A Formal Object Approach to Real-Time Specification

Jin Song Dong **John Colton** **Lin Zucconi**

Division of Information Technology
Commonwealth Scientific and Industrial Research Organisation (CSIRO)
Canberra, ACT 2601, Australia

{jsdong, johnc, zucconi}@cbr.dit.csiro.au

Abstract

Real-time safety critical systems have both functional and timing requirements. Producing the correct result at the right time is the goal of such systems. The creation of a precise specification of a real-time system is essential for the development of the system. This paper applies and extends the formal specification language Object-Z to incorporate the notion of timing behaviour for specifying real-time systems.

Keywords*: formal object modelling, real-time specification, Object-Z.*

1 Introduction

Formal methods are well accepted techniques for the development of safety-critical systems. Safety-critical systems often have both functional and timing requirements. Producing the correct result at the right time is the fundamental goal of such systems. A precise specification of a real-time system is an essential part of the development of the system. Numerous approaches for using Z[20, 13] notation in the specification phase of the formal development of real-time systems have been proposed. Most notable and successful ones are the timed refinement approach by Mahony and Hayes[15, 16], the ProCos approach by Ravn, He, Hoare *et al.*[14, 17] and the Quartz approach by Fidge *et al.*[11, 10]. Baumann and Lermer[2] also applied Z (similar to the Quartz approach) to specify parallel and distributed real-time systems. In the timed refinement and ProCos approaches, all variables are modelled as traces of values (i.e. functions from time to actual values); while in the Quartz approach, a *now* variable is added to the specification to capture the current absolute time. Although the timed refinement and ProCos approaches are expressive and can also capture concurrency well, they force the specifier to take a very abstract view of the system, e.g. all variables are viewed as constant functions over all time. A consequence of this approach is that the main features of the Z specification, such as the state schemas representing the state of a system and operation schemas representing the state transactions of the system, are buried in the specification. On the other hand, the Quartz approach of adding the *now* variable has the advantage of maintaining the original Z style specification. However, capturing the concurrency properties of a real-time system is cumbersome. Furthermore, none of this research has considered how these techniques can be integrated with object-oriented methods. As object-orientation becomes an important structural technique for modelling complex software systems, it is necessary to combine real-time and object-oriented modelling techniques into one specification language. In this way we can present a object-oriented model for a real-time system. In this paper, we incorporate the real-time modelling techniques (discussed above) with the object-orientation paradigm into a framework. This framework enhances the Object-Z notation for specifying real-time systems.

Section 2 briefly describes the Object-Z notation. Section 3 presents the framework that extends Object-Z notation for specifying real-time systems. Section 4 applies this framework to specify a robot-arm assembly unit system. This example system is based on the robot-arm system presented in [1]. Section 5 discusses the timing behaviour of composition operators. Section 6 concludes the paper and points out some further research.

2 Object-Z Background

Object-Z is an extension of the Z formal specification language to accommodate object orientation. The main reason for this extension is to improve the clarity of large specifications through enhanced structuring. In this section, some basic key features of

Object-Z (needed later in this paper) are introduced and briefly discussed. Familiarity with the basic features of the Z notation as given in [20] is assumed.

Classes

A class is a template for *objects* of that class: for each such object, its states are instances of the class' state schema and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Syntactically, a class definition is a named box, optionally with generic parameters. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and operation schemas.

ClassName[*generic parameters*]

> *state schema*
>
> *initial state schema*
>
> *operation schemas*

A generic stack example

Consider the following specification of the generic class *Stack*.

Stack[T]

> $items : \operatorname{seq} T$
> Δ
> $size : \mathbb{N}$
> ―――
> $size = \# items$
>
> *INIT*
> $items = \langle\,\rangle$
>
> *Push*
> $\Delta(items)$
> $item? : T$
> ―――
> $items' = \langle item? \rangle \frown items$
>
> *Pop*
> $\Delta(items)$
> $item! : T$
> ―――
> $items \neq \langle\,\rangle$
> $items = \langle item! \rangle \frown items'$

The state schema is nameless and contains declarations (the attributes) above the short dividing line and a predicate (class invariant) below the line. In this example, there is an attribute *items* denoting a sequence of elements of the generic type T.

The initial state schema is distinguished by the keyword *INIT*. The state schema is implicitly included in the initial state schema. In this example, an initialised stack contains no elements of T (i.e. *items* is the empty sequence).

The remaining two schemas are operation schemas. Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value. Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation).

In this example, operation *Push* prepends a given input *item*? to the existing sequence of items (an identifier ending in '?' denotes an input). Operation *Pop* outputs a value *item*! defined as the head of sequence *items* and reduces *items* to the tail of its original value (an identifier ending in '!' denotes an output).

Note that the attribute *size* representing the size of *items* is a secondary attribute. Syntactically, the declarations of secondary attributes appear below the Δ separator placed in the declaration section of the state schema. The values of the secondary attributes are always subject to change. This is achieved by implicit inclusion of secondary attributes in the Δ-list of every operation. In this case, the value of *size* is always subject to change whenever operation *Push* or *Pop* is invoked. For a detailed discussion on secondary attributes see [7, 9].

This generic stack class will be inherited to define some component objects for the robot assembly system in Section 4.

Instantiation

Objects may have object references as attributes, i.e. conceptually, an object may have constituent objects. Such references may either be individually named or occur in aggregates. For example, the declaration $c : C$ declares c to be a reference to an object of the class described by C. A declaration $c, d : C$ need not mean that c and d reference distinct objects. If the intention is that they do so at all times, then the predicate $c \neq d$ would be included in the class invariant.

