Network Topology and a Case Study in TCOZ

Brendan Mahony
Information Technology Division
Defence Science and Technology Organisation
Brendan.Mahony@dsto.defence.gov.au

Jin Song Dong
School of Computing
National University of Singapor e
dongjs@comp.nus.edu.sg

September 21, 1998

Abstract

Object-Z is strong in modeling the data and operations of complex systems. However, it is weak in specifying real-time and concurrent systems. The Timed Communicating Object-Z (TCOZ) extends Object-Z notation with Timed CSP's constructs. TCOZ is particularly well suited for specifying complex systems whose components have their own thread of control. This paper demonstrates expressiveness of the TCOZ notation through a case study on specifying a multi-lift system that operates in real-time. [1]

1 Introduction

Many formal specification and design notations have tended to concentrate either on data modeling and algorithmic concerns (eg. Z, VDM, etc.) or else on process control concerns (eg. CSP, CCS, StateCharts, etc.). Parallel and distributed systems often have intricate system states and process control structures involving concurrency and real-time interactions. To formalise such systems, it is necessary to have a notation which is able to capture both the data/algorithmic issues and the process behaviour issues in a smoothly integrated, but also highly structured and modular, manner.

Timed Communicating Object Z (TCOZ) builds on the respective strengths of the Object-Z and Timed CSP notations in order to provide a single notation for modeling both the state and timed process aspects of complex systems. The notion of blending Object-Z with CSP has been suggested by Fischer, Smith and Derick [12, 32]. TCOZ is novel in that it includes timing primitives; properly seperates process control and data/algorithm issues; fully integrates notions of refinement from both languages; supports the modeling of true multi-threaded concurrency; and distinguishes the notion of active and passive objects.

The basic TCOZ notation has been briefly described in an introductory paper by these authors [25] and this paper further enhances the TCOZ notation with a new network topology operator which simplifies the description of complex networks of communicating processes. However the main purpose of this paper is to demonstrate the expressiveness of TCOZ through a case study in the specification of a real-time multi-lift system. The lift system is chosen because it is a standard case study,

[1]This paper is appeared in *ZUM'98 The* 11^{th} *International Conference of Z Users*, volume 1493 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

having been treated in a large number of specification notations including Object-Z and CSP. Developing a TCOZ version of the lift case study puts us in a good position to make qualitative assessments of the relative merits of TCOZ.

It is assumed that the reader has some familiarity with both Object-Z [10] and CSP [17], since the mechanics of blending the two notations is considered only briefly in Section 2. A glossary of TCOZ language features may be found in Appendix A. The TCOZ specification of the lift case study is presented and evaluated in Section 3 and the features of TCOZ compared to similar languages in Section 4.

2 Aspects of TCOZ

TCOZ is essentially just a blending of Object-Z [10] and Timed CSP [29], for the most part preserving them as proper sub-languages of the blended notation. The essence of this blending is the identification of Object-Z operation specification schemas with terminating CSP processes. Thus operation schemas and CSP processes occupy the same syntactic category, operation schema expressions may appear wherever processes may appear in CSP and CSP process definitions may appear wherever operation definitions may appear in Object-Z. The primary specification structuring device in TCOZ is the Object-Z classing mechanism.

In this section we briefly consider the aspects of TCOZ which help to bring the two notations together. A detailed introduction to TCOZ and its Timed CSP and Object-Z features may be found elsewhere [25].

2.1 Declaring channels

CSP channels are given an independent, first class role in TCOZ. This allows the communications and control topology of a network of objects to be designed orthogonally to their class structure.

In order to support the role of CSP channels, the state schema convention is extended to allow the declaration of communication channels. If c is to be used as a communication channel by any of the operations of a class, then it must be declared in the state schema to be of type **chan**. Channels are type heterogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state variables, channels are viewed as global rather then as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects. The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby enhancing the modularity of system specifications.

2.2 A model of time

In TCOZ, all timing information is represented as real valued measurements in *seconds*, the SI standard unit of time [18]. Describing time and other physical quantities in terms of standard units of measurement is an important aspect of ensuring the completeness and soundness of specifications of real-time, reactive, and hybrid systems. In order to support the use of standard units of measurement, extensions to the Z typing system suggested by Hayes and Mahony [15] are adopted. Under this convention, time quantities are represented by the type

$$\mathbb{T} == \mathbb{R} \odot \mathsf{T},$$

where $\mathbb{R}$ represents the real numbers and T is the SI symbol for dimensions of time. Time literals consist of a real number literal annotated with a symbol representing a unit of time. For example, $3\,\mu\mathsf{s}$ is a literal representing a period of three microseconds, that is three millionths of the standard time unit, the second. All the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

2.3 Guards and preconditions

A novel CSP operator, the state-guard, is used to *block* or *enable* execution of an operation on the basis of an object's local state. For example, the operation $[a \geq 0] \bullet [\Delta(a) \mid a \geq 0 \land a' = \sqrt{a}]$ will replace the state variable a with its square root if a is positive otherwise it will *deadlock*, that is be blocked from executing. The blocking or enabling of this operation is achieved by the state guard $[a \geq 0] \bullet \_$ and not by the precondition $a \geq 0$ within the operation schema. If the operation schema alone is envoked with a negative, it will *diverge* rather than block. The difference between deadlock and divergence is that a divergence may be refined away by making an operation more robust, whilst a deadlock can never be refined away.

An additional function of state guards is as a substitute for CSP's indexed external choice operator. The process $[n : \mathbb{N} \mid 0 \leq n \leq 5] \bullet c?n \rightarrow P(n)$ may input any value of n between 0 and 5 (from channel c) as chosen by its environment. CSP's indexed internal choice is replaced by the operation schema and sequential composition. The process $[n! : \mathbb{N} \mid 0 \leq n! \leq 5];\ c!n \rightarrow P(n)$ may output any value of n between 0 and 5 according to its own designs.

2.4 Active and passive objects

Active objects have their own thread of control, while passive objects are controlled by other objects in a system. In TCOZ, an identifier MAIN is used to determine the behaviour of active objects of a given class. The MAIN operation is optional in a class definition. It only appears in a class definition when the objects of that class are active objects. Classes for defining passive objects will not have the MAIN definition, but may contain CSP process constructors. If ob_1 and ob_2 are active objects of the class C, then the asynchronous parallel composition behaviour the two objects can be represented as $ob_1 \mid\mid\mid ob_2$, which is a lax form of $ob_1.\mathrm{MAIN} \mid\mid\mid ob_2.\mathrm{MAIN}$

