Structuring Real-Time Object-Z Specifications

Graeme Smith* and Ian Hayes†

*Software Verification Research Centre
†School of Computer Science and Electrical Engineering
University of Queensland, Australia

Abstract. This paper presents a means of structuring specifications in real-time Object-Z: an integration of Object-Z with the timed refinement calculus. Incremental modification of classes using inheritance and composition of classes to form multi-component systems are examined. Two approaches to the latter are considered: using Object-Z's notion of object instantiation and introducing a parallel composition operator similar to those found in process algebras. The parallel composition operator approach is both more concise and allows more general modelling of concurrency. Its incorporation into the existing semantics of real-time Object-Z is presented.

1 Introduction

Object-Z [?] is an extension of Z [?] to facilitate specification in an object-oriented style. The major extension in Object-Z is the *class schema* which captures the object-oriented notion of a class by encapsulating a single state schema, and its associated initial state schema, with all the operation schemas which may change its variables. Classes may be incrementally specified using Object-Z's notion of *inheritance* which enables definitions from one class (the inherited class) to be implicitly included in another class (the inheriting class). The enhanced structuring provided by object-oriented constructs, such as classes, and techniques, such as inheritance, significantly improve the clarity of large specifications.

In an earlier paper [?], we showed how Object-Z could be extended to model systems with continuous variables and real-time constraints. The approach was to provide a semantic basis for combining Object-Z and the real-time notation of the timed refinement calculus [?,?]. This notation allows a system to be specified by constraints over time intervals on which properties hold. However, the integrated approach, referred to as real-time Object-Z, did not utilise the structuring techniques of Object-Z provided by classes and inheritance. Hence, as presented, it is not suitable for large-scale specifications, nor for specifications comprising several components such as those of concurrent or distributed systems.

In this paper, we present an overview of real-time Object-Z (Section 2) and provide extensions to utilise Object-Z's structuring techniques. In particular, we show how inheritance can be used to incrementally modify existing class specifications (Section 3), and how different classes can be composed to form multi-component systems (Section 4). Two approaches to the latter are considered: using the object instantiation technique of Object-Z, and introducing a parallel composition operator similar to those found in process algebras. The parallel composition operator approach is both more concise and allows more general modelling of concurrency. It is also easily incorporated into the existing semantics of real-time Object-Z (Section 5).

2 Real-time Object-Z

Real-time Object-Z [?] is an integration of the timed refinement calculus [?,?] with Object-Z [?]. It differs from other approaches to specifying continuous and real-time systems in Object-Z since

- it uses only standard notation from Object-Z and the timed refinement calculus (the approaches of Friesen [?] and Mahony and Dong [?] introduce additional notation into schemas of Object-Z classes),
- it maintains Object-Z's specification style (the approach of Periyasamy and Alagar [?] requires each object to be specified by two classes: one for its functionality and one for its real-time properties), and
- it models the passing of time implicitly (the approach of Dong, et al. [?] requires an explicit *Tick* operation in each class).

2.1 Timed refinement calculus

The timed refinement calculus is a Z-based notation for the specification and refinement of real-time systems. It has been extended with a simple set-theoretic notation for concisely expressing time intervals [?] and operators for accessing interval endpoints. We adopt a simplified subset of the notation based on that of Fidge, et al. [?] which provides a minimal set of operators outside those of standard set theory.

Absolute time, $\mathbb{T}$, is modelled by real numbers and, in this paper, we will assume has the units seconds. Observable variables of a system are modelled as total functions from the time domain to a type representing the set of all values the variable may assume. A system is specified by constraints on the time intervals over which properties hold. For example, the following expresses that an observable variable $v : \mathbb{T} \rightarrow \mathbb{R}$ becomes equal to a differentiable (denoted by the function symbol $\rightsquigarrow$ [?]) observable 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[1]. 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.

[1] We adopt here a simpler notation than the brackets   used by Fidge et al. [?] and our previous paper [?].

In general, the property in the 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 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. Other reserved symbols are α and ω denoting an interval's start and end times respectively.

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 [**?**]). 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.

