# Specifying Parallel and Distributed Systems in Object-Z: The Lift Case Study


Jin Song Dong  Lin Zucconi

jsdong,zucconi@cbr.dit.csiro.au

Software Engineering Group
Mathematical and Information Sciences
CSIRO
Canberra, ACT 2601, Australia

Roger Duke

rduke@cs.uq.oz.au

Software Verification Research Centre
Department of Computer Science
The University of Queensland
Brisbane, Q 4072, Australia


April 1997


**Abstract**

*There has been an increasing emphasis on formality in software system specification in the last few years. A number of standards bodies are recommending the use of formal notations for specifying software systems. Parallel and distributed systems have their own complex features such as: the concurrent interactions between various system components; the reactive nature of the systems; various message passing schemes between system components. Object-Z is an extension to the Z language specifically to facilitate specification in an object-oriented style. Because parallel and distributed systems are typically complex systems, the extra structuring afforded by the various Object-Z modelling constructs (i.e. the class, object containment constructs, and various composite operation expressions) enables the various hierarchical relationships and the communication between system components to be succinctly specified. Object-Z history invariants allow system temporal properties to be specified as well. In this paper, the use of Object-Z in the specification of parallel and distributed systems is demonstrated by presenting a case study based on a multi-lift system. To enhance the understandability of the formal model, OMT notation is used to grasp the static structure of the system, and a finite state machine diagram is used to highlight the system behaviour.*


## 1 Introduction

Formal methods are considered to be an excellent technique for achieving high integrity in software systems. Thus, there has been an increasing emphasis on formality in software system specification in the last few years. As a formal specification language can provide a precise system model which is implementation language independent, a number of standards bodies are recommending the use of formal notations for specifying software systems. For example, Z[23, 14] is recommended to define the Open Distributed Systems ODP system[16]. Object-Z[12, 19, 13] is recommended in standards work on presentation environments for multi-media objects (the PREMO project) [18]. Object-Z has also been used to define the information model of ODP trader[8] and ODP type management[1].

Parallel and distributed systems have their own complex features such as: the concurrent interactions between various system components; the reactive nature of the systems; various message passing schemes between system components. Object-Z is an extension to the Z language specifically to facilitate specification in an object-oriented style. Because parallel and distributed systems are typically complex systems, the extra structuring afforded by the various Object-Z modelling constructs (i.e. the class, object containment constructs, and various composite operation expressions) enables the various hierarchical relationships and the communication between system components to be succinctly and clearly specified. To illustrate this point, we use the well-known multi-lift system case study to demonstrate the use of Object-Z in the specification of parallel and distributed systems. To enhance the understandability of the formal Object-Z model, graphical notation OMT[20] is used to grasp the lift system static structure, and a finite state machine diagram is used to highlight the essential behaviour of the system.

The multi-lift system is a commonly used example to demonstrate the expressive power of various specification techniques in modelling concurrent reactive systems. It is not a trivial example because of the complexity caused by inherent concurrent interaction in the system[24]. A number of formal notations have been applied to the lift system. For example, CSP[15] is applied to the lift system in [21], UNITY model [3] is applied in [4] and Constraint Nets[25] is also applied in [26]. As those approaches all have a flat model of the lift (i.e. the system components are not modelled separately), the concurrent interaction between the lift system components are not clearly specified. Our approach is quite different: we model every component of the lift system as an object and model the lift system as a composite object which consists of the component objects. The concurrent interaction of the lift system components is modelled as the objects concurrently function and communicate in the system composite object. One aim of this approach is to achieve a reusable and extendible model.

Section 2 briefly describes the Object-Z notation, the example used to illustrate the Object-Z notation is also used as a generic library component which will be instantiated to define a lift system component. Section 3 informally describes the lift system requirement and expected properties. Section 4 presents the formal model of the lift system in Object-Z. Section 5 demonstrates the extendibility of the lift system model. Section 6 concludes the paper and points out needed 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 [23] 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 queue example

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

$Queue[T]$

$items : \mathrm{seq}\ T$
$\Delta$
$size : \mathbb{N}$
---
$size = \# items$

$\mathit{INIT}$
$items = \langle\, \rangle$

$Join$
$\Delta(items)$
$item? : T$
---
$items' = items \frown \langle item? \rangle$

$Leave$
$\Delta(items)$
$item! : T$
---
$items \neq \langle\, \rangle$
$items = \langle item! \rangle \frown items'$

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, it has one 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 queue contains no elements of $T$ (i.e. *items* is the empty sequence).

The remaining two schemas are operation schemas. Operation schemas have a $\Delta$-list of those attributes whose values may change. By convention, no $\Delta$-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 *Join* appends a given input *item?* to the existing sequence of items (an identifier ending in '?' denotes an input). Operation *Leave* 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 $\Delta$ 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 $\Delta$-list of every operation. In this case, the value of *size* is always subject to change whenever operation *Join* or *Leave* is invoked. For a detailed discussion on secondary attributes see [10].