2.5 Complex network topologies

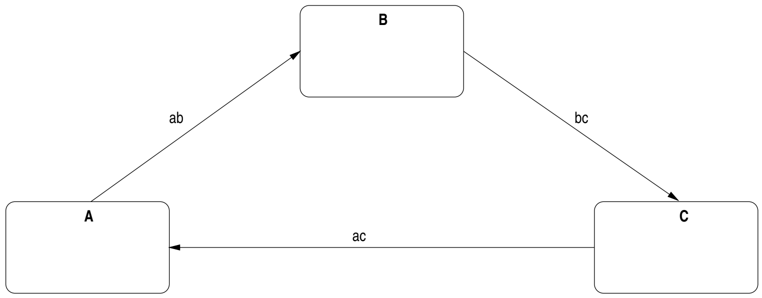

Figure 1: Two dimensional communication topology

The syntactic structure of the CSP synchronisation operator is convenient only in the case of pipe-line like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. For example, consider the communication topology shown in Figure 1, processes A and B communicate privately through the channel ab, processes A and C communicate privately through the channel ac, and processes B and C communicate privately through the channel bc. One CSP expression for such a network communication system is

$$(A[bc'/bc] \,|[\, ab, ac \,]|\, (B[ac'/ac] \,|[\, bc \,]|\, C[ab'/ab]) \setminus ab, ac, bc)[ab, ac, bc/ab', ac', bc'].$$

The hiding and renaming is necessary in order to cover cases such as C being able to communicate on channel ab.

The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology described so elegantly by Figure 1. We believe that network topologies can be better described by adopting a notation inspired by the graph-based approach embodying in Figure 1.

A *network topology abstraction* is is an expression of the form

$$(\| \; v_1, v_2, v_3 ... \bullet v_1 \xleftrightarrow{ch_{12}} v_2;\; v_2 \xleftrightarrow{ch_{23}} v_3;\; v_3 \xleftrightarrow{ch_{13}} v_1;\; ...).$$

The variables $v_1, v_2, v_3, ...$ are called the *formal network parameters* and the network connections

$$v_1 \xleftrightarrow{ch_{12}} v_2;\; v_2 \xleftrightarrow{ch_{23}} v_3;\; v_3 \xleftrightarrow{ch_{13}} v_1;\; ...$$

are called the *formal network topology*. It describes a finite graph in which each edge or *network connection* describes a single private channel connection between processes which have the role of graph nodes. Multiple connections between processes and connections between multiple processes over a single channel are represented by multiple connection expressions. In the latter case the channel becomes a party line between the various participating processes.

A network topology abstraction describes a *network constructor*, that is a function which builds a process network using its process arguments as nodes. The above network topology describes the following constructor.

$$\begin{array}{l}
\lambda\, v_1, v_2, v_3, ... \bullet \\
\quad (v_1[ch_{23}'/ch_{23}, ...] \,|[\, ch_{12}, ch_{13}, ... \,]| \\
\qquad (v_2[ch_{13}'/ch_{13}, ...] \,|[\, ch_{23}, ... \,]| \\
\qquad\quad (v_3[ch_{12}'/ch_{12}, ...] \,|[\, ... \,]| \\
\qquad\qquad \vdots \\
\qquad\quad) \\
\qquad) \\
\quad \setminus ch_{12}, ch_{13}, ch_{23}, ...)[ch_{12}/ch_{12}', ch_{13}/ch_{13}', ch_{23}/ch_{23}', ...]
\end{array}$$

Once again the hiding and renaming is necessary to ensure the privacy of the specified network connections.

The system in Figure 1 can be described by applying a suitable network topology abstraction to the processes A, B, and C.

$$(\| \; v_1, v_2, v_3 \bullet v_1 \xleftrightarrow{ab} v_2;\; v_2 \xleftrightarrow{bc} v_3;\; v_3 \xleftrightarrow{ac} v_1)(A, B, C)$$

The processes A, B, C are the *actual network parameters*. When the actual network parameters are all process names, the syntactic conventions are relaxed to allow the formal network topology to act in the guise of a process operator. For example, the network topology of Figure 1 may be described by the lax usage

$$\| (A \xleftrightarrow{ab} B;\ B \xleftrightarrow{bc} C;\ C \xleftrightarrow{ca} A).$$

Such usage is considered acceptable since the names representing the actual parameter can serve the dual purpose of also identifying the formal parameters. Other forms of lax usage allow network connections with common nodes to be run together, for example

$$\| (A \xleftrightarrow{ab} B \xleftrightarrow{bc} C \xleftrightarrow{ca} A),$$

and multiple channels above the arrow, for example

$$\| (A \xleftrightarrow{ab_1, ab_2} B).$$

We believe this TCOZ network topology convention to be a good candidate to be included in the basic CSP notation itself.

3 The lift case study

The multi-lift system is a standard example used to demonstrate the expressive power of various specification techniques in modeling concurrent reactive systems[2]. People are familiar with the user requirement of a lift system, so they can concentrate on the modeling notations. However, the lift case study is not a trivial example because of the complexity caused by inherent concurrent interaction in the system [33]. We chose the specification of the lift system as the TCOZ case study also because both CSP and Object-Z have been applied to the lift system allowing a comparison to be drawn. The CSP 'lift' model [30] describes the sequences of events for the lift system well, however, it struggles to capture the data aspects of the lift system. Furthermore, the CSP model has a flat and in places awkward structure and the communications interfaces between the lift system components (i.e. floor-buttons and lifts) are not clearly documented. The Object-Z 'lift' model [9] demonstrates the power of modeling the state change of the lift system in a structured way. However, it is complicated by the need to represent process state as data and it uses a complex, centralised control-model because of the Object-Z's single thread semantics. Neither the CSP or Object-Z model addresses the real-time issues for the lift system.

Our goals for the TCOZ specification of the lift system include:

- a model that captures both the data structures and the behaviour of the lift system;
- a true multi-threaded specification that captures the concurrent, reactive, and real-time aspects of the lift system; and
- a component based incremental style specification that is extendible and reusable.

This case study aims to demonstrate the TCOZ approach for modeling timed reactive systems. Other detailed (lift specific) issues, such as the efficiency of lift scheduling, are not considered in this paper.

[2] Many formal notations have been applied to the lift system. For example, the UNITY model [3, 5], Raddle [13], and Constraint Nets [34, 35].

3.1 System overview

A lift system for a building consists of multiple lifts each providing transport between the various floors of the building as dictated by the pressing of a range of service-request buttons. Inside each lift there is a panel of buttons, one for requesting travel to each of the building's floors. The panel buttons of any lift must be in one-to-one correspondence with the floor numbers. In general there are two service-request buttons on each floor, for upward and downward travel respectively, though on the first floor and the top floor there is only one button. Any service-request button can be pushed at any time. Once pushed the button is said to be *on* and it remains on until the requested service is provided. Pressing an internal button requests the lift to visit the corresponding floor. Pressing an external button requests a lift to visit the floor with the desired direction of travel. The lift controller has a queue which stores all current (external) floor requests. When a request arrives from a floor, the system will put the request at the end of the external request queue. When a lift becomes available, the controller will assign the first request of the queue to the lift for service. When visiting a floor, a lift door operates in the order of open-door, wait, then close-door. This normal process can be interrupted by a customer crossing the door as detected by some sensors.