2.2 Integration with Object-Z

The semantic integration of the timed refinement calculus with Object-Z was presented in our previous paper [**?**]. In this section, we provide an overview of the approach including two new extensions to the syntax.

Classes in the integrated notation comprise two parts separated by a horizontal line. The part above the line is essentially the standard Object-Z local definitions and schemas. The part below the line is further constraints on the class specified in the timed refinement calculus notation. The latter is divided into an assumption and effect part as in the timed refinement calculus [**?**]. All state variables $x : X$ in the Object-Z part above the line are interpreted as timed trace variables $x : \mathbb{T} \to X$ in the timed trace part below the line.

Although all real-time properties could be specified in the timed trace part of the class, we also allow local constants and state variables of type $\mathbb{T}$ and include, in every class, an implicit state variable $\tau : \mathbb{T}$ denoting the current time. This is captured by an implicit constraint $\forall\, t : \mathbb{T} \bullet \tau(t) = t$ in the timed traced part of the class.

As an example, consider specifying a speedometer 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.

We 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*.

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

A system specified by a class in the integrated approach is a digital system and, therefore, changes to state variables only occur at discrete points in time. However, it may interact with continuously changing variables in its environment. These variables are specified as (possibly differentiable) functions of time. For example, the speedometer's environment includes a continuous variable representing the angle of the wheel in radians from some fixed position. This can be specified as follows.

$$\mathit{wheel\_angle}? : \mathbb{T} \rightsquigarrow \mathbb{R} \qquad -\text{radians}$$

Since this definition gives the values of the wheel's angle over all time, it need not be treated as a modifiable state component and can appear as a local constant in the class. The "?" decoration on the name indicates 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.

Our first extension to the syntax allows such outputs to be declared as state variables (rather than constants) to indicate that they only change value when an operation, whose Δ-list they appear in, occurs. Our second extension to the syntax allows operation names to appear as Boolean variables in the timed trace part of the class. The variable representing an operation is true in all intervals during which the operation is occurring. Examples of these features and the other features of a real-time Object-Z class are provided by the following specification of the speedometer.

Speedometer$_0$

$\mathit{wheel\_circum} == 1.5 \qquad -\text{metres}$

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

$\mathit{last\_calculation} : \mathbb{T}$
$\mathit{speed}! : \mathit{Speed}$

INIT

$\mathit{last\_calculation} < \tau - 2 * \mathit{wheel\_circum}$
$\mathit{speed}! = 0$

CalculateSpeed

$\Delta(\mathit{last\_calculation}, \mathit{speed}!)$

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

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

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

The speedometer calculates the speed (*speed*!) from the wheel circumference (*wheel_circum* $= 1.5$ metres) and the wheel angle (*wheel_angle*?) which implicitly records the number of whole revolutions of the wheel. 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. It 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 the wheel completing 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.)

This latter constraint is feasible since the class has an assumption predicate which limits the rate of change of *wheel_angle*? ($\underline{s}$ v denotes the derivative of a differentiable variable v [?]). 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 which states that *CalculateSpeed* is a sub-interval of any interval where the wheel angle mod 2π is 0, and then becomes non-zero.

Note that operations in real-time Object-Z do not have input and output parameters: all communication is performed through environmental variables such as *wheel_angle*? and *speed*!. This restriction enables a straightforward definition of refinement as shown in Section 5.

3 Inheritance

The speedometer specification of Section 2 works as we would expect when the wheel of the vehicle is rotating. If it stops rotating, however, the *CalculateSpeed* operation does not occur and so the speed output by the class is that which was last calculated.

To overcome this problems we could add an operation which detects that the wheel is no longer rotating and sets the output speed to 0. Adding an operation can be done in standard Object-Z using inheritance [?]. When an Object-Z class inherits another it implicitly includes its constants, state schema, initial state schema and operations (and may extend these definitions or add to them). We extend the notion of inheritance to also implicitly include assumption and effect predicates in the timed trace part of the inherited class. Hence, the desired modification to the speedometer can be specified as follows.