The term $c.att$ denotes the value of attribute *att* of the object referenced by c, and $c.Op$ denotes the evolution of the object according to the definition of *Op* in the class C.

Further details on Object-Z are given in [8] and [18].

3 Adding Real-Time to Object-Z

In this section, we will present a framework that extends Object-Z notation for specifying real-time systems. The aims of this extension are the following:

- The concurrent active and interactive nature of objects forms an essential feature of object-oriented systems. Therefore a good treatment for concurrent object interaction is needed in our framework, in particular for the interactions between input-output environments and the core system components.
- When all component objects are composed together to form a system, local time of system components must be synchronised with a unique clock time. Hence, the framework needs to synchronise all system components to share the same absolute time variable.
- Any language extension should aim to make minimum disruption of the existing language semantics. Therefore the framework aims to maintain current Object-Z specification style and semantics.

Firstly let's borrow the basic time type model from [12]:

$$\mathbb{A} == \mathbb{N}$$

(representing the absolute discrete time domain)

$$\mathbb{E} == \mathbb{P}_1 \, \mathbb{A}$$

(representing the possible execution times of each operation)

Object-Z is state-based language. As time is part of the state of a real-time system, every object of a system must be associated with a notion of time. Therefore let every Object-Z class implicitly have a secondary attribute '$now : \mathbb{A}$' to represent the absolute current time state. The property that the absolute time is shared by every object in a system is captured by a global system invariant

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$$

where $\mathbb{O}$[4] denotes the universe of all object identities, regardless of class, that may be in the system. Also let every operation of any Object-Z class implicitly include the predicate

$$now' \geq now$$

to guarantee time cannot go backwards. If D (of type $\mathbb{E}$) represents the set of possible execution times for an operation, then in general the time execution requirement of the operation can be expressed in the predicate part of the operation schema as

$$now' - now \in D$$

Real-time behaviour of a system is often influenced and controlled by the input environment of the system. Therefore, the real-time requirement of a system often has certain assumptions or constraints on the input system environment. Using object-oriented modelling techniques, we can specify the input environment as a component object of the whole system. The main functionality of such an input environment object is to pass (input) objects to the core system body at various times. For instance, an input environment object can be modelled by the following class frame

```
┌─ InputEnv ──────────────────────────────
│ ┌────────────────────────────────────
│ │ supply : 𝔸 ⇸ InputObjects
│ │ ...
│ └────────────────────────────────────
│ ...
└─────────────────────────────────────────
```

Notice that the timed refinement and ProCos approaches (defining variables as functions from time to values) are taken to model the input environment.

Any system specified using Object-Z includes a system object which is defined by a system class. The system object consists of all component objects of the system. Every operation performed on a system object will change the state of the system object and/or the state of component objects of the system object. As the time variable *now* is part of the state of the system object, and the input environment is also modelled as a part of the system object, some of the preconditions of system operations are contributed by the time behaviour of the input environment object. The system class also needs to have an operation *IdleTick* which only changes time by one time unit forward when no other operation is enabled. For example, a system class which has n operations can be specified as

```
┌─ SystemX ───────────────────────────────
│ ┌────────────────────────────────────
│ │ inenv : InputEnv
│ │ ...
│ └────────────────────────────────────
│ Op₁ ≙ ...
│ Op₂ ≙ ...
│ ...
```

$$
\begin{array}{|l}
Op_n \mathrel{\widehat{=}} \ldots \\
\begin{array}{|l}
\hline
\textit{IdleTick} \\
\neg\, pre(op_1) \\
\neg\, pre(op_2) \\
\ldots \\
\neg\, pre(op_n) \\
now' - now \in \{1\} \\
\hline
\end{array} \\
\hline
\end{array}
$$

As time always changes and time is also an attribute of a system, the overall behaviour of the system (*SystemX* in this case) can be modelled as a continued sequence of system operations including *IdleTick*, i.e.

$$
\begin{aligned}
& process \mathrel{\widehat{=}} \\
& \quad (Op_1 \;[\!]\; Op_2 \;[\!]\; \ldots \;[\!]\; Op_n \;[\!]\; IdleTick) \fatsemi process
\end{aligned}
$$

The choice operator '$[\!]$' defines a system operation in terms of a choice between two given operations. The choice is angelic in that if only one component operation is enabled that operation will be selected; if both are enabled the selection is non-deterministic. The sequential operator '$\fatsemi$' defines a system operation in terms of the sequential composition of two given operations. The semantics of these operators are defined in [19].

4 Case Study: Robot Arm Unit Specification

In this case study, the system requirement is based on the description of a robot-arm system in [1].