Furthermore, the following timing properties must be captured in the model:

- lift travel time between two consequent floors is a constant, however there is a constant time delay for acceleration and braking;
- without interrupts, the lift door should be kept at the 'open' state for a fixed time period before closing.

3.2 Specification structure

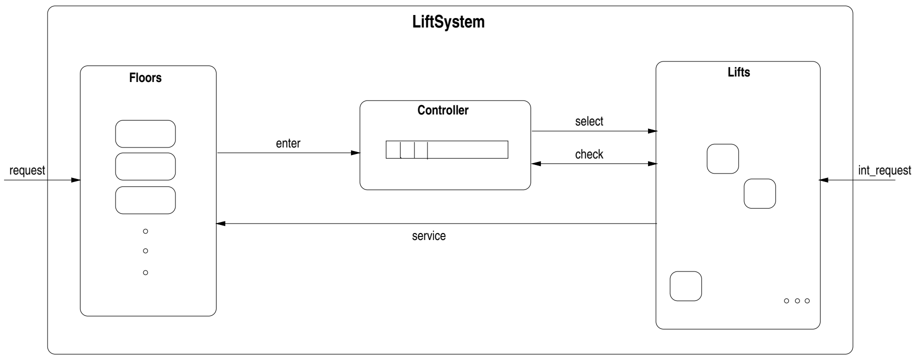

Figure 2: Lift system communication diagram

The lift specification is developed in a bottom-up manner, beginning with models of basic (passive) component objects, such as 'buttons', which are then used to develop more complex (active) component objects such as 'floors (requests)', 'lifts' and a 'controller'. Then the lift system is modeled as an active composite object which composes the component objects with their interactions (through channels).

Figure 2 illustrates the communication interfaces between lift system components. There are three major components, the service-request panels on each floor, the lifts themselves, and the central controller which mediates service requests from the floors. External requests received by a floor on the *request* channel cause the floor's corresponding service button to be lit and the request is communicated to the controller on the *enter* channel. The button remains lit until a confirmation is received on the *service* channel. Requests received by the controller on the *enter* channel are enqueued and sent to idle lifts on the *select* channel on a first-in-first-out basis. Whenever a lift receives an internal request on the *int_request* channel, the corresponding button is lit and the requested entered into its itinerary. Whilst the lift has local requests pending it services them in strict order according to its current movement direction, reversing direction at the extreme floors. The behaviour of the active lifts is monitored on the *check* channel and if a request can be serviced en route it is dispatched to the lift in question and dequeued. Once the lift becomes idle, it may accept an external request on the *select* channel, move to the requested floor, and send a confirmation to the floor panel on the *service* channel.

3.3 Buttons

A basic component of the lift system is the button panels 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.

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

Buttons are modeled using a simple passive class which records their current state and provides operations for turning them on and off.

Button

$state : ButtonStatus$	INIT $state = Off$
TurnOn $\Delta(state)$ $state' = On$	*TurnOff* $\Delta(state)$ $state' = Off$

3.4 The building

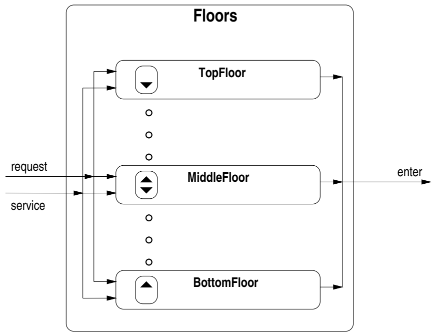

Figure 3: External service request panels.

Our model of the building concentrates on the behaviour of the service-request panels on each floor, as depicted in Figure 3, other floor related issues are elided.

3.4.1 Floor panels

Floors may be divided into two classes, those from which it is possible to travel upward and those from which is possible to travel downward. The *TopFloor* is a floor from which only downward travel is possible,

$$
\begin{array}{l}
\textit{TopFloor} \\
\hline
\begin{array}{l|l}
\begin{array}{l}
\textit{downbutton} : \textit{Button}_{©} \\
\textit{num} : \mathbb{N} \\
\textit{request}, \textit{enter}, \textit{service} : \textbf{chan}
\end{array}
&
\begin{array}{l}
\text{INIT} \\
\hline
\textit{downbutton}.\text{INIT}
\end{array}
\end{array} \\
\textit{PressDown} \mathrel{\widehat=} \textit{request}?(\textit{num}, \textit{Down}) \to \\
\quad [\textit{downbutton}.\textit{state} = \textit{Off}] \bullet \textit{downbutton}.\textit{TurnOn};\ \textit{enter}!(\textit{num}, \textit{Down}) \to \text{SKIP} \\
\quad [\textit{downbutton}.\textit{state} = \textit{On}] \bullet \text{SKIP} \\
\textit{DownOff} \mathrel{\widehat=} \textit{service}?(\textit{num}, \textit{Down}) \to \textit{downbutton}.\textit{TurnOff} \\
\text{MAIN} \mathrel{\widehat=} \mu\, T \bullet (\textit{PressDown} \;\square\; \textit{DownOff});\ T \\
\hline
\end{array}
$$

and the *BottomFloor* is a floor from which only upward travel is possible.

$$
\begin{array}{l}
\textit{BottomFloor} \\
\hline
\begin{array}{l|l}
\begin{array}{l}
\textit{upbutton} : \textit{Button}_{©} \\
\textit{num} : \mathbb{N} \\
\textit{request}, \textit{enter}, \textit{service} : \textbf{chan}
\end{array}
&
\begin{array}{l}
\text{INIT} \\
\hline
\textit{upbutton}.\text{INIT}
\end{array}
\end{array} \\
\textit{PressUp} \mathrel{\widehat=} \textit{request}?(\textit{num}, \textit{Up}) \to \\
\quad [\textit{upbutton}.\textit{state} = \textit{Off}] \bullet \textit{upbutton}.\textit{TurnOn});\ \textit{enter}!(\textit{num}, \textit{Up}) \to \text{SKIP} \\
\quad [\textit{upbutton}.\textit{state} = \textit{On}] \bullet \text{SKIP} \\
\textit{UpOff} \mathrel{\widehat=} \textit{service}?(\textit{num}, \textit{Up}) \to \textit{upbutton}.\textit{TurnOff} \\
\text{MAIN} \mathrel{\widehat=} \mu\, B \bullet (\textit{PressUp} \;\square\; \textit{UpOff});\ B \\
\hline
\end{array}
$$

A *MiddleFloor* is a floor from which both upward and downward travel is possible. Object-Z's class inheritance features are used to allow both upward and downward travel for the *MiddleFloor* class.

$$
\begin{array}{l}
\textit{MiddleFloor} \\
\hline
\textit{TopFloor}, \textit{BottomFloor} \\
\hline
\quad \textit{downbutton} \neq \textit{upbutton} \\
\hline
\text{MAIN} \mathrel{\widehat=} \mu\, M \bullet (\textit{PressDown} \;\square\; \textit{DownOff} \;\square\; \textit{PressUp} \;\square\; \textit{UpOff});\ M \\
\hline
\end{array}
$$

Inheritance provides a subclassing mechanism for specification reuse (not a true subtyping mechanism). When a new active class is derived from an existing active class, the objects of the new class often have different behaviour to the objects of the existing class. As a consequence, the MAIN process must always be redefined explicitly.