Speedometer

$Speedometer_0$

TimeOut

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

$\tau - last\_calculation > 2 * wheel\_circum$
$speed! = 0$

true

$\langle \delta \geqslant 3 * wheel\_circum \rangle \subseteq \langle \text{true} \rangle \mathbin{;} \langle TimeOut \lor CalculateSpeed \rangle \mathbin{;} \langle \text{true} \rangle$

The operation *TimeOut* is enabled when the time since the last calculation is greater than $2 * wheel\_circum$. This corresponds to a speed of less than half a metre per second (1.8 km/hr). The additional effect predicate ensures that *TimeOut* does occur before the time since the last calculation is greater than $3 * wheel\_circum$.

Semantically, inheritance in real-time Object-Z is the same as inheritance in standard Object-Z with the addition of conjoining of assumption predicates and conjoining of effect predicates from the inherited and inheriting classes. The variables and operations in the inherited assumption and effect predicates will be renamed to reflect any renaming in the inherited class [**?**].

4 Composition

To specify systems of concurrent, interacting objects, we need to be able to compose different classes. For example, consider specifying a cruise control system which is required to keep a car travelling at a desired speed set by the driver [**?**]. At any time, the driver can resume control of the car by applying the brake.

The system comprises three main components: a speedometer, a controller which accepts input from the driver, and a throttle which controls the car's speed. It is illustrated in Figure 1. (Arrows indicate the direction of information flow.)

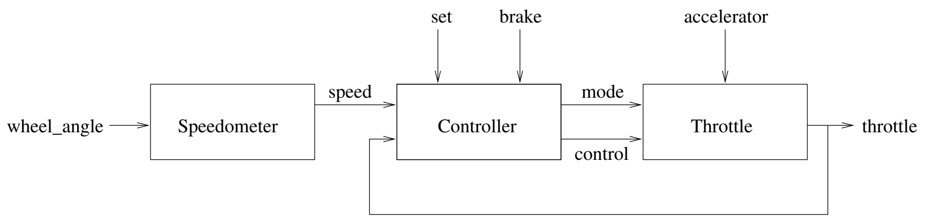

Figure 1: Cruise Control System.

In Section 4.1, we specify the classes for the controller and throttle in such a way that they can interact with each other and the speedometer of Section 3 as shown in Figure 1. In Section 4.2, we look at composing the components using standard Object-Z composition and by introducing a parallel composition operator. The semantics of this operator is provided in Section 5.

A fundamental difference between our specification and that of Mahony and Dong using TCOZ [**?**] is our use of continuous variables for modelling the wheel angle and throttle inputs from the environment. TCOZ, based on timed CSP, can only model discrete events corresponding to reading these variables and cannot model the variables themselves.

4.1 Component classes

The classes for the controller and throttle components are specified using real-time Object-Z as described in Section 2.

Controller The controller operates in two modes: *rest* when the speed of the car is being controlled by the driver, and *set_point* when a desired speed has been set by the driver and this speed is being maintained by the cruise control system.

$$Mode ::= rest \mid set\_point$$

Initially, the controller is in *rest* mode and is changed to *set_point* when the driver presses a button indicating that he or she wants the car to maintain its current speed. The controller reverts to *rest* mode when the brake is applied. While in *set_point* mode, the controller provides the throttle component with a desired value of the throttle setting. This is initially the current throttle setting and is updated periodically. The updated values of this setting are calculated from four parameters

- the current speed of the car,
- the desired speed (set by the driver),
- the speed of the car at the last calculation, and
- the current value of the throttle.

We abstractly specify this calculation by the function *desired_throttle*.

$$\mid \; desired\_throttle : Speed \times Speed \times Speed \times \mathbb{R} \rightarrow \mathbb{R}$$

The controller class is specified as follows.