4.1 Informal Description and System Requirements

The assembly unit consists of a robot system, a conveyor belt, and an assembly-tray (see Figure 1 for an illustration). The belt normally carries two kinds of parts, say a *cup* and a *dish* (by a cup and a dish, we mean any two parts that can be assembled together). The robot system picks a cup and a dish from the belt, assembles them and places the assembled product on a tray. The belt stops momentarily for a pre-specified period of time whenever a part is in the pre-specified location known as the pick-up zone for the part to be picked up. This is done by a sensor located under the belt. A part will be trashed if it is not picked up in the pick up zone. However, it is guaranteed that the parts in the belt are separated by a minimum distance in order to ensure a minimum time delay between any two consecutive entries of parts in the pick-up zone. The parts may arrive through the belt in any order. It is required that for any arbitrary placement of n cups and n dishes on the belt the robot should produce n assemblies. This necessitates that no cup or dish placed in the belt is trashed. The robot system consists of two arms, a vision system and a stack. The vision system can recognise the assembly parts (i.e., cup and dish) and record the number of cups or dishes been recognised. The stack is for temporarily storing the parts. The vision system is continuously focussed on the belt so that the scanning and recognition begins when the part on the conveyor belt enters the camera's view and the belt is stopped. When the vision system recognises a cup or a dish with a maximal time delay constraint, the left arm or the right arm is activated so as to pick the part from the belt. Initially the left arm is free and the stack is empty. Whenever both arms are free and the stack is empty, and the vision system recognises a part then the left arm picks up the part from the pick-up zone in the conveyor belt. When the left arm holds a part, the right arm, if free, picks up the next part from the conveyor belt, when signaled by the vision system. If the part on the right arm is the same as the part on the left arm, the part in the right arm is pushed into the stack; otherwise the parts are assembled and placed on an assembly-tray. If the left arm is free but the stack is not empty, the left arm picks up (pops) a part from the stack. In the design of the assembly process, for efficiency reasons, the left arm is made free soon after the assembly, while the right arm is made free only after placing the assembly on the assembly-tray.

System requirements

- Functional requirements:
  - f1: Whenever both the arms are free,

    $$Asm\_cnt = Min(Cup\_cnt, Dish\_cnt)$$

    where Asm_cnt is the total number of assemblies made by the robot, Cup_cnt is the number of cups detected, and $Dish\_cnt$ is the number of dishes detected by the vision system.
  - f2: A non-empty stack will always contain parts of the same type.
- Timing requirements:
  - t1: Once a part enters the camera's view and stops, the vision system should recognise the part within a certain time interval.
  - t2: Once the vision system recognises the part, a robot arm should pick the part within

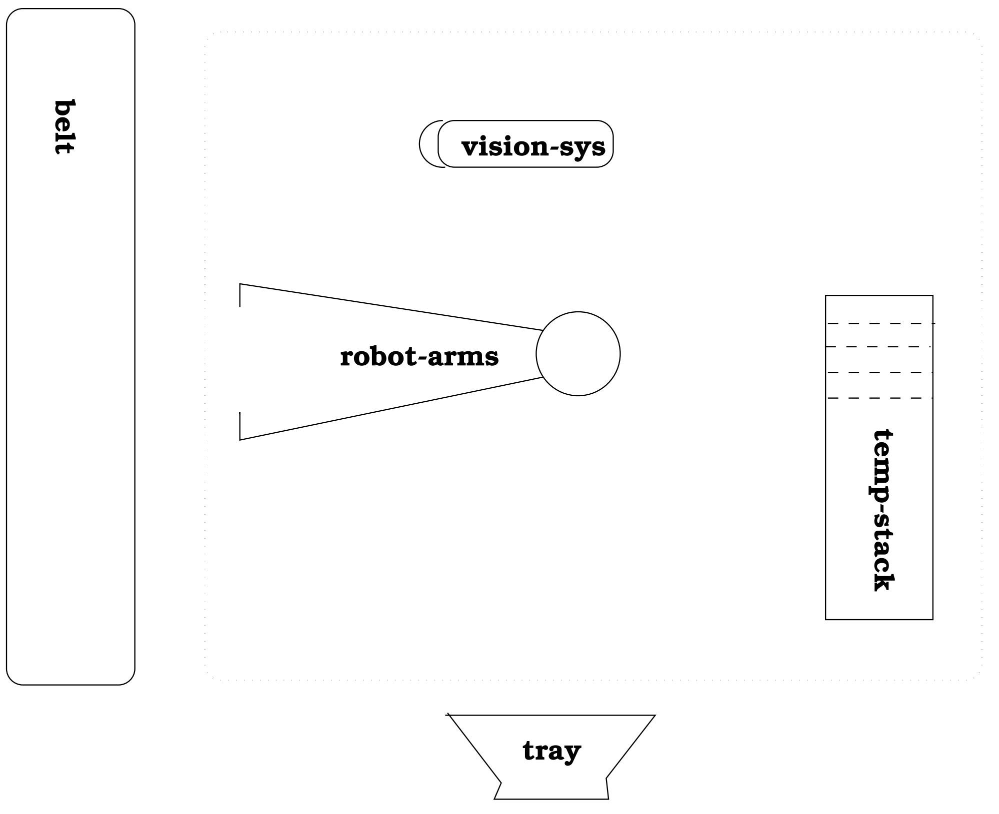

Figure 1: A robotic assembly unit.

a certain time interval. Time for releasing a part from a robot arm must also occur within a certain time interval.

- t3: A part must be grasped and removed from the belt within the time interval during which the belt is in the stopped state.
- t4: To ensure that a robot arm is in *ready* state before the belt is stopped, the system requires that the duration of releasing a part from a robot arm must be within the time interval during which the belt is moving.

4.2 Formal System Specification in Object-Z

The first task of using an object-oriented approach to model a system is to identify all component objects of the overall system; in this case it is trivial. Then one needs to identify the core component objects and the input and output environment objects of the core components. The core component objects of this robotic assembly system are a vision-system, two robot-arms and a temporary-stack. The input environment object of the robot arm unit is a conveyor belt. The output environment object of the robot arm unit is an assembly-tray. The system specification starts with the definition of basic objects such as Cup, Dish and Product (composite of Cup and Dish), then the specification of input and output environments of the core system, followed by the specification of the core components, and finally the overall system object specification which captures the interactions between all component objects.

