Timed Communicating Object Z

Brendan Mahony and Jin Song Dong, *Member*, *IEEE*

Abstract—This paper describes a timed, multithreaded object modeling notation for specifying real-time, concurrent, and reactive systems. The notation Timed Communicating Object Z (TCOZ) builds on Object Z's strengths in modeling complex data and algorithms, and on Timed CSP's strengths in modeling process control and real-time interactions. TCOZ is novel in that it includes timing primitives, properly separates process control and data/algorithm issues and supports the modeling of true multithreaded concurrency. TCOZ is particularly well-suited for specifying complex systems whose components have their own thread of control. The expressiveness of the notation is demonstrated by a case study in specifying a multilift system that operates in real-time.

Index Terms—Software/system specification, formal methods, real-time systems, concurrency, object-oriented modeling, Z, CSP.

1 Introduction

Many formal specification and design notations have tended to concentrate either on data modeling and algorithmic concerns (e.g., Z, VDM, etc.) or else on process control concerns (e.g., CSP, CCS, StateCharts, etc.). Complex systems often have intricate system states and process control structures involving concurrency and real-time interactions. To formalize such systems, it is necessary to have a notation which is able to capture both the data/algorithmic issues and the process behavior issues in a smoothly integrated but also highly structured and modular manner. In consequence, the blending of state-modeling and process languages has become an active area of research [21], [49], [50], [24].

Object Z [19] is an object-oriented extension of the Z formal specification language. Z is a model-oriented specification language with powerful features for describing complex data structures and their operations. Object Z improves the clarity of large Z specifications through enhanced structuring. However, the process semantics of Object Z mean that process execution is single-threaded, operations are atomic (there is no notion of the duration of operations), and process control logic is tightly coupled with class structure. Therefore, it is difficult to use Object Z to model concurrent real-time reactive systems.

Timed CSP [46] is an extension of Hoare's Communicating Sequential Processes (CSP) notation. It builds on CSP's strengths in modeling process control issues, such as concurrency and synchronization, by adding primitives for modeling real-time issues. However, CSP has only the most rudimentary mechanisms for modeling data and algorithmic issues and it is cumbersome to capture the state of a complex system.

This paper describes an integration of Object Z and Timed CSP, called Timed Communicating Object Z (TCOZ), and presents a case study on using TCOZ to specify a real-time multilift system. 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 process aspects of complex systems. The notion of blending Object Z with CSP has been suggested independently by Fischer [21] and Smith [49]. The most obvious novelty of TCOZ is that it is built on Timed CSP and includes primitives for treating timing issues; however, in addressing the issue of time, it has been necessary to make several inovations which impact positively even on the treatment of "untimed systems." TCOZ adopts a finer grain of atomicity than either Fischer or Smith. Operations are considered to represent a sequence of (unspecified) update events, rather than to constitute atomic events in themselves. This opens the possibility of treating operation composition and refinement in TCOZ, including the introduction of multithreaded concurrency at the operation level. TCOZ adopts an explicit mechanism for enabling operations (and indeed, arbitrary processes) which is distinct from the operation definition itself. This increases the potential for reuse of operation specifications and allows the notions of operation and process refinement to be reconciled. TCOZ adopts the CSP channel-based communications paradigm in its full generality and enhances it by the introduction of a novel network topology operator that allows the communications interfaces of complex TCOZ processes to be visualized through simple network-topology graphs. This improves decoupling of class definitions by simplifying the interfaces between objects. For the most part, these topics can be touched on only briefly in this paper and they will be the subjects of future more detailed correspondences.

The TCOZ notation has been briefly described and exercized in introductory papers by these authors [39], [40], [10]. This paper combines elements of these papers, but describes the notation and its use in greater detail. Important issues such as formal syntax and semantics are explained for the first time.