Controller

$speed? : \mathbb{T} \rightarrow Speed$
$throttle? : \mathbb{T} \rightarrow \mathbb{R}$

$mode! : Mode$
$control! : \mathbb{R}$
$desired\_speed : Speed$
$previous\_speed : Speed$

Init

$mode! = rest$

Set

$\Delta(mode!, desired\_speed, previous\_speed, control!)$

$mode!' = set\_point$
$desired\_speed' = speed?(\tau)$
$previous\_speed' = speed?(\tau)$
$control! = throttle?(\tau)$

Control

$\Delta(control!, previous\_speed)$

$mode! = set\_point$
$\tau' - \tau < 0.1$
$control! = desired\_throttle(speed?(\tau), desired\_speed, previous\_speed, throttle?(\tau))$
$previous\_speed' = speed?(\tau)$

Brake

$\Delta(mode!)$

$mode!' = rest$

true

$\langle mode = set\_point \wedge \delta = 0.2 \rangle \subseteq \langle \text{true} \rangle \,;\, \langle Control \rangle \,;\, \langle \text{true} \rangle$

The timed refinement calculus predicate states that, when in *set_point* mode, the *Control* operation occurs in every 0.2 second interval. Since the duration of this operation is less than 0.1 seconds (as specified by its second predicate), this ensures that the operation occurs repeatedly with a period of less than 0.3 seconds (see Figure 2).

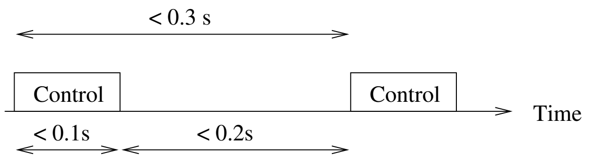

Figure 2: Occurrence of *Control*.

Throttle The throttle incrementally adjusts its output to reach a desired target output. This target depends on the mode of the controller. If its mode is *rest* the target output is equal to the input from the accelerator, otherwise it is equal to the control input from the controller. The throttle class is specified as follows.

Throttle

$$\begin{array}{l}
\mathit{throttle\_adjust} : \mathbb{R} \\
\mathit{mode?} : \mathbb{T} \to \mathit{Mode} \\
\mathit{accelerator?} : \mathbb{T} \rightsquigarrow \mathbb{R} \\
\mathit{control?} : \mathbb{T} \to \mathbb{R}
\end{array}$$

$$\mathit{throttle!} : \mathbb{T} \to \mathbb{R}$$

Init

$$\mathit{throttle!} = 0$$

UpdateThrottle

$$\Delta(\mathit{throttle!})$$

$$\begin{array}{l}
\tau' - \tau < 0.1 \\
\exists\, t : \mathbb{R} \bullet \\
\quad (\mathit{mode?}(\tau) = \mathit{rest} \Rightarrow t \in \mathit{accelerator?}(\!|\ \tau \ldots \tau'\ |\!)) \land \\
\quad (\mathit{mode?}(\tau) = \mathit{set\_point} \Rightarrow t = \mathit{control?}(\tau)) \land \\
\quad (\mathit{throttle!} < t - 0.5 * \mathit{throttle\_adjust} \Rightarrow \\
\qquad \mathit{throttle!}' = \mathit{throttle!} + \mathit{throttle\_adjust}) \land \\
\quad (\mathit{throttle!} > t + 0.5 * \mathit{throttle\_adjust} \Rightarrow \\
\qquad \mathit{throttle!}' = \mathit{throttle!} - \mathit{throttle\_adjust}) \land \\
\quad (\mathit{throttle!} \in t \pm 0.5 * \mathit{throttle\_adjust} \Rightarrow \mathit{throttle!}' = \mathit{throttle!})
\end{array}$$

true

$$\langle \delta = 0.1 \rangle \subseteq \langle \text{true} \rangle \,;\, \langle \mathit{UpdateThrottle} \rangle \,;\, \langle \text{true} \rangle$$

The timed refinement calculus predicate states that the *UpdateThrottle* operation occurs repeatedly with a period of less than 0.2 seconds.

4.2 Composing the components

When composing the components of the cruise control system we need to ensure

– that the corresponding inputs and outputs (e.g., *speed*! of *Speedometer* and *speed*? of *Controller*) are identified and equated, and
– that operations which use inputs, do not do so at a time when another component is updating the corresponding output.

The second condition is necessary since the value of a variable updated by an operation is undefined for the duration of the operation. This may be, for example, because in an implementation the value is stored as a sequence of bytes which are updated one at a time. At any time during the operation, some of the bytes may be updated and others not.