**Inheritance**

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.

As an example of inheritance, consider the derivation of a generic non-repeated queue from $Queue[T]$.

$$
\begin{array}{|l}
\hline
NRQueue[T] \\
Queue[T] \\
\hline
size = \#(\mathrm{ran}\, items) \\
\hline
Remove \\
\Delta(items) \\
item? : T \\
\hline
items' = squash(items \rhd \{item?\}) \\
\hline
\end{array}
$$

The class invariant $size = \#(\mathrm{ran}\, items)$ further restricts the queue to be non-repeated. Operation *Remove* will remove a given element from the queue and preserve the original order. Note that if $item? \notin \mathrm{ran}\, items$, then $items' = items$. To indicate that the environment selects the item to be removed from the queue, the parameter $item?$ is modelled as an input rather than an output (where the *Leave* operation is modelled).

This generic queue class will be instantiated to define the request queue for lift system in Section 4.

**History Invariants**

A *history invariant* is an optional predicate over histories of objects of the class expressed in temporal logic. Such predicates further constrain possible behaviour. For example, suppose we wish to specify in the generic queue that whenever *Leave* or *Delete* is continuously enabled it must eventually occur.

$$
\begin{array}{|l}
\hline
FairQueue[T] \\
Queue[T] \\
\hline
\Box(items \neq \langle\rangle \;\Rightarrow\; \Diamond((\bigcirc \#items) < \#items)) \\
\hline
\end{array}
$$

In this example, the history invariant is expressed using the temporal logic operators $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next). For an introduction to temporal logic and a semantic description of these operators see Moszkowski[17]. History invariants are particularly useful in capturing the temporal properties of the lift system, e.g. the property that all requests will be satisfied by lifts is specified by using temporal predicates in Section 3.

**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 [12] and [19].

## 3 Multiple Lift System Requirements

A lift system for an multi-floor building consist of multiple elevators. Inside the elevator there is a panel of floor buttons each of which indicates a destination floor. Outside the elevator there are two direction buttons on each floor which call for service, one for up and the other for down (except the first floor and the top floor on which only one button is needed). Any button can be pushed at any time. Any external floor button will be on from the time it is pushed until the elevator with the same travel direction stops at the floor and opens the door. Any internal 'on' button of a lift will be turned off when the lift visits the corresponding floor. The following system properties must be captured in the model:

- safety property (S): The lift door must be closed before any movement (Up or Down).
- system property (P1): When a lift has no request, it should remain at its final destination.
- system property (P2): The lift has to satisfy an external floor request only if it goes in the same direction or if the request is the external destination of the lift.
- system property (P3): When a request arrives from a floor the system will put the request at the end of the external request queue.
- system property (P4): If an external request in the queue is serviced by some lift on the way Up or Down, then this request should be removed from the request queue.
- temporal property (T1): whenever there are some (one or more) idle lifts and the external service queue is not empty, the longest waiting request is removed from the queue and assigned to one idle lift as the temporal final destination.
- temporal property (T2): All request for lifts from floors must be serviced eventually.
- temporal property (T3): All internal requests for lifts must be serviced eventually.
- temporal property (T4): If a lift's door is open, eventually it will be closed.

## 4 Lift System Specification

Firstly an OMT diagram (Figure 1) is presented that outlines the class structure of the lift system.