The subscript '$\__{©}$' (object containment[3] [7]) indicates that the button objects are contained in their corresponding floor object.

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 [6].

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

[3] 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 [7].

3.4.2 The building

The building is modeled as an aggregate of active floor objects. Individual floors do not communicate with each other, but rather with the central controller and with the lifts. Thus the MAIN processes of the individual floor objects are combined using asynchronous composition, $\_ ||| \_$.

Building

$floors : \mathrm{seq}\, Floor_{©}$

$\forall i : \mathrm{dom}\, floors \bullet floors(i).num = i$
$floors(1) \in BottomFloor \wedge floors(\#floors) \in TopFloor$
$\forall i : (2 \mathinner{..} (\#floors - 1)) \bullet floors(i) \in MiddleFloor$

$\mathrm{MAIN} \mathrel{\widehat=} |||\, fl : \mathrm{dom}\, floors \bullet floors(fl)$

3.5 Lifts

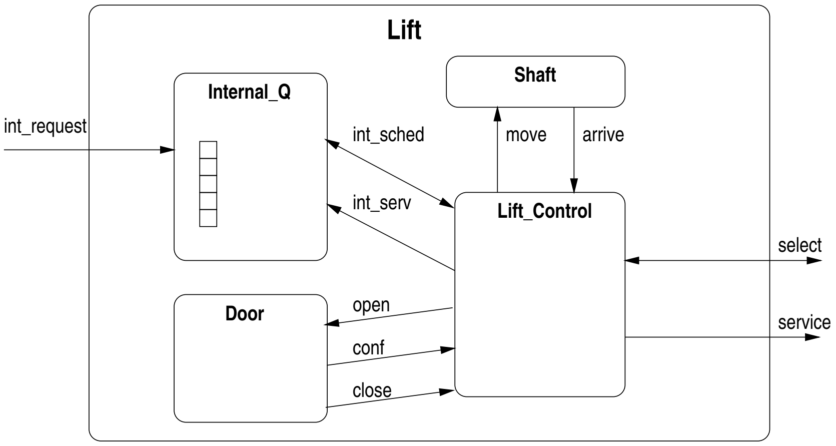

Figure 4: Internal lift communication diagram

A lift consists of four parts as depicted in Figure 4, a door for allowing access to and from the lift, a shaft for transporting the lift, an internal queue for determining the lift itinerary, and a controller for coordinating the behaviour of the other components. This division structures the specification in such a way as to limit the complexity of the individual components and to highlight the potential for concurrency.

3.5.1 Lift door control

The lift door controller is treated as a separate class so as to ensure a clear description of its timing and safety properties. Under this limited aim, the class may be described entirely within the Timed CSP idiom.

The controller interfaces on a channel *servo* with a servomechanism that activates the door to open or close and on a channel *sensor* to determine when the door is open, closed, or blocked from closing. The messages that may be set on these channels are then

$$DoorMess ::= ToOpen \mid Opened \mid ToClose \mid Closed \mid Interrupt.$$

The timing property of the door is that once it is open, it must remain open for time period

$$t_o : \mathbb{T} \qquad \text{[open time]}$$

before closing. The safety property is that if the closing of the door is blocked (as indicated by receipt of an *Interrupt* message) the door must be reopened.

The door cycle is initiated by receipt of an *open* signal from the lift controller and completed by sending a *close* signal. As soon as the door is open, a *conf* signal must be sent to the lift controller so as to indicate fulfillment of a service request.

Door

$open, conf, close, servo, sensor : \mathbf{chan}$

$OpenDoor \mathrel{\widehat=} servo!(self, ToOpen) \rightarrow sensor?(self, Opened) \rightarrow \textsc{Skip}$
$CloseDoor \mathrel{\widehat=} servo!(self, ToClose) \rightarrow sensor?(self, Closed) \rightarrow \textsc{Skip}$
$CycleDoor \mathrel{\widehat=} OpenDoor;\ conf \rightarrow$
$\quad (\mu\, CD \bullet \textsc{Wait}\ t_o;\ CloseDoor\ \nabla\{sensor?(self, Interrupt)\}\ OpenDoor;\ CD)$
$\textsc{Main} \mathrel{\widehat=} \mu\, D \bullet open \rightarrow CycleDoor;\ close \rightarrow D$

In the *CycleDoor* process we make use of the CSP interrupt primitive which allows an exception handling behaviour to be triggered by the occurrence of an unusual event. The normal door cycle follows the order of open-door, wait t_o, then close-door. This normal cycle can be interrupted and re-started by the event which detects a message *Interrupt* from the channel *sensor*. This message acts as an interrupt on the normal cycle and control is returned to the start of the normal cycle.

3.5.2 Moving the lift

The essentially analog nature of the movement of the lift presents something of a modeling problem. The technique used throughout this specification has been to abstract real-world interactions as CSP events (eg the *request* and *int_request*) channels, but the movement betweens floors is by its nature a time-consuming process. We thus adopt the common technique of delimiting the period of movement by start and finish events. The start event of the movement process is a communication to the lift-shaft apparatus of the number of floors to be moved. The finish event is a communication from the lift-shaft apparatus that the lift has arrived at the destination floor. The timing properties of lift movement are described by two time constants.

$$\bar{t} : \mathbb{T} \qquad \text{[time to move one floor Up or Down]}$$
$$delay : \mathbb{T} \qquad \text{[acceleration and braking delay]}$$

The normal time to pass from one floor to another is $\bar{t}$ for each floor travelled plus a delay of *delay* caused by initial acceleration and final braking of the lift.

Shaft

$move, arrive : \mathbf{chan}$

$\textsc{Main} \mathrel{\widehat=} \mu\, S \bullet [n \neq 0] \bullet move?n \rightarrow \textsc{Wait}\ |n| * \bar{t} + delay;\ arrive \rightarrow S$

Such event based models are highly abstract and perhaps are less satisfying when applied to the complex process of moving the lift, than when applied to the more event-like process of pressing a button. However, the channel based interfaces of TCOZ processes mean that such models must be used. In their favour it must be pointed out that from the point of view of the lift controller the matters of essential interest are precisely when the movement commences and when it finishes.

3.5.3 Lift itinerary

The itinerary of the lift is determined by the requests made by passengers using the internal floor-request panel and those dispatched from the central control. For the purposes of limiting design complexity a separate class is defined to determine the lift's internal service itinerary.

This internal queue makes use of a *panel* of buttons to communicate with passengers and to maintain a record of the floor-requests pending. A dependent state variable[4] *irs* records the set of destinations that have been selected by the passengers at any given time. This set may in turn be split into those destinations above a given floor, *ups*, and those below, *dns*. Floor services are requested on the *int_request* channel and confirmed on the *int_serv* channel. Scheduling services are requested by passing the current floor *fl*? and movement direction *md*?