In this section we consider two approaches to composing real-time Object-Z classes: object instantiation and parallel composition.

Object instantiation In standard Object-Z, systems are composed from instances of classes called objects [**?**]. Given an object a, we can refer to a variable or constant x of the object's class by the notation $a.x$. Similarly, we can refer to the initial condition or an operation Op of the object's class by $a.\textsc{Init}$ and $a.Op$ respectively. Adopting this approach, we might specify the cruise control system as follows.

CruiseControl

$s : \mathit{Speedometer}$
$c : \mathit{Controller}$
$t : \mathit{Throttle}$

$s.\mathit{speed}! = c.\mathit{speed}?$
$c.\mathit{mode}! = t.\mathit{mode}?$
$c.\mathit{control}! = t.\mathit{control}?$
$t.\mathit{throttle}! = c.\mathit{throttle}?$

$\textsc{Init}$

$s.\textsc{Init} \land c.\textsc{Init} \land t.\textsc{Init}$

$\mathit{CalculateSpeed} \mathrel{\widehat=} s.\mathit{CalculateSpeed}$
$\mathit{TimeOut} \mathrel{\widehat=} s.\mathit{TimeOut}$
$\mathit{Set} \mathrel{\widehat=} c.\mathit{Set}$
$\mathit{Control} \mathrel{\widehat=} c.\mathit{Control}$
$\mathit{Brake} \mathrel{\widehat=} c.\mathit{Brake}$
$\mathit{UpdateThrottle} \mathrel{\widehat=} t.\mathit{UpdateThrottle}$

CruiseControl comprises an object of each component class and explicitly equates their corresponding inputs and outputs. The initial condition and operations of *CruiseControl* are constructed explicitly from those of the component classes. We assume that the real-time predicates of the component classes are implicitly maintained for each of the objects. We also assume a common τ variable for each object and *CruiseControl*.

The condition that inputs are not used when they are being updated holds automatically in this case since the semantics of real-time Object-Z does not allow operations within a single class to overlap in time [**?**] (see Appendix A). This means, however, that our system as specified exhibits no concurrency. Concurrency can be specified explicitly. For example, the concurrent occurrence of operations *CalculateSpeed* and *Brake* could be specified by adding an additional operation to *CruiseControl* of the form

$$\mathit{CalculateSpeed\&Brake} \mathrel{\widehat{=}} \mathit{CalculateSpeed} \land \mathit{Brake}$$

However, this is an undesirable approach for two reasons.

1. Explicitly stating all combinations of operations which can occur concurrently may become unwieldy for large systems comprising many components. Indeed, even the explicit identification of corresponding inputs and outputs and construction of individual operations can be verbose when using object instantiation.
2. The conjoined operations must have the same start and finish times. While this allows us to specify synchronising events, it does not allow us to specify events which partially overlap.

Parallel composition To overcome the problems with specifying concurrency, we introduce a parallel composition operator "$\|$" for classes. The idea of this operator is that it allows each component to satisfy its specified behaviour over time synchronising with other components on environmental variables with common basenames (i.e., apart from the "?" and "!") and on common-named operations. Since operations may in general overlap, if we require that two (or more) operations should be mutually exclusive in time, this needs to be explicitly specified.

One way to do this is to specify *monitor* classes which are composed with the components. A monitor class has a set of "dummy" operations (i.e., operations which perform no function). These operations comprise a subset of the total operations of the specified system which are not allowed to overlap in time. Due to the semantics of real-time Object-Z classes, these operations do not overlap within the monitor class. Due to synchronisation with the common-named operations of the component classes, these operations also cannot overlap in time.