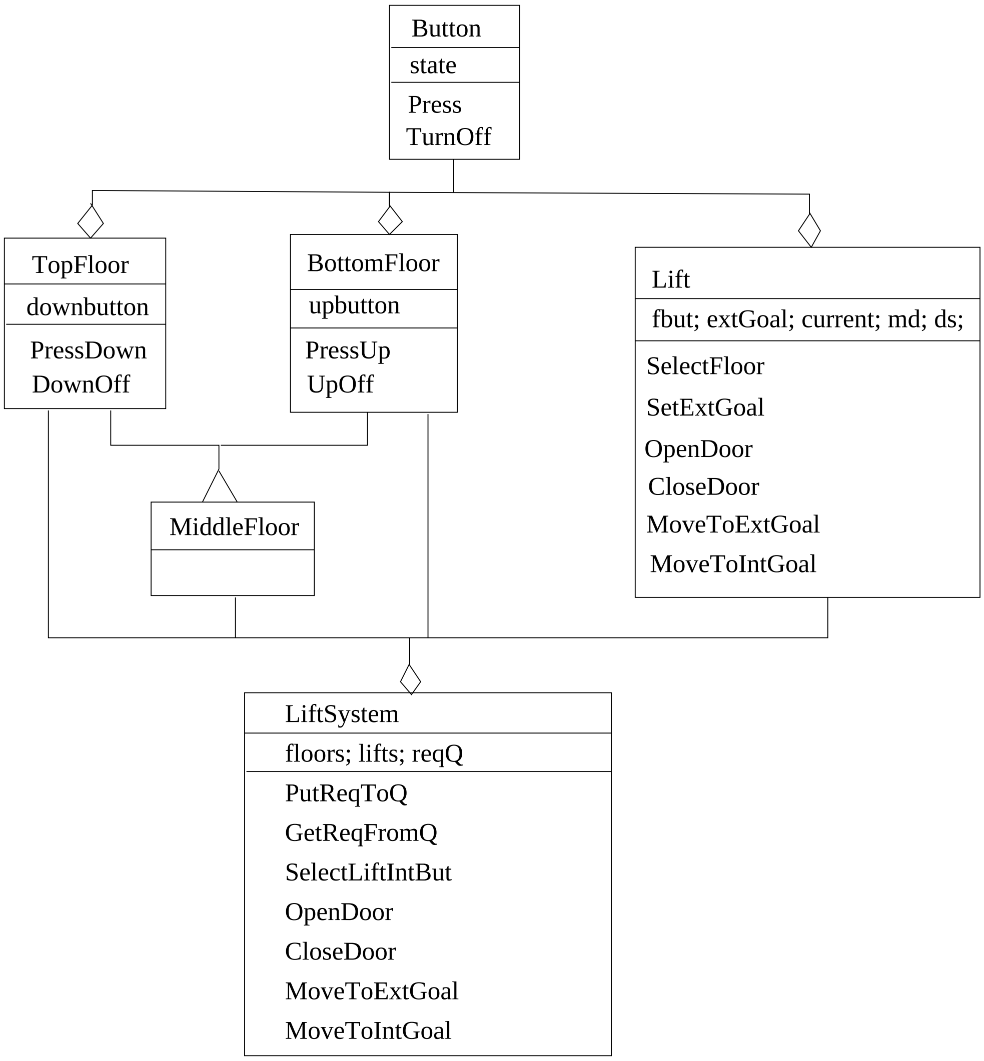



Figure 1: The Lift System's Structure in OMT

## 4.1 Buttons (Basic Component Objects)

The basic component objects of the lift system is the buttons on the floors or inside the lifts. Buttons have a common behaviour; they can be pushed On by people and turned Off by the system. The button status is defined as:

$$ButtonStatus ::= On \mid Off$$

The button is defined as:

*Button*

$state : ButtonStatus$

*INIT*

$state = Off$

*Press*

$\Delta(state)$

$state' = On$

*TurnOff*

$\Delta(state)$

$state' = Off$

## 4.2 Floors

The top-floor and bottom-floor can be defined as:

$$
\begin{array}{l}
\textbf{TopFloor} \\
\hline
\quad downbutton : Button_{©} \\
\quad \textit{INIT} \\
\quad \hline
\quad downbutton.\textit{INIT} \\
PressDown \mathrel{\widehat{=}} downbutton.Press \\
DownOff \mathrel{\widehat{=}} downbutton.TurnOff \\
\hline
\Box(downbutton.state = On \\
\qquad \Rightarrow \Diamond downbutton.state = Off)
\end{array}
$$

$$
\begin{array}{l}
\textbf{BottomFloor} \\
\hline
\quad upbutton : Button_{©} \\
\quad \textit{INIT} \\
\quad \hline
\quad upbutton.\textit{INIT} \\
PressUp \mathrel{\widehat{=}} upbutton.Press \\
UpOff \mathrel{\widehat{=}} upbutton.TurnOff \\
\hline
\Box(upbutton.state = On \\
\qquad \Rightarrow \Diamond upbutton.state = Off)
\end{array}
$$

Note that the subscript '©' (unique object containment[1][9]) indicates that the button object is contained in the floor object.

A floor in between the top and bottom floors is defined by (multiple) inheriting the models of the top-floor and bottom-floor as:

$$
\begin{array}{l}
\textbf{MiddleFloor} \\
TopFloor, BottomFloor \\
\hline
\quad downbutton \neq upbutton
\end{array}
$$

Note that the inheritance used here is a subclassing for specification code reuse, not a subtyping.

The history invariants captures the system temporal property (T2), "All request for lifts from floors must be serviced eventually," as floor buttons On represent the external requests.

As a floor can be either top-floor or bottom-floor or middle-floor, the general type of a floor is defined as a class-union[7, 5]:

$$Floor \mathrel{\widehat{=}} TopFloor \cup BottomFloor \cup MiddleFloor$$

### 4.3 Lifts

The lift move direction is defined as:

$$MoveDirection ::= Up \mid Down$$

[1] Unique object containment ensures that no object directly or indirectly contains itself; and no object is directly contained in two distinct objects. For a detailed discussion see [9].