The remainder of the paper is organized as follows: In Section 2, Object Z and Timed CSP notations are briefly

- *B. Mahony is with the Information Technology Division, Defence Science and Technology Organisation, Salisbury, Australia. E-mail: Brendan.Mahony@dsto.defence.gov.au.*
- *J.S. Dong is with the School of Computing, National University of Singapore, Lower Kent Ridge Rd., Singapore. E-mail: dongjs@comp.nus.edu.sg.*

Manuscript received 24 Mar. 1998; revised 12 Oct. 1998; accepted 5 Feb. 1999.
Recommended for acceptance by J. Gannon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106596.

introduced. The advantages and disadvantages of the two notations in modeling timing, concurrency, complex data, and algorithmic aspects are demonstrated using a common example, a timed collection. In Section 4, the blended notation, TCOZ, is introduced and the timed collection example used to show how it uses the strengths of the individual notations to address their respective weaknesses. In Section 4, the case study on specifying a real-time multilift system is presented. In Section 5, a discussion of related work is presented. Finally, a syntax for TCOZ is presented as Appendix A.

2 Object-Z and Timed CSP

The common example of a generic timed collection is used throughout this section and the next section to illustrate the differences between and to demonstrate the advantages and disadvantages of the Object Z, Timed CSP, and TCOZ notations, respectively.

2.1 Generic Timed Collection Example

The generic timed collection denotes a collection of elements of type X with a time stamp. Operations are allowed to add elements to and delete elements from the collection. When deleting an element from the collection, the oldest element should be removed and output to the environment. The collection has the following timing properties: First, that it takes a small but nonzero time (t_a and t_d, respectively[1]) to update the internal state during an *add* or *delete* operation. Second, each element of the collection becomes *stale* if it is not passed on within t_o time units of being added to the collection. Stale elements should never be passed on, but are instead purged from the collection upon becoming stale. The *purge* operation has a duration of t_p.

2.2 A Model of Time

In this paper, all timing information is represented as real valued measurements in *seconds*, the SI standard unit of time [30]. 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 [27] are adopted. Under this convention, time quantities are represented by the type

$$\mathbb{T} == \mathbb{R}\ \mathtt{s},$$

which represents real-valued time measured in seconds. Time literals consist of a real number literal annotated with a symbol representing a unit of time. For example, 3 μs is a literal representing a period of three microseconds. All the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

The timing constants associated with the timed collection example are introduced via axiomatic definitions.

$$\bar{t}_a, t_d, t_p, t_o : \mathbb{T}.$$

2.3 Object Z

The main Object Z construct is the *class* definition. A class is a template for *objects* of that class: for each object of a class, the object's states are instances of the class' state schema and the object's state transitions are instances of the class' operation schemas. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Since Object Z has no standard conventions for handling timing and process control issues, it is necessary to model these issues explicitly in the class state. One such approach is Timed Object Z [12], which incorporates ideas from various Z-based approaches for specifying real-time requirements [20], [41]. The Timed Object Z approach consists of two conventions. First, environmental factors are modeled as functions of time and are included in the system state. In the timed collection example, the environment is modeled by functions $left$ and $right$. These functions represent the participation of the environment in Add and $Delete$ operations, respectively. The second extension is to include a global real-time clock, conventionally represented by a distinguished state attribute now. The clock may only be updated during an operation and the next operation must start as soon as the previous one is finished. In order to model the time taken between operations, it is thus necessary to introduce a $Wait$ operation.

2.3.1 Timed Collection in Object Z

The use of Timed Object Z is illustrated by the $TimedCollection$ class in Fig. 1. The generic function ps (purge stale) is defined in Fig. 2, e.g.,

$$ps(2\ \mu\mathrm{s}, \{(1\ \mu\mathrm{s}, b), (7\ \mu\mathrm{s}, c)\}) = \{(1\ \mu\mathrm{s}, b), (5\ \mu\mathrm{s}, c)\}.$$

The first schema of a class is called the state schema. It describes the various state *attributes* and the class *invariant*. The class attributes are divided into *primary* attributes and *secondary* attributes, which appear below a Δ separator placed in the declaration section of the state schema. The important aspect of secondary attributes in the context of the $TimedCollection$ classes is the fact that they are subject to change by every operation. For a detailed discussion of secondary attributes, see Dong et al. [14]. The predicate below the line in the state schema is called the class invariant. It describes the state properties that must be established initially and preserved by every operation.