For example, since the *Controller* operations *Set* and *Control* utilise the output *speed*! of *Speedometer*, they should not occur at times when the *Speedometer* operation *CalculateSpeed*, which changes *speed*!, occurs. The required monitor class is as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Monitor}_{SC}} \\
\hline
\mathit{CalculateSpeed} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\mathit{Set} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\mathit{Control} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\hline
\end{array}$$

Similarly, since the *Set* and *Control* operations of *Controller* utilise the output *throttle*! of *Throttle*, we need a monitor class with the operations *Set*, *Control* and *UpdateThrottle*. This class also ensures that the *mode*! and *control*! outputs of *Controller* are not updated when used by *Throttle*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Monitor}_{CT}} \\
\hline
\mathit{UpdateThrottle} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\mathit{Set} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\mathit{Control} \mathrel{\widehat{=}} [\,\text{true}\,] \\
\hline
\end{array}$$

By having two separate monitor classes we do not preclude the possibility of *CalculateSpeed* and *UpdateThrottle* occurring concurrently. The cruise control system is specified as follows.

$$\begin{aligned}
\mathit{CruiseControl} = \mathit{Speedometer} \parallel \mathit{Monitor}_{SC} \parallel \mathit{Controller} \\
\parallel \mathit{Monitor}_{CT} \parallel \mathit{Throttle}
\end{aligned}$$

The definition is both concise (due to implicit modelling of concurrency) and allows more general modelling of concurrency (by allowing partially overlapping, as well as synchronising, operations). The semantics of the parallel operator is discussed in Section 5.

5 Semantics of parallel composition

Appendix **??** gives the semantics of a real-time Object-Z class as developed in our previous work [**?**]. A class C is represented by a set of real-time histories. These consist of the signature of the class, i.e., its sets of input, output and local variables and set of operations, together with traces of the variables and the occurrences of operations. A trace is the value of a variable over time. It is represented by a total function from time to value.

$$\mathit{Trace} == \mathbb{T} \rightarrow \mathit{Value}$$

An operation is represented by an identifier corresponding to its name.

$$\mathit{Operation} == \mathit{Ident}$$

From this semantics, we extract a model of a class which we use to define refinement and parallel composition. This model separates the class's signature from its set of traces and operation occurrences. A signature of a class is specified as:

$$\begin{array}{|l}
\textit{Signature} \\
\hline
\textit{inputs}, \textit{outputs}, \textit{locals} : \mathbb{F}\,\textit{Ident} \\
\textit{opids} : \mathbb{F}\,\textit{Operation} \\
\hline
\end{array}$$

and a history as:

$$\begin{array}{|l}
\textit{RTHistory} \\
\hline
\textit{trace} : \textit{Ident} \mathrel{+\!\!\!\twoheadrightarrow} \textit{Trace} \\
\textit{occurs} : \textit{Operation} \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{P}\,\mathbb{I} \\
\hline
\end{array}$$

where $\mathbb{I}$ is the set of all time intervals.

A class is then modelled as follows.

$$\begin{array}{|l}
\textit{RTClass} \\
\hline
\textit{sig} : \textit{Signature} \\
\textit{histories} : \mathbb{P}\,\textit{RTHistory} \\
\hline
\forall\, h : \textit{histories} \bullet \\
\qquad \operatorname{dom} h.\textit{trace} = \textit{sig}.\textit{inputs} \cup \textit{sig}.\textit{outputs} \cup \textit{sig}.\textit{locals} \wedge \\
\qquad \operatorname{dom} h.\textit{occurs} = \textit{sig}.\textit{opids} \\
\hline
\end{array}$$

Since the only communication mechanism between classes is environmental variables, and environmental inputs cannot be constrained by a class [**?**], all information about when operations can be refused by a particular class are captured by its histories. Hence, a class, C, is refined by a class, D, if C and D have the same signatures and the histories of C contain the histories of D.

$$\begin{array}{|l}
\_ \sqsubseteq \_ : \textit{RTClass} \leftrightarrow \textit{RTClass} \\
\hline
C \sqsubseteq D \Leftrightarrow C.\textit{sig} = D.\textit{sig} \wedge D.\textit{histories} \subseteq C.\textit{histories}
\end{array}$$

When two classes are composed, the signature of the resulting composite class has those inputs of either class which are not also an output of the other class. That is, inputs which have the same name as an output in the other class are semantically identified with this output and no longer appear as an input to the composite class. This models the output's value being communicated to the input.