The lift-door state is defined as:

$$DoorStatus ::= Open \mid Closed$$

A lift consists a list of buttons in a one-one correspondence with floors. The main status of the lift is the move direction (Up or Down) and the lift door status (Open or Close). It is modelled as

$$
\begin{array}{l}
\textbf{Lift} \\
\hline
\upharpoonright(\textit{INIT}, SetExtGoal, OpenDoor, CloseDoor, \\
MoveToExtGoal, MoveToIntGoal) \\
\\
\quad fbuts : \operatorname{seq} Button_{©} \\
\quad md : MoveDirection \\
\quad ds : DoorStatus \\
\quad current : \mathbb{N} \\
\quad extGoal : \mathbb{P}(\mathbb{N} \times MoveDirection) \\
\quad \Delta \\
\quad inUse : \mathbb{B} \\
\quad \hline
\quad current \in \operatorname{dom} fbuts \\
\quad inUse = (On \in \{b : \operatorname{ran} fbuts \bullet b.state\} \\
\qquad \lor extGoal \neq \varnothing) \\
\\
\quad \textit{INIT} \\
\quad \hline
\quad \forall b : \operatorname{ran} fbuts \bullet b.\textit{INIT} \\
\quad md = Up \\
\quad current = 1 \\
\quad extGoal = \varnothing \\
\quad ds = Closed \\
\\
\quad \textbf{ToSelectFloor} \\
\quad \hline
\quad \Delta(fbuts) \\
\quad n? : \operatorname{dom} fbuts \\
\quad \hline
\quad n? \neq current \\
\quad fbuts(n?).state = Off \\
\\
\quad \textbf{SetExtGoal} \\
\quad \hline
\quad \Delta(extGoal) \\
\quad n? : \operatorname{dom} fbuts \\
\quad m? : MoveDirection \\
\quad \hline
\quad extGoal' = \{(n?, m?)\} \\
\\
\quad SelectFloor \mathrel{\widehat{=}} ToSelectFloor \bullet fbuts(n?).Press \\
\\
\quad \textbf{ToOpenDoor} \\
\quad \hline
\quad \Delta(ds, md, extGoal) \\
\quad \hline
\quad ds = Close \land ds' = Open \\
\quad fbuts(current).state = On \lor \\
\qquad \exists\, g : extGoal \bullet current = first(g) \\
\quad \exists\, g : extGoal \bullet \\
\qquad\quad (current = first(g) \land \\
\qquad\quad md' = second(g) \land \\
\qquad\quad extGoal' = \varnothing) \lor \\
\qquad \nexists g : extGoal \bullet \\
\qquad\quad (current = first(g) \land \\
\qquad\quad md' = md \land \\
\qquad\quad extGoal' = extGoal)
\end{array}
$$

$$OpenDoor \mathrel{\widehat=} ToOpenDoor \bullet fbuts(current).TurnOff$$

$$\begin{array}{l}
\textit{CloseDoor} \\
\hline
\Delta(ds) \\
\hline
ds = Open \wedge ds' = Close
\end{array}$$

$$\begin{array}{l}
\textit{MoveToExtGoal} \\
\hline
\Delta(current, md) \\
\hline
ds = Closed \\
extGoal \neq \varnothing \\
fbuts(current).state = Off \\
\nexists g : extGoal \bullet current = first(g) \\
\exists\, g : extGoal \bullet \\
\quad (first(g) > current \wedge \\
\quad md' = Up \wedge \\
\quad current' = current + 1) \vee \\
\quad (first(g) < current \wedge \\
\quad md' = Down \wedge \\
\quad current' = current - 1)
\end{array}$$

$$\begin{array}{l}
\textit{MoveToIntGoal} \\
\hline
\Delta(current, md) \\
\hline
ds = Closed \\
extGoal = \varnothing \\
inUse \wedge fbuts(current).state = Off \\
md = Up \Rightarrow \\
\quad (\exists\, i : \mathrm{dom}\, fbuts \bullet \\
\quad\; (i > current \wedge \\
\quad\; fbuts(i).state = On) \Rightarrow \\
\qquad (md' = Up \wedge \\
\qquad current' = current + 1)) \\
\quad \vee (\nexists i : \mathrm{dom}\, fbuts \bullet \\
\qquad (i > current \wedge \\
\qquad fbuts(i).state = On) \Rightarrow \\
\qquad\quad (md' = Down \wedge \\
\qquad\quad current' = current - 1)) \\
md = Down \Rightarrow \\
\quad (\exists\, i : \mathrm{dom}\, fbuts \bullet \\
\qquad (i < current \wedge \\
\qquad fbuts(i).state = On) \Rightarrow \\
\qquad\quad (md' = Down \wedge \\
\qquad\quad current' = current - 1)) \\
\quad \vee (\nexists i : \mathrm{dom}\, fbuts \bullet \\
\qquad (i < current \wedge \\
\qquad fbuts(i).state = On) \Rightarrow \\
\qquad\quad (md' = Up \wedge \\
\qquad\quad current' = current + 1))
\end{array}$$

$$\begin{array}{l}
\hline
\forall\, b : \mathrm{ran}\, fbuts \bullet \\
\quad \Box(b.state = On \Rightarrow \\
\qquad \Diamond b.state = Off) \\
\Box(ds = Open \Rightarrow \Diamond ds = Closed)
\end{array}$$

The projection symbol '$\upharpoonright$' indicates visibility, i.e. introduces the list of those features (local schemas) which are accessible via the dot notation. If the list is omitted, as in previous examples, all features are visible.