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

on the *int_sched* channel.

The operations of the controller are turning the panel buttons on and off, in response to service requests and confirmations, and determining the next destination for the lift itinerary. The next destination is the first requested destination in the current direction of movement, reversing at either extremity of movement. A scheduling request is only serviced if there are pending floor-service requests.

Internal_Q

$$
\begin{array}{l}
panel : \mathrm{seq}\, Button_{©} \\
int\_request, int\_sched, int\_serv : \mathbf{chan} \\
\Delta \\
irs : \mathbb{F}\,\mathbb{N} \\
ups, dns : \mathbb{N} \nrightarrow \mathbb{F}\,\mathbb{N} \\
\hline
irs = \{f : \mathrm{dom}\, panel \mid panel(f).state = On\} \\
ups = \lambda\, fl : \mathrm{dom}\, panel \bullet \{n : irs \mid n \geq fl\} \\
dns = \lambda\, fl : \mathrm{dom}\, panel \bullet \{n : irs \mid n \leq fl\}
\end{array}
$$

NextUp

$$
\begin{array}{l}
fl?, dest! : \mathrm{dom}\, panel \\
\hline
(ups(fl?) \neq \varnothing \wedge dest! = min\ ups(fl?)) \\
\vee \\
(ups(fl?) = \varnothing \wedge dns(fl?) \neq \varnothing \wedge \\
dest! = max\ dns(fl?))
\end{array}
$$

NextDown

$$
\begin{array}{l}
fl?, dest! : \mathrm{dom}\, panel \\
\hline
(dns(fl?) \neq \varnothing \wedge dest! = max\ dns(fl?)) \\
\vee \\
(dns(fl?) = \varnothing \wedge ups(fl?) \neq \varnothing \wedge \\
dest! = min\ ups(fl?))
\end{array}
$$

[4] Syntactically, the declarations of dependent/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. For a discussion see [8].

$Next \mathrel{\widehat=}$
$\quad [md? : MoveDirection \mid md? = Up] \wedge NextUp \vee$
$\quad [md? : MoveDirection \mid md? = Down] \wedge NextDown$
$\textsc{Main} \mathrel{\widehat=} \mu\, IQ \bullet$
$\quad [fl : \operatorname{dom} panel] \bullet int\_request?(self, fl) \rightarrow panel(fl).TurnOn;\ IQ \ \Box$
$\quad [fl : \operatorname{dom} panel] \bullet int\_serv?fl \rightarrow panel(fl).TurnOff;\ IQ \ \Box$
$\quad [fl : \operatorname{dom} panel;\ md? : MoveDirection \mid irs \neq \varnothing] \bullet$
$\qquad int\_sched?(fl, md) \rightarrow Next;\ int\_sched!dest \rightarrow IQ$

3.5.4 Lift controller

The lift controller keeps record of the current floor and movement direction and provides the interface between the lift environment and the other lift components. The lift controller exhibits of three modes of behaviour.

The lift begins at rest awaiting either a passenger destination request or a dispatch from the central controller. Any passengers inside the lift are given a period of time

$t_p : \mathbb{T}$ [wait for passenger input]

to make an internal request before the lift accepts any external requests.

The controller determines that an internal request is pending through the willingness of the internal queue to perform a scheduling transaction. Once the queue has indicated the next destination the central controller is checked to see if there is an external request from an intermediate floor. If so the liftservices the external request first. If not it services the internal request: the new direction is calculated and set; the lift is moved to the new floor and the door is opened; the internal queue and the central controller are notified; and then once the door is closed control is returned to the rest mode.

If, after waiting t_p, an external request becomes available before any internal request: the new move direction is set; the lift is moved to the new floor and the door is opened; the floor service-request panel is notified; and then, once the door is closed, control is returned to the rest mode.

LiftControl

$fl : \mathbb{N}$
$md : MoveDirection$
$move, arrive : \mathbf{chan}$ [shaft]
$open, conf, close : \mathbf{chan}$ [door]
$int\_sched, int\_serv : \mathbf{chan}$ [internal Q]
$select, check, service : \mathbf{chan}$ [external channels]

SetFloor

$\Delta(fl)$
$dest? : \mathbb{N}$

$fl' = dest?$

SetDir

$\Delta(md)$
$dir? : MoveDirection$

$md' = dir?$

CalDir

$dest? : \mathbb{N}$
$dir! : MoveDiection$

$dest? = fl \wedge dir! = md \vee$
$dest? > fl \wedge dir! = Up \vee$
$dest? < fl \wedge dir! = Down$

$Move \mathrel{\widehat=}$
$\quad [dest : \mathbb{N} \mid dest \neq fl] \bullet move!(dest - fl) \rightarrow arrive \rightarrow open \rightarrow conf \rightarrow SetFloor \ \Box$
$\quad [dest : \mathbb{N} \mid dest = fl] \bullet open \rightarrow conf \rightarrow \textsc{Skip}$

$$
\begin{array}{|l}
\mathit{Internal} \mathrel{\widehat=} \mathit{CalDir};\ \mathit{SetDir};\ \mathit{Move};\ \mathit{int\_serv}!\mathit{fl} \to \mathit{close} \to \textsc{Skip} \\
\mathit{Get\_Internal} \mathrel{\widehat=} \\
\quad \mathit{int\_sched}!(\mathit{fl}, \mathit{md}) \to [\mathit{dest} : \mathbb{N}] \bullet \mathit{int\_sched}?\mathit{dest} \to \mathit{check}!(\mathit{fl}, \mathit{dest}, \mathit{md}) \to \\
\qquad \mathit{check} \to \mathit{Internal} \;\Box \\
\qquad [\mathit{dest} : \mathbb{N}] \bullet \mathit{check}?\mathit{dest} \to \mathit{External} \\
\mathit{External} \mathrel{\widehat=} \mathit{Move};\ \mathit{service}!(\mathit{fl}, \mathit{md}) \to \mathit{close} \to \textsc{Skip} \\
\mathit{Get\_External} \mathrel{\widehat=} [\mathit{dest} : \mathbb{N};\ \mathit{dir} : \mathit{MoveDirection}] \bullet \mathit{select}?(\mathit{dest}, \mathit{dir}) \to \mathit{SetDir};\ \mathit{External} \\
\textsc{Main} \mathrel{\widehat=} \mu\, \mathit{LC} \bullet \mathit{Get\_Internal} \;\Box\; (\textsc{Wait}\ t_p;\ \mathit{Get\_External});\ \mathit{LC} \\
\hline
\end{array}
$$

3.5.5 The bank of lifts

Each lift consists of a door, a shaft, and a controller.