The outputs, local variables and operations of a composite class are those from either class. The outputs and operations which are common to the component classes must satisfy the constraints of both classes within the composite class. (We assume that the local variables of the component classes are distinct. Our semantics could be made more general by adding some form of renaming to ensure this, thus removing the need for this assumption.)

The binary operator **comp** composes the signatures S and T of two classes.

$$\begin{array}{|l}
\_\ \mathbf{comp}\ \_ : \mathit{Signature} \times \mathit{Signature} \rightarrowtail \mathit{Signature} \\
\hline
(S, T) \in \mathrm{dom}(\_\ \mathbf{comp}\ \_) \Leftrightarrow S.\mathit{locals} \cap T.\mathit{locals} = \varnothing \land \\
(S\ \mathbf{comp}\ T).\mathit{inputs} = (S.\mathit{inputs} \setminus T.\mathit{outputs}) \cup (T.\mathit{inputs} \setminus S.\mathit{outputs}) \land \\
(S\ \mathbf{comp}\ T).\mathit{outputs} = S.\mathit{outputs} \cup T.\mathit{outputs} \land \\
(S\ \mathbf{comp}\ T).\mathit{locals} = S.\mathit{locals} \cup T.\mathit{locals} \land \\
(S\ \mathbf{comp}\ T).\mathit{opids} = S.\mathit{opids} \cup T.\mathit{opids}
\end{array}$$

The **comp** operator is commutative and associative.

The history of a component class can be derived from a history of a composite class by restricting the history of the composite class to the component class's signature.

The binary operator **restrict** extracts, from a real-time history h, a history corresponding to the signature S of a component class.

$$\begin{array}{|l}
\_\ \mathbf{restrict}\ \_ : \mathit{RTHistory} \times \mathit{Signature} \rightarrow \mathit{RTHistory} \\
\hline
(h\ \mathbf{restrict}\ S).\mathit{trace} = (S.\mathit{inputs} \cup S.\mathit{outputs} \cup S.\mathit{locals}) \lhd h.\mathit{trace} \land \\
(h\ \mathbf{restrict}\ S).\mathit{occurs} = S.\mathit{opids} \lhd h.\mathit{occurs}
\end{array}$$

The parallel combination of two classes, $C \parallel D$, has a signature which is the composition of the signatures of C and D. Each of the histories of the parallel combination, if restricted to the signature of C (respectively D), is a trace of C (D).

$$\begin{array}{|l}
\_ \parallel \_ : \mathit{RTClass} \times \mathit{RTClass} \rightarrowtail \mathit{RTClass} \\
\hline
(C, D) \in \mathrm{dom}(\_ \parallel \_) \Leftrightarrow (C.\mathit{sig}, D.\mathit{sig}) \in \_\ \mathbf{comp}\ \_ \land \\
(C \parallel D).\mathit{sig} = C.\mathit{sig}\ \mathbf{comp}\ D.\mathit{sig} \land \\
(C \parallel D).\mathit{histories} = \{h : \mathit{RTHistory} \mid \\
\qquad\qquad h\ \mathbf{restrict}\ C.\mathit{sig} \in C.\mathit{histories} \land \\
\qquad\qquad h\ \mathbf{restrict}\ D.\mathit{sig} \in D.\mathit{histories}\}
\end{array}$$

Parallel composition of classes is commutative and associative. Furthermore, it is monotonic with respect to refinement in both its arguments.

6 Conclusion

In this paper, we have shown how Object-Z's class and inheritance constructs can be used to structure specifications in real-time Object-Z: an integration of Object-Z with the timed refinement calculus. In particular, for composing Object-Z classes to form multi-component systems, we have introduced a parallel composition operator similar to those found in process algebras. This operator provides a means of composition which is both concise (due to implicit modelling of concurrency) and allows more general modelling of concurrency (by allowing partially overlapping, as well as synchronising, operations). The existing semantics of real-time Object-Z was extended to accommodate the parallel composition operator in such a way that the operator is commutative, associative and monotonic with respect to refinement.