4.2.1 Predefined Constants and Basic Objects

The following are predefined time constants and constraints on these constants.

$$
\begin{array}{l}
\mathit{StartTime} : \mathbb{A} \\
\mathit{MinMoveTime}, \mathit{StopTime} : \mathbb{A} \\
\mathit{MaxRecogniseTime} : \mathbb{A} \\
\mathit{MaxPickTime} : \mathbb{A} \\
\mathit{MaxReleaseTime} : \mathbb{A} \\
\hline
\mathit{MaxRecogniseTime} + \mathit{MaxPickTime} \leq \mathit{StopTime} \qquad [\mathtt{g1}] \\
\mathit{MaxReleaseTime} \leq \mathit{MinMoveTime} \qquad [\mathtt{g2}]
\end{array}
$$

Note that *StartTime* denotes the starting time for the whole system.

The constraints `g1` and `g2` with the following class definitions ensure the timing requirements `t3` and `t4`.

Let

$$
\begin{array}{l}
\mathit{Cup} \\
\hline
\quad \ldots
\end{array}
$$

$$
\begin{array}{l}
\mathit{Dish} \\
\hline
\quad \ldots
\end{array}
$$

$$
\begin{array}{l}
\mathit{Product} \\
\hline
\quad \mathit{uppart} : \mathit{Cup}_{©} \\
\quad \mathit{downpart} : \mathit{Dish}_{©}
\end{array}
$$

represent Cup, Dish and Product objects.

Notice that the semantics of the containment '©'[5] ensures the following property:

$$
\begin{array}{l}
\forall u_1, u_2 : \mathit{Product} \bullet \\
\quad u_1 \neq u_2 \Rightarrow \\
\qquad u_1.\mathit{uppart} \neq u_2.\mathit{uppart} \land \\
\qquad u_1.\mathit{downpart} \neq u_2.\mathit{downpart}
\end{array}
$$

A general type of an object which refers to either Cup or Dish can be modelled as a class-union[3, 4]:

$$\mathit{Part} \mathrel{\widehat=} \mathit{Cup} \cup \mathit{Dish}$$

4.2.2 Input and Output Environment

The input environment object, the conveyor belt, constantly stops and supplies a Cup or Dish for a robot arm to pick up. It can be modelled as:

$$
\begin{array}{l}
\mathit{Belt} \\
\hline
\quad \mathit{supply} : \mathbb{A} \rightarrowtail\!\!\!\!\!\!\!+\;\; \mathit{Part} \\
\quad \hline
\quad \forall p : \mathrm{ran}\, \mathit{supply} \bullet \\
\qquad \mathrm{dom}(\mathit{supply} \rhd \{p\}) = \\
\qquad\quad \mathit{min}\, \mathrm{dom}(\mathit{supply} \rhd \{p\}) \mathrel{..} \mathit{max}\, \mathrm{dom}(\mathit{supply} \rhd \{p\}) \\
\qquad\quad \land \#(\mathit{supply} \rhd \{p\}) = \mathit{StopTime} \\
\quad \forall m, n : (\mathbb{A} \setminus \mathrm{dom}\, \mathit{supply}) \bullet \\
\qquad\quad (m - 1 \in \mathrm{dom}\, \mathit{supply} \land \\
\qquad\quad\; n + 1 \in \mathrm{dom}\, \mathit{supply} \land \\
\qquad\quad\; m \mathrel{..} n \subseteq (\mathbb{A} \setminus \mathrm{dom}\, \mathit{supply})) \Rightarrow \\
\qquad\qquad n - m \geq \mathit{MinMoveTime} \\
\\
\quad \mathrm{I{\scriptsize NIT}} \\
\quad \hline
\quad \mathit{min}(\mathrm{dom}\, \mathit{supply}) = \mathit{StartTime} \\
\\
\quad \mathit{ReadyForPick} \\
\quad \hline
\quad \mathit{part}! : \mathit{Part} \\
\quad \hline
\quad \exists p : \mathrm{ran}\, \mathit{supply} \bullet \\
\qquad\quad \mathit{now} = \mathit{min}(\mathrm{dom}(\mathit{supply} \rhd \{p\})) \\
\qquad\quad \mathit{now}' \leq \mathit{max}(\mathrm{dom}(\mathit{supply} \rhd \{p\})) \\
\qquad\quad p = \mathit{part}!
\end{array}
$$

The class invariants precisely capture the constraints of the timing behaviour of the conveyor belt. For example, the class invariant predicate

$$
\begin{array}{l}
\forall p : \mathrm{ran}\, \mathit{supply} \bullet \\
\quad \mathrm{dom}(\mathit{supply} \rhd \{p\}) = \\
\qquad \mathit{min}\, \mathrm{dom}(\mathit{supply} \rhd \{p\}) \mathrel{..} \mathit{max}\, \mathrm{dom}(\mathit{supply} \rhd \{p\}) \\
\qquad \land \#(\mathit{supply} \rhd \{p\}) = \mathit{StopTime}
\end{array}
$$

ensures that the belt is stopped for a contiguous fixed time interval to supply each part. (Note that the set $\{1, 3\}$ is not contiguous as $\mathit{min}\{1, 3\} \mathrel{..} \mathit{max}\{1, 3\} == \{1, 2, 3\}$)

The output environment object, tray, can be modelled as

$$
\begin{array}{l}
\mathit{Tray} \\
\hline
\quad \mathit{items} : \mathrm{seq}\, \mathit{Product} \\
\\
\quad \mathrm{I{\scriptsize NIT}} \\
\quad \hline
\quad \mathit{items} = \langle\, \rangle \\
\\
\quad \mathit{Push} \\
\quad \hline
\quad \Delta(\mathit{items}) \\
\quad \mathit{unit}? : \mathit{Product} \\
\quad \hline
\quad \mathit{items}' = \langle \mathit{unit}? \rangle \frown \mathit{items}
\end{array}
$$

Alternatively the class *Tray* can be defined by inheriting the instantiated generic class *Stack* from Section 2 as:

$$
\begin{array}{|l}
\hline
\textit{Tray} \\
\textit{Stack}[\textit{Product}_{©}] \\
\qquad\qquad [\textit{Push}[\textit{unit}?/\textit{item}?], \mathbf{remove}\ \textit{Pop}] \\
\hline
\end{array}
$$

This demonstrates that a compact system specification can be developed from a generic class library.