Lift

$$
\begin{array}{|l}
\hline
\mathit{iq} : \mathit{Internal\_Q}_{\copyright} \\
\mathit{lc} : \mathit{LiftControl}_{\copyright} \\
s : \mathit{Shaft}_{\copyright} \\
d : \mathit{Door}_{\copyright} \\
\hline
\textsc{Main} \mathrel{\widehat=} \|(\mathit{lc} \xleftrightarrow{\mathit{move}, \mathit{arrive}} s;\ \mathit{lc} \xleftrightarrow{\mathit{open}, \mathit{close}, \mathit{conf}} d;\ \mathit{lc} \xleftrightarrow{\mathit{int\_sched}, \mathit{int\_serv}} \mathit{iq}) \\
\hline
\end{array}
$$

The collection of all the lifts in the system is modeled as an aggregate of the individual lifts acting autonomously.

Lifts

$$
\begin{array}{|l}
\hline
\mathit{lifts} : \mathbb{P}\, \mathit{Lift}_{\copyright} \\
\hline
\textsc{Main} \mathrel{\widehat=} |||\ l : \mathit{lifts} \\
\hline
\end{array}
$$

3.6 The central controller

The responsibility of the central controller is to dispatch floor requests to idle lifts. It consists of a request queue with channels that connect the floors and the lifts. The network topology is described graphically in Figure 2.

Controller

$$
\begin{array}{|l}
\hline
\mathit{reqQ} : \mathrm{seq}(\mathbb{N} \times \mathit{MoveDirection}) \\
\mathit{enter}, \mathit{select}, \mathit{check}, \mathit{service} : \mathbf{chan} \\
\Delta \\
\mathit{req} : \mathbb{N} \leftrightarrow \mathit{MoveDirection} \\
\mathit{ups}, \mathit{dns} : \mathbb{N} \times \mathbb{N} \to \mathbb{P}\,\mathbb{N} \\
\hline
\mathit{req} = \mathrm{ran}\ \mathit{reqQ} \\
\forall f_1, f_2 : \mathbb{N} \bullet \\
\quad \mathit{ups}(f_1, f_2) = \{ f : \mathbb{N} \mid f\ \underline{\mathit{req}}\ \mathit{Up} \land f_1 \le f \le f2 \} \\
\quad \mathit{dns}(f_1, f_2) = \{ f : \mathbb{N} \mid f\ \underline{\mathit{req}}\ \mathit{Down} \land f_1 \le f \le f2 \} \\
\hline
\end{array}
$$

INIT

$$
\begin{array}{|l}
\hline
\mathit{reqQ} = \langle\ \rangle \\
\hline
\end{array}
$$

Join

$$\Delta(reqQ)$$
$$req? : (\mathbb{N} \times MoveDirection)\ \mathbf{on}\ enter$$
$$\overline{\qquad}$$
$$reqQ' = reqQ \frown \langle req? \rangle$$

Remove

$$\Delta(reqQ)$$
$$(dest?, md?) : \mathbb{N} \times MoveDirection$$
$$\overline{\qquad}$$
$$reqQ' = \mathrm{squash}(reqQ \rhd \{(dest?, md?)\})$$

Find

$$fl?, d? : \mathbb{N}$$
$$md? : MoveDirection$$
$$dest! : \mathbb{N}$$
$$suc! : \mathbb{B}$$
$$\overline{\qquad}$$
$$md? = Up \land suc! = (ups(fl?, d?) \neq \varnothing) \land$$
$$suc! \Rightarrow dest! = min\ ups(fl?, d?)$$
$$\lor$$
$$md? = Down \land suc! = (dns(fl?, d?) \neq \varnothing) \land$$
$$suc! \Rightarrow dest! = min\ dns(fl?, d?)$$

$$Dispatch \mathrel{\widehat=}$$
$$\quad [(dest, md) : \mathbb{N} \times MoveDirection \mid reqQ \neq \langle\ \rangle \land (dest, md) = \mathrm{head}\ reqQ] \bullet$$
$$\qquad select!(dest, md) \to Remove$$

$$CheckServ \mathrel{\widehat=}$$
$$\quad [fl, d : \mathbb{N};\ md : MoveDirection] \bullet check?(fl, d, md) \to Find;$$
$$\qquad [suc] \bullet check!dest \to Remove\ \Box$$
$$\qquad [\neg\ suc] \bullet check \to \textsc{Skip}$$

$$\textsc{Main} \mathrel{\widehat=} \mu\ C \bullet (Join\ \Box\ Dispatch\ \Box\ CheckServ);\ C$$

The controller receives requests from the floors and enters them into the *reqQ* queue (*Join* operation). In the ordinary case these requests are dispatched in first-in-first-out manner (*Dispatch* operation) as idle lifts become available, but if in the course of servicing internal requests a lift can visit a floor whilst moving in the floor required direction, the request is removed from the queue (*CheckServ* operation).

3.7 The lift system

The lift system consists of the floors of the building, the bank of lifts and the central controller. The number of floor-service buttons in each lift must be exactly the number of floors in the building.

LiftSystem

$$bd : Building$$
$$ls : Lifts$$
$$contr : Controller$$
$$\overline{\qquad}$$
$$\forall\ l : ls.lifts \bullet \#l.iq.panel = \#bd.floors$$

$$\textsc{Main} \mathrel{\widehat=} \| (bd \xleftrightarrow{enter} contr \xleftrightarrow{select, check} ls \xleftrightarrow{service} bd)$$

The lift system behaviour $\textsc{Main}$ describes the communication channels between the independent concurrently executing system components: the lifts, the floors, and the controller. The floors communicate service requests to the controller through the *enter* channel, the controller dispatches these requests to the lifts through the *select* and *check* channels, and the lifts indicate successful servicing of requests to the floors through the *service* channel.

3.8 Discussion

The application of TCOZ to the multi-lift system has been convincingly successful, despite the relatively modest real-time aspects of the specification. The powerful combination of object-oriented structuring, data modeling, and process modeling features available in TCOZ has allowed the clean presentation of an ambitiously detailed (when compared to versions described in other languages) treatment of the multi-lift system. This strongly supports our claims that TCOZ represents a highly scalable and reusable method for describing real-time and concurrent systems. As an example, consider modifying the specification so as to allow 'opportunistic' servicing of requests entered after a lift starts moving. The modular nature of the specification immediately draws attention to the *Shaft* class which controls the movement of the lift between floors. The interface of this class needs to be expanded to include events to indicate the lift is approaching the next floor and to stop the lift at the next floor. Armed with these additional controls the *LiftControl* class can easily be modified to react opportunistically to new service requests.

One surprise in the development of the TCOZ lift was the degree to which the process model idiom came to dominate. The starting point of the development had been a data oriented standard Object-Z specification [9]. Rather than being a simple matter of adding real-time and concurrent features to the existing specification it soon became clear that much of its 'data' was in fact being used to represent 'process' behaviour which could be more elegantly represented using the CSP process modeling features of TCOZ. The final approach adopted was to model the system primarily as a network of communicating processes and to make use of Object-Z's data modeling features to simplify and structure the specification by abstracting away from algorithmic specifics and reusing common data components such as buttons.