In the class *Lift*, the attribute *fbuts* (floor buttons) models the panel of floor buttons where the index of the sequence models the floor numbers. The attribute *md* and *ds* models the move direction status and lift-door status. The attribute *current* models the floor position where the lift is currently at. The attribute *extGoal* represents an external request which is assigned to the lift as a temporary destination.

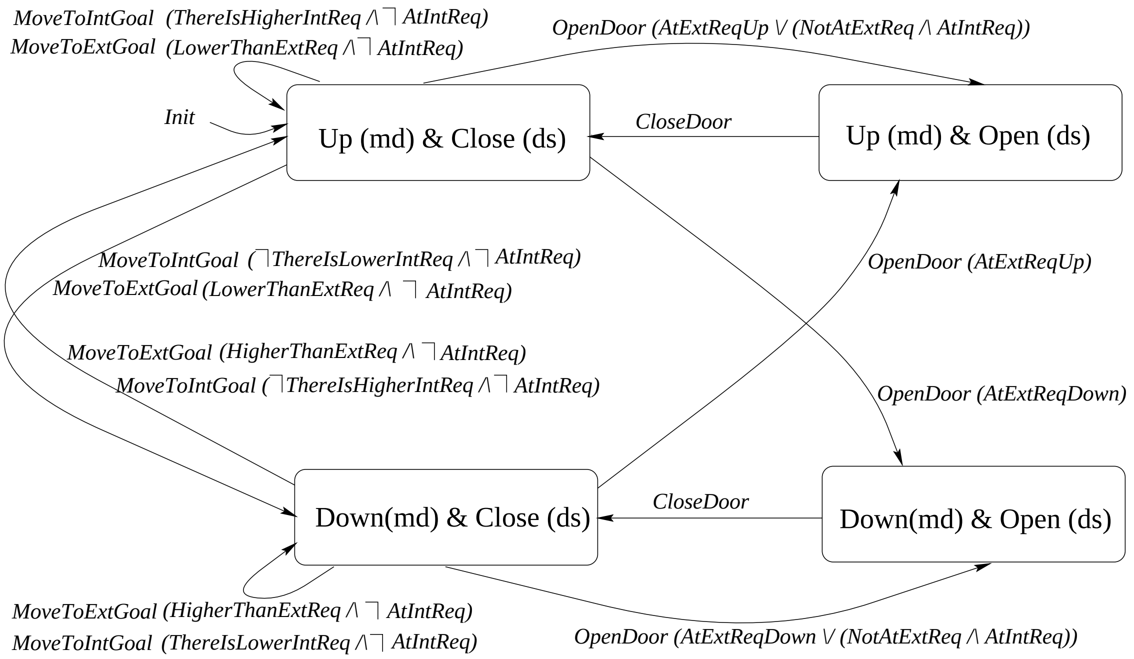
Inside a lift, a floor button can be selected by people. This action is modelled by operation *SelectFloor*. Note that the notation $operation_1 \bullet operation_2$ denotes environment enrichment in that the schema text of $operation_1$ enriches the environment in which $operation_2$ is interpreted. This environment enrichment frame also can used to model multiple concurrent operation interactions (see the next sub-section). The lift door can be opened (*OpenDoor*) and closed (*CloseDoor*).

There are two kinds of move operations: one is driven by the external request (*MoveToExtGoal*), the other is driven by the internal request (*MoveToIntGoal*). The lift move direction is dominated by the external request (if it has been assigned to the lift) rather than the internal request. This is ensured by the precondition $extGoal = \varnothing$ of the operation *MoveToIntGoal*.

The system safety property (S), "The lift door must be closed before any movement (Up or Down)," is ensured by the predicate $ds = Closed$ in the preconditions of both the operation *MoveToExtGoal* and *MoveToIntGoal*. Also, as preconditions of both the operation *MoveToExtGoal* and *MoveToIntGoal* requires that there is an external and/or internal request, therefore the system property (P1), "When a lift has no request, it should remain at its final destination", is ensured. The system property (P2), "The lift has to satisfy an external floor request only if it goes in the same direction or if the request is the external destination of the lift", is also captured in the definition of the operation *MoveToExtGoal*.