4.2.3 The Core System Components

The vision system has two attributes which record the number of cups and dishes. The vision system can be modelled as:

$$
\begin{array}{|l}
\hline
\textit{VisionSystem} \\
\quad \begin{array}{|l}
\hline
\textit{cup\_cnt} : \mathbb{N} \\
\textit{dish\_cnt} : \mathbb{N} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
\textit{cup\_cnt} = 0 \\
\textit{dish\_cnt} = 0 \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Recognise} \\
\Delta(\textit{cup\_cnt}, \textit{dish\_cnt}) \\
\textit{part}? : \textit{Part} \\
\hline
\textit{part}? \in \textit{Cup} \Rightarrow \\
\quad (\textit{cup\_cnt}' = \textit{cup\_cnt} + 1 \wedge \\
\quad\ \ \textit{dish\_cnt}' = \textit{dish\_cnt}) \\
\textit{part}? \in \textit{Dish} \Rightarrow \\
\quad (\textit{cup\_cnt}' = \textit{cup\_cnt} \wedge \\
\quad\ \ \textit{dish\_cnt}' = \textit{dish\_cnt} + 1) \\
\textit{now}' - \textit{now} \in 0 \mathinner{\ldotp\ldotp} \textit{MaxRecogniseTime} \\
\hline
\end{array} \\
\hline
\end{array}
$$

The last conjunct in the predicate part of the operation *Recognise* captures the timing requirement t1.

The major components of the system are the two robot arms. A robot arm is always in 'Ready' (holding nothing) or 'Taken' (holding a cup or dish) state. It can perform two operations, *Pick* and *Release*.

$$
\begin{array}{|l}
\hline
\textit{RobotArm} \\
\textit{State} \mathrel{\widehat=} \textit{ready} \mid \textit{taken} \\
\quad \begin{array}{|l}
\hline
s : \textit{State} \\
\textit{hold} : \textit{Part} \cup \textit{Null} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{INIT} \\
s = \textit{ready} \\
\textit{hold} \in \textit{Null} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Pick} \\
\Delta(s, \textit{hold}) \\
\textit{part}? : \textit{Part} \\
\hline
\textit{hold}' = \textit{part}? \\
s = \textit{ready} \\
s' = \textit{taken} \\
\textit{now}' - \textit{now} \in 0 \mathinner{\ldotp\ldotp} \textit{MaxPickTime} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Release} \\
\Delta(s, \textit{hold}) \\
\textit{part}! : \textit{Part} \\
\hline
\textit{part}! = \textit{hold} \\
\textit{hold}' \in \textit{Null} \\
s = \textit{taken} \\
s' = \textit{ready} \\
\textit{now}' - \textit{now} \in 0 \mathinner{\ldotp\ldotp} \textit{MaxReleaseTime} \\
\hline
\end{array} \\
\hline
\end{array}
$$

The last conjunct in the predicate part of the operation *Pick* and the last conjunct in *Release* capture the timing requirement t2.

A stack for temporarily storing parts can be modelled by inheriting the generic stack class from Section 2.

$$
\begin{array}{|l}
\hline
\textit{TempStack} \\
\textit{Stack}[\textit{Part}_{©}] \\
\qquad\qquad [\textit{Push}[\textit{part}?/\textit{item}?], \textit{Pop}[\textit{part}!/\textit{item}!]] \\
\quad \begin{array}{|l}
\hline
\operatorname{ran} \textit{items} \subseteq \textit{Cup} \vee \operatorname{ran} \textit{items} \subseteq \textit{Dish} \\
\# \textit{items} = \# \operatorname{ran} \textit{items} \\
\hline
\end{array} \\
\hline
\end{array}
$$

The class invariant of the *TempStack* ensures the functional requirement f2.

4.2.4 The Assembly Unit System

The assembly unit system consists of a belt, a vision system, two robot arms, a stack, and a tray.

$$
\begin{array}{l}
\textbf{AssemblyUnitSystem} \\
\hline
b : Belt \\
v : VisionSystem \\
l, r : RobotArm \\
s : TempStack \\
t : Tray \\
\hline
l \neq r \\
(l.s = ready \land r.s = ready) \Rightarrow \\
\quad \# t.items = min\{v.cup\_cnt, v.dish\_cnt\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{INIT} \\
\hline
b.\mathrm{INIT} \land v.\mathrm{INIT} \\
l.\mathrm{INIT} \land r.\mathrm{INIT} \\
s.\mathrm{INIT} \land t.\mathrm{INIT} \\
now = StartTime
\end{array}
$$

