel (clock.ClockPulse \wedge IncCount)\end{aligned}$$

$\ldots$

5.1 Example Refinement

Given the compositional decoupling of the classes above, we can now refine the way in which the synchronisation in the operation *Count* take place. For example, we could refine the specification such that *IncCount* is triggered by the falling edge of the clock pulse. Assuming the end time of the *ClockPulse* operation coincides with its falling edge, we need to refine *ClockPulse* to equate τ' with $overlap?$

ClockPulse

$\Delta(last\_clock)$
$overlap? : \mathbb{T}$

$\tau' < \tau + 1/freq$
$last\_clock' = \tau$
$t! = overlap?$
$\tau' = overlap?$

and *CountTime* to equate $overlap!$ with τ.

CountTime

$overlap!, last\_clock?, \tau? : \mathbb{T}$

$freq \neq 0$
$\tau? \leqslant overlap! < \tau? + 1/freq$
$\tau = overlap!$

6 Conclusion

This paper has presented an approach for refactoring Real-Time Object-Z specifications. This allows design issues to be ignored at a high level of abstraction and introduced via a sequence of refinement steps. The approach was illustrated by refactoring a speedometer specification to include a clock and a wheel sensor component. Additionally, it was illustrated how further refinements can be used to introduce the precise timing and causality of synchronisations between components.

A complete design methodology, however, also needs to account for the fact that it is often desirable for initial specifications of real-time systems to be 'ideal' and only approximate the final implementation. For example, a specification which sets a variable v to be equal to a continuous environmental variable u is not implementable. The reading of variable u (by a sensor) would necessarily include some finite error (due to both quantization error and time delay). Hence, the best we could hope for is that $v = u \pm e$ for some small e.

However, including such errors in an initial abstract specification distracts the specifier from the essential functionality of the system and complicates formal analysis. Hence, some means of incrementally introducing such details of the physical implementation is required. This issue is tackled for Timed Interval Calculus (TIC) specifications in [14,16]. The approach is to interleave so-called *realisation* steps with refinement steps. A complete set of realisation rules for TIC is presented in [14] and applied to a non-trivial case study in [16]. Adapting these rules for Real-Time Object-Z is an area of possible future work.

Acknowledgement

We would like to acknowledge the support of Australian Research Council (ARC) Discovery Grant, DP0558408.

References

[1] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornelio. Algebraic Reasoning for Object-Oriented Programming. *Sci. Comput. Program.*, 52(1-3):53–100, 2004.

[2] J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* FACIT Series. Springer-Verlag, 2001.

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

[4] C.J. Fidge, I.J. Hayes, A.P. Martin, and A.K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In J. Jeuring, editor, *Mathematics of Program Construction*

(MPC'98), volume 1422 of *Lecture Notes in Computer Science*, pages 188–206. Springer-Verlag, 1998.

[5] R. Gheyi and P. Borba. Refactoring Alloy specifications. *Electronic Notes in Theoretical Computer Science*, 95:227–243, 2004.

[6] K. Lano. *Formal Object-oriented Development*. Springer-Verlag, 1995.

[7] K. Lano and S. Goldsack. Refinement of Distributed Object Systems. In E. Najm and J.-B. Stefani, editors, *Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 99–114. Chapman and Hall, March 1996.

[8] B.P. Mahony and J.S. Dong. Sensors and actuators in TCOZ. In J. Wing, J.C.P. Woodcock, and J. Davies, editors, *World Congress on Formal Methods (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1166–1185. Springer-Verlag, 1999.

[9] T. McComb. Refactoring Object-Z Specifications. In M. Wermelinger and T. Margaria-Steffen, editors, *FASE '04: Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2004.

[10] T. McComb and G. Smith. Architectural Design in Object-Z. In P. Strooper, editor, *ASWEC '04: Australian Software Engineering Conference*, pages 77–86. IEEE Computer Society Press, 2004.

[11] T. McComb and G. Smith. Compositional class refinement in Object-Z. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2006.

[12] T. McComb and G. Smith. Introducing objects through refinement. In *FM 2008: Formal Methods*, Lecture Notes in Computer Science. Springer, 2008.

[13] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[14] G. Smith. Stepwise development from ideal specifications. In J. Edwards, editor, *Australasian Computer Science Conference (ACSC 2000)*, volume 22 of *Australian Computer Science Communications*, pages 227–233. IEEE Computer Society, 2000.

[15] G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

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

[17] G. Smith and I.J. Hayes. An introduction to Real-Time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.

[18] J. Woodcock and J. Davies. *Using Z: Specification, refinement, and proof.* Prentice Hall, 1996.