As the definition of some operations *ToOpenDoor*, *MoveToExtGoal* and *MoveToIntGoal* are relatively complex, to improve the understandability of the model, we present a finite state machine diagram Figure 2 to highlight the essential behaviour of the lift. Because the main characteristic states of the lift are the move direction and the door status. A permutation of the cross state space of *MoveDirection* and *DoorStatus* forms the states of the finite state machine for the lift. Transactions between states are labelled with Object-Z operations with instantiated preconditions. The instantiated preconditions for each operations are also partitioned and labelled so that they can be presented in a reusable fashion. We believe this improves the readability of the finite state machine diagram for a complex system.

The history invariants of the *Lift* class capture the temporal properties (T3), "All internal requests for lifts must be serviced eventually", and (T4), "If a lift's door is open, eventually it will be closed".




*AtExtReqDown* == ∃ *g:extGoal * current = first(g) ∧ second(g) = Down*

*AtExtReqUp* == ∃ *g:extGoal * current = first(g) ∧ second(g) = Up*

*AtIntReq* == *fbuts(current).state = On*

*NotAtExtReq* == ∄ *g:exGoal*current = first(g)*

*LowerThanExtReq* == ∃ *g:extGoal * current < first(g)*

*HigherThanExtReq* == ∃ *g:extGoal * current > first(g)*

*ThereIsHigherIntReq* == ∃ *i: dom fbuts * (i > current ∧ fbuts(i).state = On)*

*ThereIsLowerIntReq* == ∃ *i: dom fbuts * (i < current ∧ fbuts(i).state = On)*

Figure 2: The essential behaviour of a lift in a finite state machine diagram

### 4.4 The Lift System

Firstly, the external request queue is modelled by inheriting and instantiating the generic class $NRQueue$ (from Section 2) with a fairness condition captured by the system invariant.

$$
\begin{array}{|l}
\textit{RequestQueue} \\
\hline
NRQueue[\mathbb{N} \times MoveDirection] \\
\hline
\forall\, i : \mathrm{ran}\; items \bullet \Box(\Diamond i \notin \mathrm{ran}\; items) \\
\hline
\end{array}
$$

The lift system consists of a list of floors, a group of lifts and a service queue. It is modelled as:

$$
\begin{array}{|l}
\textit{LiftSystem} \\
\hline
\upharpoonright(\textsc{Init}, PutReqToQ, GetReqFromQ, \\
SelectLiftIntBut, OpenDoor, CloseDoor, \\
MoveToExtGoal, MoveToIntGoal) \\
\\
\begin{array}{|l}
\hline
floors : \mathrm{seq}\; Floor \\
lifts : \mathbb{P}\; Lift \\
reqQ : RequestQueue \\
\hline
\forall\, l : lifts \bullet \\
\quad \# l.fbuts = \# floors \\
floors(1) \in BottomFloor \\
floors(\# floors) \in TopFloor \\
\forall\, i : (1 \ldots (\# floors - 1)) \bullet \\
\quad floors(i) \in MiddleFloor \\
\hline
\end{array}
\end{array}
$$

$$\begin{array}{|l}
\underline{\quad \textit{INIT} \quad}\\
\forall f : \mathrm{ran}\, \mathit{floors} \bullet f.\mathit{INIT}\\
\forall l : \mathit{lifts} \bullet l.\mathit{INIT}
\end{array}$$

$$\begin{array}{|l}
\underline{\quad \mathit{WhichFloorBut} \quad}\\
n? : \mathrm{dom}\, \mathit{floors}\\
m? : \mathit{MoveDirection}
\end{array}$$

$\mathit{PressFloorButUp} \mathrel{\widehat=} \mathit{WhichFloorBut} \bullet$
$\quad [m? = \mathit{Up}] \wedge \mathit{floor}(n?).\mathit{PressUp}$

$\mathit{PressFloorButDown} \mathrel{\widehat=} \mathit{WhichFloorBut} \bullet$
$\quad [m? = \mathit{Down}] \wedge \mathit{floor}(n?).\mathit{PressDown}$

$\mathit{PressFloorBut} \mathrel{\widehat=}$
$\quad \mathit{PressFloorButUp} \mathbin{[]} \mathit{PressFloorButDown}$

$\mathit{PutReqToQ} \mathrel{\widehat=} (\mathit{PressFloorBut} \wedge \mathit{reqQ}.\mathit{Join}) \bullet$
$\quad [\mathit{item}? = (n?, m?)]$

$$\begin{array}{|l}
\underline{\quad \mathit{FindIdleLift} \quad}\\
n? : \mathrm{dom}\, \mathit{floors}\\
m? : \mathit{MoveDirection}\\
l! : \mathit{lifts}\\
\hline
\exists\, l : \mathit{lifts} \bullet\\
\quad \neg\, l.\mathit{inUse} \wedge l.\mathit{ds} = \mathit{Closed}
\end{array}$$