$$
\begin{array}{l}
LeftArmPick \mathrel{\widehat{=}} \\
\quad [s.items = \varnothing] \land b.ReadyForPick \\
\qquad\qquad \parallel (v.Recognise \mathbin{⨾} l.Pick) \\
RightArmPick \mathrel{\widehat{=}} \\
\quad b.ReadyForPick \parallel (v.Recognise \mathbin{⨾} r.Pick) \\
RightArmPushToStack \mathrel{\widehat{=}} \\
\quad [(\{l.hold, r.hold\} \subseteq Cup) \lor \\
\quad\ (\{l.hold, r.hold\} \subseteq Dish)] \\
\qquad\quad \land r.Release \parallel s.Push \\
LeftArmGetFromStack \mathrel{\widehat{=}} l.Pick \parallel s.Pop
\end{array}
$$

$$
\begin{array}{l}
\textbf{Make} \\
\hline
part_1, part_2? : Part \\
unit! : Product \\
\hline
(part_1? \in Cup \land part_2? \in Dish) \Rightarrow \\
\quad (unit!.uppart = part_1? \land \\
\quad\ unit!.downpart = part_2?) \\
(part_1? \in Dish \land part_2? \in Cup) \Rightarrow \\
\quad (unit!.uppart = part_2? \land \\
\quad\ unit!.downpart = part_1?)
\end{array}
$$

$$
\begin{array}{l}
Assemble \mathrel{\widehat{=}} \\
\quad [(l.hold \in Cup \land r.hold = Dish) \\
\quad\ \lor (r.hold \in Cup \land l.hold = Dish)] \bullet \\
\qquad (l.Release[part_1?/part?] \land \\
\qquad\ r.Release[part_2?/part?]) \\
\qquad \parallel Make \\
\qquad \parallel t.Push
\end{array}
$$

$$
\begin{array}{l}
\textbf{IdleTick} \\
\hline
\neg\, pre(LeftArmPick) \\
\neg\, pre(RightArmPick) \\
\neg\, pre(RightArmPushToStack) \\
\neg\, pre(LeftArmGetFromStack) \\
\neg\, pre(Assemble) \\
now' - now \in \{1\}
\end{array}
$$

The second conjunct of the class invariant ensures the functional requirement `f1`.

The initial state of the system is that all components of the system are in their initial state together with the time starting at the *StartTime*. From the global system invariant

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$$

all components of the system are also started at the same time.

The overall behaviour of the system can be described as the following process:

$$
\begin{array}{l}
A\_process \mathrel{\widehat{=}} \\
\qquad (LeftArmPick \\
\qquad\ [\!]\ RightArmPick \\
\qquad\ [\!]\ RightArmPushToStack \\
\qquad\ [\!]\ RightArmPushToStack \\
\qquad\ [\!]\ LeftArmGetFromStack \\
\qquad\ [\!]\ Assemble \\
\qquad\ [\!]\ IdleTick) \mathbin{⨾} A\_process
\end{array}
$$

This overall behaviour ensures the timing requirements `t3` and `t4`. For instance, the appendix demonstrates the formal proof outlines for showing that the timing behaviour of the system operation *LeftArmPick* of the class *AssemblyUnitSystem* meets the timing requirement `t3`.

This completes the specification of the assembly unit system.

5 Timing Behaviour of Composition Operations

Object-oriented systems are typically designed by first specifying the basic underlying objects of the system independently in terms of their own individual attributes and operations, and then specifying the system as a whole in terms of the way the underlying objects interact. In this section, we investigate the timing behaviour of composition operators. Experience with using Object-Z to design a range of object-oriented systems has suggested that four composition operators: conjunction, parallel, choice and sequential composition, are necessary, and, at least for commonly occurring architectures, sufficient. To facilitate the discussion, consider the following example.

Suppose a class X is defined as:

$$
\begin{array}{|l}
\hline X \\
\quad \begin{array}{|l} \ldots \\ \hline \end{array} \\
\quad \begin{array}{|l} \hline Op_1 \\ \ldots \\ now' - now \in D_1 \\ \hline \end{array} \\
\quad \begin{array}{|l} \hline Op_2 \\ \ldots \\ now' - now \in D_2 \\ \hline \end{array} \\
\ldots \\
\hline
\end{array}
$$

where $D_1 \in \mathbb{E}$ and $D_2 \in \mathbb{E}$.

As the time attribute now is treated no differently than any other secondary attribute, it is trivial to deduce the timing behaviour of each composition operation from the semantics of the composition operations[19]. For example, as the predicate parts of the two operations Op_1 and Op_2 are conjoined to become the predicate part of $Op_1 \wedge Op_2$, the duration $(now' - now)$ of $Op_1 \wedge Op_2$ will be in $D_1 \cap D_2$. Similarly, duration $(now' - now)$ of other composition operations will be in

- $D_1 \cap D_2$ for $Op_1 \parallel Op_2$,
- $D_1 \cup D_2$ for $Op_1 \mathbin{[]} Op_2$,
- $D_1 \dotplus D_2$ for $Op_1 \fatsemi Op_2$, where

$$
\begin{array}{|l}
\_ \dotplus \_ : \mathbb{E} \times \mathbb{E} \rightarrow \mathbb{E} \\
\hline
\forall A, B : \mathbb{E} \bullet \\
\qquad A \dotplus B = \{a : A;\ b : B \bullet a + b\}
\end{array}
$$

e.g. $\{1,3\} \dotplus \{2,4\} = \{3,5,7\}$.

In some circumstances, composition of timing behaviour may not follow the composition of functional behaviour (conjunctions or disjunctions of predicates). For example, from an observational point of view, Figure 2 illustrates a natural combined timing behaviour of two operations executing in parallel.