Acknowledgements

This work is funded by Australian Research Council Large Grant A49801500, *A Unified Formalism for Concurrent Real-Time Software Development*.

A Semantics of real-time classes

To provide a semantics for our integrated notation, we show how to map the standard Object-Z semantics to timed traces. Smith [**?**, §2.3] gives a history model for an Object-Z class in terms of sequences of states and operations. We introduce that semantics and then show how to relate it to timed traces.

A.1 Histories

Let *Ident* denote the set of all identifiers, and *Value* the set of all values of any type. A state is an assignment of values to a set of identifiers representing its attributes. It can be defined by a finite partial function from identifiers to values:

$$State == Ident \mathrel{+\!\!\!\!\twoheadrightarrow} Value.$$

An operation can be defined as an identifier corresponding to the operation's name (since operations do not have input and output parameters in real-time Object-Z):

$$Operation == Ident$$

The history of an object consists of (possibly infinite) sequences of states and operations. The sequence of states is non-empty as there must be at least an initial state. The set of attributes of every state in the sequence comprises the state variables of the object's class[2] and hence must be the same. If the sequence of operations is finite, then the sequence of states has the initial state plus an element corresponding to the final state of every operation. Hence the sequence of states is one longer than the sequence of operations. If the sequence of operations is infinite, then so is the sequence of states.

These conditions are captured by the following schema where $\mathrm{seq}_\infty X == \mathrm{seq}\, X \cup (\mathbb{N}_1 \rightarrow X)$. For state variables corresponding to environmental outputs, *attributes* contains their names without the "!" decoration. The fact that they are environmental outputs is captured in *TraceHistory* (Section A.3).

[2] In Smith's model [**?**] the attributes of states include, as well as state variables, all constants the object's class can refer to. Here we take an alternative view that the values of such constants are parameters to the semantics.

History

$states : \mathrm{seq}_\infty\ State$
$ops : \mathrm{seq}_\infty\ Operation$
$attributes : \mathbb{F}\ Ident$
$opids : \mathbb{F}\ Operation$

$states \neq \langle\rangle$
$\forall\, i : \mathrm{dom}\, states \bullet \mathrm{dom}(states\ i) = attributes$
$\mathrm{ran}(ops) \subseteq opids$
$\forall\, i : \mathbb{N}_1 \bullet i \in \mathrm{dom}\, ops \Leftrightarrow i + 1 \in \mathrm{dom}\, states$

A.2 Start and finish times

To map histories to timed traces, we extend the standard definition of an Object-Z history given above. The first extension is to allow for the start and finish times of each operation. The variable *start* denotes the sequence of start times of operations, and the variable *finish* denotes a sequence, with indices starting from 0, of finish times. We use $finish(0)$ to represent the time at which the initialisation completed, and if the sequence of operations is finite we add an extra start time, with value ∞, representing that after the last operation, the state is stable forever.

TimedHistory

History
$start : \mathrm{seq}_\infty\ \mathbb{T}$
$finish : \mathbb{N} \pfun \mathbb{T}$

$\forall\, i : \mathbb{N} \bullet i \in \mathrm{dom}\, ops \Leftrightarrow \{i, i+1\} \subseteq \mathrm{dom}\, start$
$\mathrm{dom}\, start \neq \mathbb{N}_1 \Rightarrow last(start) = \infty$
$\mathrm{dom}\, finish = \{0\} \cup \mathrm{dom}\, ops$
$\forall\, i : \mathrm{dom}\, ops \bullet start(i) \leq finish(i)$
$\forall\, i : \mathrm{dom}\, finish;\ j : \mathrm{dom}\, start \bullet i < j \Rightarrow finish(i) \leq start(j)$

A.3 Timed traces

The next extension is to add timed traces of variables. The timed trace of a variable is a mapping from time to the value of the variable at that time.

$$Trace == \mathbb{T} \rightarrow Value$$

We add a timed trace for every environmental variable and each state variable of the class. The names of environmental variables appear in the semantics without their "?" or "!" decorations. This information is captured instead by three sets of identifiers: *inputs* for environmental inputs, *output* for environmental outputs, and *locals* for local state variables. The local state variables are a