$\mathit{GetReqFromQ} \mathrel{\widehat=} (\mathit{reqQ}.\mathit{Leave} \wedge \mathit{FindIdleLift}) \bullet$
$\quad ([\mathit{item}! = (n?, m?)] \wedge l!.\mathit{SetExtGoal})$

$\mathit{SelectLiftIntBut} \mathrel{\widehat=}$
$\quad [l? : \mathit{lifts}] \bullet l?.\mathit{SelectFloor}$

$$\begin{array}{|l}
\underline{\quad \mathit{LiftStop} \quad}\\
l : \mathit{lifts}\\
\hline
l.\mathit{fbuts}(l.\mathit{current}).\mathit{state} = \mathit{On} \vee\\
\quad (l.\mathit{md} = \mathit{Up} \wedge\\
\quad \mathit{floors}(l.\mathit{current}).\mathit{upbutton} = \mathit{On}) \vee\\
\quad (l.\mathit{md} = \mathit{Down} \wedge\\
\quad \mathit{floors}(l.\mathit{current}).\mathit{downbutton} = \mathit{On})
\end{array}$$

$\mathit{OpenDoor} \mathrel{\widehat=} \mathit{LiftStop} \bullet$
$\quad l.\mathit{OpenDoor} \wedge (\mathit{reqQ}.\mathit{Remove} \bullet$
$\qquad [\mathit{item}? = (l.\mathit{current}, l.\mathit{md})]) \wedge$
$\qquad [l.\mathit{md} = \mathit{Up} \wedge$
$\qquad \mathit{floors}(l.\mathit{current}).\mathit{upbutton} = \mathit{On}] \bullet$
$\qquad\quad \mathit{floors}(l.\mathit{current}).\mathit{UpOff}$
$\qquad []$
$\qquad [l.\mathit{md} = \mathit{Down} \wedge$
$\qquad \mathit{floors}(l.\mathit{current}).\mathit{downbutton} = \mathit{On}] \bullet$
$\qquad\qquad \mathit{floors}(l.\mathit{current}).\mathit{DownOff}$

$\mathit{CloseDoor} \mathrel{\widehat=} [l : \mathit{lifts}] \bullet l.\mathit{Close}$

$\mathit{MoveToExtGoal} \mathrel{\widehat=}$
$\quad [l : \mathit{lifts}] \bullet l.\mathit{MoveToExtGoal}$

$\mathit{MoveToIntGoal} \mathrel{\widehat=}$
$\quad [l : \mathit{lifts}] \bullet l.\mathit{MoveToIntGoal}$

$$\begin{array}{|l}
\hline
\Box((\exists\, l : \mathit{lifts} \bullet\\
\quad \neg\, l.\mathit{inUse} \wedge l.\mathit{ds} = \mathit{Closed}) \wedge\\
\quad \mathit{reqQ} \neq \varnothing) \Rightarrow \bigcirc(\, \mathit{op}) = \mathit{GetReqFromQ})
\end{array}$$

In the class *LiftSystem*, the floors in the lift system are modelled as a sequence *floors* which the index models the position of the floor, while the lifts are modelled as a set *lifts* where order of the lifts is not important. The class invariant captures that the panel buttons of any lift must match to the floor numbers.

The system property (P3) states that "when a floor button is pushed, a service request is inserted at the end of the external request queue." This concurrent interaction between a floor and a service queue is modelled by the conjunction operator '$\wedge$' and the environment enrichment frame '$op_1 \bullet op_2$' in the definition of *PutReqToQ*. In a similar fashion, the operation *GetReqFromQ* models the concurrent interaction between the request queue and the chosen idle lift. The operation *OpenDoor* also captures a complex concurrent interaction between a floor, a lift and the request queue. The system property (P4), "If a external request in the queue is serviced by some lift on the way Up or Down, then this request should be removed from the request queue," is captured in the definition of the operation *OpenDoor*. Note that 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 semantics of these composite operators $\wedge$ and $[]$ are defined in [2, 22].

The history invariant captures temporal property (T1), "whenever there are some (one or more) idle lifts and the external service queue is not empty, the longest waiting request is removed from the queue and assigned to one idle lift as the temporary final destination."

Now the lift system is completely modelled according to the requirements in Section 3. In the next section, we demonstrate the extendibility of this model by adding a more stringent requirement to the system.