A perceived weakness of the TCOZ approach was identified in its handling of the interface between TCOZ processes and the real world. Although the abstraction of button presses as communications on external channels is reasonably acceptable, the inability to describe 'continuously' changing aspects of the system such as 'lift position' is particularly disturbing. Whilst modeling the moving process by start and finish events provides an adequate interface to the TCOZ specification it goes no way at all toward ensuring that lift shaft satisfies our informal intuitions as to its behaviour. The specification is thus strictly speaking not of a system for moving people between floors of a building, but rather simply a description of a method of controlling such a system in a satisfactory manner.

One advantage of choosing the lift system to exercise the TCOZ language is the availability of existing lift specifications in both CSP [30] and Object-Z [9]. Apart from the ability of TCOZ to describe timing aspects not addressed by either of these specifications, such as the correct behaviour of the door opening cycle, TCOZ represents an improvement in expressibility, modularity, and reusability over both existing specifications.

The CSP 'lift' model [30] is similar in spirit to the TCOZ lift presented here, except that it does not consider the door cycle nor the movement of the lift. However, in contrast to the highly modular approach of the TCOZ specification, the CSP version is forced to adopt a quite flat structure because the only structuring facility available is the process definition. The channel interface declarations and the network topology operator provide valuable information to the reader of the TCOZ specification regarding the source and destination of communications, which is not available to the reader of the CSP specification. Although some attempt is made to structure the CSP specification document through the use of section headings to distinguish system components, the essentially global nature of all process definitions and communication channels makes it difficult to comprehend individual process definitions without extensive reference to the rest of the specification. The standard CSP synchronisation operator ($\_ \parallel \_$) is a particular point of weakness as it offers no visual feedback as to the interface between network components. The use of the network topology operator (and associated diagrams) is a particular strength of the TCOZ specification.

The weaknesses in CSP's treatment of data and algorithms not only adds syntactic clutter to the CSP lift, but also appears to influence the structure of the specification. In particular, the difficulty in abstracting various calculations away from the specific direction of travel results in a 'split' specification, with many features being repeated for both the 'up' and 'down' directions of travel. For example, the specification of a single lift [30, p 26] includes subprocesses $Lift(f, up)$ and $Lift(f, down)$ which differ primarily in the method of calculating the next floor on the itinerary. In contrast the TCOZ version abstracts this calculation into the *Next* operation of the *Internal_Q* class allowing the description of the lift's gross behaviour to be independent of the direction of travel, even despite using a more complex method of determining the itinerary. Although a specification of this form is possible in CSP, the lack of strong, modular data modeling facilities acts as a strong disincentive to this form of abstraction.

The Object-Z 'lift' model [9] provided a structured and reusable model and was able to describe the sequence of state changes of the overall lift system. However, Object-Z's process semantics forces the specification to include extra data for describing process state and all system components to be viewed passively (except the lift system class) which leads to a complex, centralised control-model. The lift components, such as the shaft and the door, were abstractively modeled as internal state components (rather than component objects) of the lift. The system class *LiftSystem* [9, pp 146,147] became very complex (and lengthy) because the global ordering of synchronisations between lifts, floor requests and request queue must be explicitly determined. In the TCOZ model only the local ordering of these events need be specified, the global ordering is implicit in the CSP event synchronisation model. In addition, the TCOZ model gives the freedom of viewing the lift system components as active entities allowing a more natural, modular, and reusable description of the lift system.

4 Related work

The expressiveness of the integrated notation TCOZ (Timed CSP + Object-Z) has been demonstrated through the lift case study in Section 3. The basis for the successful blending of the two notations is the duality between state transition semantics and event semantics. This has long been recognised [16, 21, 27] and has undoubtedly helped shape the development of Object-Z's behavioural semantics. Perhaps the most mature formalism based on this duality is Butler's blending of CSP with Back's Action Systems [1]. An important lesson from this work is the need to distinguish strongly between the notions of guard and precondition in the state-transition view. The failure to do so in Object-Z has made it impossible to reconcile algorithmic refinement with the default behavioural semantics, greatly reducing the value of refinement in Object-Z. The adoption of a distinct notion of state guards in TCOZ makes possible a full blending of Z-style algorithm and CSP-style process refinement.

The notion of blending the untimed CSP and Object-Z has been proposed independently by Fischer [11] and Smith [31]. Both take the approach of identifying the notion of channel with that of operation and operation invocations with atomic communications of both inputs and outputs. The latter prevents the modeling of timing and concurrency at the operation level and complicates the CSP semantics through the mixing of elements of external (inputs) and internal (outputs) choice in a single event. The former is undesirable from a theoretical standpoint because it confuses communications interfaces which are essentially process related attributes with algorithmic structures which are essentially data related attributes. An object's communications interface should be determined by high-level considerations of the overall system structure, whilst the operational interface should be determined by consideration of the internal data structures. The purpose of the class envelope is to resolve such tensions locally, not to propagate them up and down the design hierarchy. The practical consequences of the identification of channel and operation is the promotion of both high degrees of coupling between classes and unnatural class structures. Neither formalism makes a complete distinction between preconditions, guards, and operations and consequently refinement issues are complicated in both. Smith adopts a semantics which is unable to model process divergence and as a consequence must identify preconditions with guards, making process and algorithmic refinement incompatible. The semantics adopted by Fischer does allow a distinction between guards and preconditions, but guards are tightly coupled with operations so that the same operation may not be used in differing circumstances as is the *Remove* operation in the *Controller* class on p. 13. Moreover, a convention is introduced whereby when an operation guard is not explicitly defined the precondition is used by default, thus complicating both the understanding of the process behaviour and the refinement of the operation. Issues, such as real-time and the distinction between active and passive objects, are not addressed by either formalism.

More generally, the need for specification notations capable of addressing both data/algorithmic issues and process control issues is now widely recognised. Several notations now exist aimed at bridging this divide. These fall essentially into two classes, those that adopt a process-algebra/event-based style (LOTOS [20], ESTEREL [2], RAISE [14]) and those that adopt a transition system style (UNITY [3], Action Systems [1], TLA [22], ASTRAL [4]). Since it is closest in spirit to TCOZ, we briefly consider the relationship between TCOZ and the real-time extension to LOTOS, E-LOTOS [19].

The LOTOS specification language [20] is very similar in approach to TCOZ, blending CSP-like process primitives with an algebraic-style data-specification language. E-LOTOS [19] is a recently developed real-time extension to LOTOS. The process and real-time primitives of E-LOTOS are influenced by the Timed-CSP, therefore these aspects of E-LOTOS are similar to TCOZ. The primary difference lies in the data modeling and structuring aspects of the two formalisms. The data modeling language of E-LOTUS is equational algebra based, whilst TCOZ is model based. The module construct of E-LOTOS is similar to the class construct of TCOZ in that it can encapsulate states and operations. Modules can be reused via the *imports* mechanism which is similar to the class inheritance. However E-LOTOS modules cannot be instantiated as a type, while TCOZ classes can. Therefore, the notions of object and composition of objects (aggregation) are missing from E-LOTOS. In TCOZ this adds another dimension of potential for reuse of specifications. E-LOTOS's subtyping is a simple record-type extension mechanism which is less powerful than the TCOZ's polymorphic typing (inheritance hierarchy and class union).