The expected concurrent behaviour of two operations executing in parallel (as in Figure 2) can be defined as:

$$
\begin{array}{l}
Op_1 \wedge Op_2 == \\
\quad ((Op_1 \wedge Op_2) \setminus (now, now')) \wedge \\
\quad [\Delta(now) \mid now' - now \in D_1 \shortparallel D_2]
\end{array}
$$

where

$$
\begin{array}{|l}
\_ \shortparallel \_ : \mathbb{E} \times \mathbb{E} \rightarrow \mathbb{E} \\
\hline
\forall A, B : \mathbb{E} \bullet \\
\qquad A \shortparallel B = \{x : (A \cup B) \mid \\
\qquad\qquad x \geq max\{min(A), min(B)\}\}
\end{array}
$$

e.g. $\{1,3\} \shortparallel \{2,4\} = \{2,3,4\}$

6 Conclusion and Further Work

In this paper, we have integrated various Z approaches for specifying real-time requirement with object-oriented modelling techniques into a framework. This framework extends Object-Z notation so that it can be used to specify real-time systems. This extension has maintained the existing Object-Z semantics and style by not adding any new constructs into Object-Z. The framework has been successfully applied to a case study of a robotic assembly system which has both functional and timing requirements. We also are in the progress of formalising process scheduling requirements for an aircraft operational flight program [6]. This framework suggests the use of timed refinement and ProCos approach to model input environment and the use of Quartz approach to model the core system components. We believe this suggestion will work well in general. However, the combined approach may also be necessary to model the core system components, e.g. modelling some attributes of a system component as constant functions from time to values (not necessary for this robot arm study). The timing behaviours of the composition operations have also been discussed. As this paper is a report on the initial research work on object-oriented real-time formal specification, we believe there will be other natural combined time behaviours for these composition operations. Therefore further research will involve more case studies in order to find other common expected timing behaviours for composite operations.

Acknowledgements

We would like to thank Colin Fidge and Roger Duke for many helpful discussions on the issues raised in this paper.

References

[1] R. Achuthan, V. S. Alagar, and T. Radhakrishnan. An Object-Oriented Modeling of Real-Time Robotic Assembly System. In Alex Stoyenko, editor, *The First IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)*, pages 310–313, Florida, November 1995. IEEE Computer Society Press.

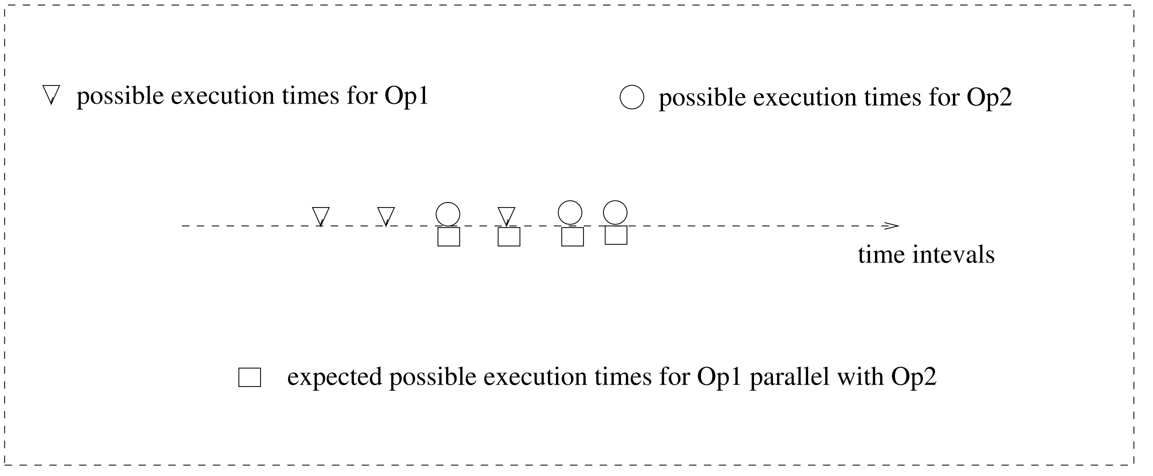

Figure 2: Two parallel operations.

[2] P. Baumann and K Lermer. Specifying Parallel and Distributed Real-Time Systems in Z. In *The 4th International Workshop on Parallel and Distributed Real-Time Systems*, Hawaii, April 1996.

[3] J.S. Dong. Living with Free Type and Class Union. In *The 1995 Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 304–312. IEEE Computer Society Press, December 1995.

[4] J.S. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Proc. 12th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS 12 & 9*, pages 181–190. Prentice-Hall, November 1993.

[5] J.S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.

[6] J.S. Dong, N. Fulton, L. Zucconi, and J. Colton. Formalising Process Scheduling Requirements for an Aircraft Operational Flight Program. Technical Report 96-52, Mathematical and Information Sciences, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia, 1996. (revised in 1997).

[7] J.S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. In Alex Stoyenko, editor, *The First IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)*, pages 31–38, Ft. Lauderdale, USA, November 1995. IEEE Computer Society Press.

[8] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

[9] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[10] C. Fidge. Adding Real Time to Formal Program Development. In *Proceedings of FME'94: Industrial Benefit of Formal Methods*, pages 618–638, Barcelona, October 1994. Springer-Verlag.

[11] C. Fidge, P. Kearney, and M. Utting. Quartz: An integrated formal development method for real-time software. Technical Report 94-26, Software Verification Research Centre, The University of Queensland, Australia, September 1994. (To appear in IEEE Software 1997).