## 5 Extendibility

Suppose when an external request arrives from a floor and leaves from the service queue, the nearest idle lift has to be selected to satisfy it. To capture this, firstly a 'distance' function is defined over natural numbers:

$$\begin{array}{|l}
\mathit{distance} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}\\
\hline
\forall\, a, b : \mathbb{N} \bullet\\
\quad (a \geq b \Rightarrow \mathit{distance}(a, b) = a - b)\\
\quad (a < b \Rightarrow \mathit{distance}(a, b) = b - a)
\end{array}$$

Then the lift system can be simply extended as:

$$\begin{array}{|l}
\underline{\quad \mathit{SmartLiftSystem} \quad}\\
\mathit{LiftSystem}\\
\begin{array}{|l}
\underline{\quad \mathit{FindIdleLift} \quad}\\
\mathit{distance}(l!.\mathit{current}, n?) =\\
\quad \mathit{min}\{l : \mathit{lifts} \mid\\
\qquad \neg\, l.\mathit{inUse} \wedge l.\mathit{ds} = \mathit{Closed} \bullet\\
\qquad\quad \mathit{distance}(l.\mathit{current}, n?)\}
\end{array}
\end{array}$$

The disruption to the existing model is minimal.

## 6 Conclusion and Further Work

In this paper, the use of the Object-Z notation in the specification of parallel and distributed systems has been demonstrated by presenting a case study based on a multi-lift system. The essential features of parallel and distributed systems, such as: the concurrent interactions between various system components; the reactive nature of the systems; various message passing schemes between system components, are precisely captured by various Object-Z constructs. The extra structuring afforded by the various Object-Z modelling techniques (i.e. the class encapsulation, object containment constructs, and various composite operation expressions) enables the various hierarchical relationships and the communication between system components to be succinctly specified, leading to an extendible and reusable model. Additional history invariants allow system temporal properties to be specified as well. The use of the OMT notation for grasping the static structure of the system and the use of a finite state machine diagram for highlighting the system behaviour greatly assist the understanding the formal model. This, in effect, outlines an integrated approach (linking various methods) to the specification of parallel and distributed systems.

Additional planned work will be to consider real-time issues (not just temporal ordering properties) of parallel and distributed systems, e.g. to extend the lift model to capture the time duration constraints of the system operations. We believe it is feasible to apply the real-time extensions to Object-Z investigated in [6, 11] to capture the real-time constraints in parallel and distributed systems.

## Acknowledgements

We wish to thank Ian Gorton for suggesting the 'Lift' example as the case study and his comments of the early drafts of this paper.

## References

[1] W. Brookes and J. Indulska. "ODP Types and Their Management: an Object-Z Specification". In *Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing, ICODP'95*. North-Holland, February 1995.

[2] S. Butler and R. Duke. Defining Composition Operators for Object Interaction. To appear in *Object Oriented Systems*, 1996.

[3] K. M. Chandy and J. Misra. *Parallel Programming Design*. Addison-Wesley, 1988.

[4] H. C. Cunningham, V.R.Shan, and S. Shen. Devising a Formal Specification for an Elevator Controller. TR 94-10, Department of Computer and Information Science, University of Mississippi, USA, 1994.

[5] 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.

[6] J.S. Dong, J. Colton, and L. Zucconi. A Formal Object Approach to Real-Time Specification. In *the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, Seoul, Korea, December 1996. IEEE Computer Society Press.

[7] 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.

[8] J.S. Dong and R. Duke. An Object-Oriented Approach to the Formal Specification of ODP Trader. In J. Meer, B. Mahr, and S. Storp, editors, *Open Distributed Processing, II (ICODP'93)*, pages 341–352, Berlin, 1994. North-Holland.

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

[10] 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.

[11] J.S. Dong and L. Zucconi. A Framework for Adding Time into Formal Object Models. In *the 3rd IEEE International Workshop on Object-oriented Real-time Dependable System (WORDS'97)*, Newport Beach, California, February 1997. IEEE Computer Society Press.

[12] 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.

[13] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

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

[15] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[16] ISO/IEC JTC1/SC21. *ODP Trading Function*, June 1995.

[17] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

[18] G. J. Reynolds, D. A. Duce, and D. J. Duke. *Report of the ISO/IEC JTC1/SC24 Special Rapporteur Group on Formal Description Techniques*, June 1994.

[19] 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.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[21] M. D. Schwartz and N. M. Delisle. Specifying A Lift Control System with CSP. In *The 4th IEEE International Workshop on Software Specification and Design (IWSSD'87)*, pages 21–27, Monterey, California, April 1987. IEEE Computer Society Press.

[22] 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.

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

[24] J.C.P. Woodcock, S. King, and I.H. Sorensen. Mathematics for Specification and Design: The Problem with Lifts. In *The 4th IEEE International Workshop on Software Specification and Design (IWSSD'87)*, pages 265–268. IEEE Computer Society Press, 1987.

[25] Y Zhang and A. K. Mackworth. Constraint Nets: A semantic model of real-time embedded system . Technical Report 92-10, Computer Science Department, University of British Columbia, 1992.

[26] Y Zhang and A. K. Mackworth. Design and Analysis of Embedded Real-Time Systems: An Elevator Case Study. Technical Report 93-04, Computer Science Department, University of British Columbia, 1993.