5 Conclusions and further work

Timed CSP and Object-Z complement each other not only in their expressive capabilities, but also in their underlying semantics. In addition, the object oriented flavour of Object-Z provides an ideal foundation for promoting modularity and separation of concerns in system design. The combination of the two, TCOZ, treats data and algorithmic concerns in the Object-Z style and treats process control and communication concerns in the CSP style. Timing concerns are distributed according to their status as relating to algorithm or control. The notion of active and passive objects are clearly distinguished in TCOZ model.

This powerful modeling combination, TCOZ, has been successfully applied to a comprehensive case study on specifying a real-time multi-lift system. In comparison to the CSP model [30] and the standard Object-Z model [9] of the lift system, the TCOZ model not only captures the complex system state and behaviour, but also captured the true concurrent real-time interactions between various system components of the lift system. The lift case study also provides feedback to the development of TCOZ. For example, the development of the modeling notation for complex network topologies is motivated by the lift case study. A particular weakness of the language has been identified in its ability interface with 'real-world' aspects of a system. This clearly limits the applicability of the notation to the software aspects of a system. Future work will be directed toward improving its capabilities in this direction so as to integrate it into a more holistic approach to real-time and embedded systems design. One promising approach is to enhance TCOZ with features of the Timed Refinement Calculus [23, 26] which allow convincing descriptions of continuously varying real-world observables.

TCOZ preserves in large part both the syntax and semantics of the individual notations and hence can potentially benefit from the large body of experience developed in the use of and tool support for the individual notations and their parents, such as verification and refinement techniques and tools, model checkers, etc..

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [24]. Additional planned work includes developing refinement rules for the TCOZ specification language based on existing Z and CSP refinement systems. Schneider has described a system for capturing and verifying abstract temporal requirements of Timed CSP processes [28], it is hoped that this might also form a valuable addition to the TCOZ notation.

Acknowledgements

We would like to thank John Colton, Ian Hayes and Keith Gallagher for many useful comments. This work is supported in part by the DSTO/CSIRO Fellowship programme.

References

[1] R. J. R. Back and J. von Wright. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer Verlag, 1990.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[4] A. Coen-Porisini, C. Ghezzi, and R. Kemmerer. Specification of Realtime Systems Using ASTRAL. Technical Report 96-30, Computer Science Department, University of California, Santa Barbara, US, January 1997.

[5] 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.

[6] 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.

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

[8] 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.

[9] J.S. Dong, L. Zucconi, and R. Duke. Specifying parallel and distributed systems in Object-Z. In G. Agha and S. Russo, editors, *The 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 140–149, Boston, Massachusetts, 1997. IEEE Computer Society Press.

[10] 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.

[11] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowmann and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

[12] C. Fisher and G. Smith. Combining csp and object-z: Finite or infinite trace semantics? In *IFIP Joint International Conference on Formal Description Techniques and Protocal Specification, Testing and Verification*, pages 503–518. Chapman & Hall, November 1997.

[13] I. R. Forman. Design by decomposition of multiparty interactions in Raddle87. In *The 5th IEEE International Workshop on Software Specification and Design (IWSSD'89)*, pages 2–10. IEEE Computer Society Press, 1989.

[14] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. Bendix Nielson, S. Prehn, and K. R. Wagner. *The Raise Specification Language.* Prentice Hall, New York, 1992.

[15] I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7(3), 1995.

[16] Jifeng He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[17] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[18] International Organization for Standardization, Geneva. *Units of measurement: handbook on international standards for units of measurement*, 1979.

[19] ISO. SC21/WG7 Working Draft on Enhancements to LOTOS, ISO Working Group 7. December 1997.

[20] ISO 8807. *LOTOS – A formal description technique based on the temporal ordering of observational behaviour*, 1989.

[21] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[22] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.

[23] B. P. Mahony. Networks of predicate transformers. Technical Report 95-05, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, February 1995.

[24] B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: The semantics of TCOZ. Technical Report 97-24, Mathematical and Information Sciences, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia, 1997.

[25] B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*. IEEE Computer Society Press, April 1998. (accepted).

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

[27] C. C. Morgan. Of wp and CSP. In W. H. J. et al Feijen, editor, *Beauty is our Business: a Birthday Salute to Edsger W. Dijkstra*, pages 319–326. Springer-Verlag, 1989.

[28] S. Schneider. *Correctness and Communication in Real-Time Systems.* PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1990. Available as Technical Monograph PRG-84.

[29] S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[30] 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.

[31] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In *Proceedings of FME'97: Industrial Benefit of Formal Methods*, Graz, Austria, September 1997. Springer-Verlag.

[32] G. Smith and J. Derick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 293–302, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

[33] 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.

[34] 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.

[35] 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.

A TCOZ glossary

Notation	Explanation
c : **chan**	declare c to be a channel
$\perp$	divergent process
$\textsc{Stop}$	deadlocked process
$\textsc{Skip}$	terminate immediately
$\textsc{Wait}\ t$	delay termination by t
$a \to P$	communicate a then do P
$a@t \to P$	communicate a at time t then do P
$[t : \mathbb{T}] \bullet a@t \to P$	record time of a event in variable t
$c.a$	communicate a on channel c
$c?a$	input a on channel c
$c!a$	output a from channel c
$[b] \bullet P$	enable P only if b
$P;\ Q$	perform P until termination, then perform Q
$P \Box Q$	perform the first enabled of P and Q
$[i : I] \bullet P$	perform P with first enabled value of i (indexed external choice)
$P \sqcap Q$	perform either of P and Q
$[i! : I];\ P$	perform P with any value of i (indexed internal choice)
$v := e$	syntactic sugar for $[\Delta v \mid v' = e]$
$P \setminus A$	hide the events A from the environment of P
$P \lvert[A]\rvert Q$	synchronise P and Q on events from A

continued on next page

Notation	Explanation
$(\Vert\, p_1,\ldots,p_n \bullet \ldots;\ p_i \xleftrightarrow{A} p_j;\ \ldots)$	network topology abstraction with parameters $p_1,\ldots,p_n$ and network connections including p_i communicating with p_j on private channels from A
$P \mid\mid\mid Q$	P and Q running without sychronisations
$P \rhd\{t\}\ Q$	if P does not begin by time t, perform Q instead
$P \swarrow\{t\}\ Q$	perform P until time t, then transfer control to Q
$P \nabla\ e \rightarrow Q$	perform P until exception e, then transfer control to Q
$P \bullet \textsc{Deadline}\ t$	behaviours of P which terminate before time t
$P \bullet \textsc{WaitUntil}\ t$	after P idle until time t