[12] C. Fidge, M. Utting, P. Kearney, and I. Hayes. Integrating Real-Time Scheduling Theory and Program Refinement. In *Proceedings of FME'96: Industrial Benefit of Formal Methods*, pages 325–346, Oxford, March 1996. Springer-Verlag.

[13] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, 2nd edition, 1993.

[14] Jifeng He. *Provably Correct Systems*. McGraw-Hill, 1995.

[15] B. P. Mahony. *The Specification and Refinement of Timed Processes*. PhD thesis, The University of Queensland, 1992.

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

[17] A. Ravn, J. He, C.A.R. Hoare, M. Müller-Olm, E. Olderog, M. Schenke, M. Hansen, and H. Rischel. The ProCoS Approach to the Design of Real-Time Systems: Linking Different Formalisms. Tutorial Notes for FME'96, 1996.

[18] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[19] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

[20] J.M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 2nd edition, 1992.

Appendix: Formal Proof Outline for Ensuring a Timing Constraint

This appendix is a formal proof outline for showing that the timing behaviour of the system operation *LeftArmPick* of the class *AssemblyUnitSystem* meets the timing requirement `t3`.

Firstly, let's consider an object a of the class *AssemblyUnitSystem* (i.e. $a : \mathit{AssemblyUnitSystem}$). To show that the timing behaviour of *LeftArmPick* meets the timing requirement `t3` for the object a is to prove the following predicate is true when the operation $a.\mathit{LeftArmPick}$ is invoked.

$$a.now' - a.now \in 0 \mathrel{..} \mathit{StopTime}$$

The definition of the operation *LeftArmPick* is

$$[s.items = \varnothing] \land b.\mathit{ReadyForPick} \parallel (v.\mathit{Recognise} \fatsemi l.\mathit{Pick})$$

From the global system invariant

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$$

the timing constraints of $v.\mathit{Recognise} \fatsemi l.\mathit{Pick}$ can be expanded as:

$$\begin{array}{l}
[v.now' - v.now \in 0 \mathrel{..} \mathit{MaxRecogniseTime} \land \\
\quad l.now = v.now \land l.now' = v.now'] \\
\fatsemi \\
[l.now' - l.now \in 0 \mathrel{..} \mathit{MaxPickTime} \land \\
\quad l.now = v.now \land l.now' = v.now']
\end{array}$$

By the semantics of the sequential operator $\fatsemi$ (exists a middle state), the following timing constraints can be deduced:

$$\begin{array}{l}
[l.now' - l.now \in \\
\quad 0 \mathrel{..} (\mathit{MaxRecogniseTime} + \mathit{MaxPickTime}) \land \\
\qquad l.now = v.now \land l.now' = v.now']
\end{array}$$

Now for the class *Belt*, after the inclusion of the class invariant into the operation *ReadyForPick*, the timing constraints of the operation $b.\mathit{ReadyForPick}$ will be:

$$\begin{array}{l}
[\forall\, p : \operatorname{ran} \mathit{supply} \bullet \\
\quad \operatorname{dom}(b.\mathit{supply} \rhd \{p\}) = \mathit{min}(\operatorname{dom}(b.\mathit{supply} \rhd \{p\})) \mathrel{..} \mathit{max}(\operatorname{dom}(b.\mathit{supply} \rhd \{p\})) \\
\quad \#(b.\mathit{supply} \rhd \{p\}) = \mathit{StopTime} \land \\
\exists\, p : \operatorname{ran} \mathit{supply} \bullet \\
\quad now = \mathit{min}(\operatorname{dom}(\mathit{supply} \rhd \{p\})) \\
\quad now' \leq \mathit{max}(\operatorname{dom}(\mathit{supply} \rhd \{p\})) \\
\quad p = part!]
\end{array}$$

The following constraint can be deduced by the timing constraint of the operation $b.\mathit{ReadyForPick}$:

$$[b.now' - b.now \in 0 \mathrel{..} \mathit{StopTime}]$$

Again from the global system invariant

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$$

and the semantics of the 'parallel' operator $\parallel$ (conjuncting the predicate parts of the sub-operations), the timing constraints of

$$b.\mathit{ReadyForPick} \parallel (v.\mathit{Recognise} \fatsemi l.\mathit{Pick})$$

can be expanded and deduced as:

$$\begin{array}{l}
b.now' - b.now \in 0 \mathrel{..} \mathit{StopTime} \land \\
l.now' - l.now \in 0 \mathrel{..} (\mathit{MaxRecogniseTime} + \mathit{MaxPickTime}) \land \\
b.now = v.now = l.now \land b.now' = v.now' = l.now'
\end{array}$$

As $\mathit{MaxRecogniseTime} + \mathit{MaxPickTime} \leq \mathit{StopTime}$ (`g1` of Section 4.2.1), the timing constraints of $b.\mathit{ReadyForPick} \parallel (v.\mathit{Recognise} \fatsemi l.\mathit{Pick})$ can be simplified as the following predicate:

$$l.now' - l.now \in 0 \mathrel{..} (\mathit{MaxRecogniseTime} + \mathit{MaxPickTime})$$

Again from the global system invariant

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.now = o_2.now$$

the following timing constraint of the operation $a.LeftArmPick$ can be deduced:

$$[a.now' - a.now \in 0 \mathrel{..} (MaxRecogniseTime + MaxPickTime) \land \\ a.now = l.now \land a.now' = l.now']$$

This implies

$$a.now' - a.now \in 0 \mathrel{..} StopTime$$

(because of `g1` of Section 4.2.1)

Therefore, the timing behaviour of the system operation *LeftArmPick* of the class *AssemblyUnitSystem* meets the timing requirement `t3`.