In the $TimedCollection$ class, the primary attribute $mems$ denotes a finite set of time-stamped elements of the generic type X. The other primary attributes are the $left$ and $right$ environment variables. Since these variables represent interactions with the environment, they are not subject to change by any of the class operations.

In the $TimedCollection$ class, the secondary attributes are now, denoting the current time; and t_i, denoting the idle time since the completion of the last operation; the tuple $(t, oldest)$, denoting the oldest element being in the

[1] For ease of presentation (especially in the Object Z and Timed CSP versions), we adopt exact timing constraints in the timed collection example.

$TimedCollection[X]$

$left, right : \mathbb{T} \mathrel{+\!\!\!\!\rightarrow} X$
$mems : \mathbb{F}(\mathbb{T} \times X)$
Δ
$now : \mathbb{T}$ [current time]
$t_i, idle : \mathbb{T}$ [idling parameters]
$t : \mathbb{T}$ [oldest item]
$oldest : X$

$mems \neq \varnothing \Rightarrow (t, oldest) \in mems \wedge t = min\,\mathrm{dom}\,mems$
$\mathbf{let}\; rm = \{\tau : \mathrm{dom}(left \cup right) \mid \tau \geq now \bullet \tau - now\} \cup \mathrm{dom}\,mems \bullet$
$\quad rm \neq \varnothing \Rightarrow idle = min(rm)$

INIT

$mems = \varnothing \wedge now = 0\,\mu\mathrm{s}$
$t_i = 0\,\mu\mathrm{s}$

Add

$\Delta(mems)$
$e? : X$

$now \in \mathrm{dom}\,left$
$left(now) = e?$
$now' - now = t_a \wedge t_i' = 0\,\mu\mathrm{s}$
$mems' = ps(t_i + t_a, mems)$
$\quad \cup\{(t_o, e?)\}$

$Delete$

$\Delta(mems)$
$e! : X$

$mems \neq \varnothing$
$now \in \mathrm{dom}\,right$
$right(now) = e!$
$now' - now = t_d$
$t_i' = 0\,\mu\mathrm{s}$
$e! = oldest$
$mems' = ps(t_i + t_d,$
$\quad mems \setminus \{(t, oldest)\})$

$Purge$

$\Delta(mems)$

$t_i \in \mathrm{dom}\,mems$
$now' - now = t_p \wedge t_i' = 0\,\mu\mathrm{s}$
$mems' = ps(t_i + t_p, mems)$

$Wait$

$idle > 0\,\mu\mathrm{s}$
$now' - now = idle$
$t_i' = t_i + idle$

Fig. 1. Object Z Model of the Timed Collection.

collection, when it is nonempty; and the time variable $idle$, denoting the time until the next operation.

The INIT schema describes the allowed initial values for the class attributes, in this case, $mems$. The initial schema implicitly includes the state schema, so that the initial state includes all the class attributes and satisfies the class invariant. The timed collection initially contains no elements.

$[X]$

$ps : (\mathbb{T} \times \mathbb{F}(\mathbb{T} \times X)) \rightarrow \mathbb{F}(\mathbb{T} \times X)$

$ps(t, \varnothing) = \varnothing$
$\forall\, t : \mathbb{T};\; s : \mathbb{F}(\mathbb{T} \times X) \bullet$
$\quad ps(t, s) = \{(t_o, e) : s \mid$
$\quad\quad t_o > t \bullet (t_o - t, e)\}$

Fig. 2. Purge stale function.

The remaining four schemas are operation schemas which describe the allowed state transitions for the class. The declaration parts of operation schemas may include a Δ-list of those (primary) attributes whose values may change. By convention, any primary attribute not appearing in the Δ-list may not change value. The values of the secondary attributes are always subject to change. Every operation schema implicitly includes the state schema in unprimed form (the state before the operation) and primed form (the state after the operation).

The timing behavior of the $TimedCollection$ requires careful explanation. The Add operation may occur only when there is an item available on $left$. It updates timeouts in the existing collection and purges any stale items (this is described in the ps funtion definition), and adds the new item with the maximum timeout of t_o. This activity must take exactly t_a and the idle time is set to $0\,\mu$s. The $Delete$ operation is enabled only when the $right$ environment is willing to accept the oldest item; it communicates the item and then deletes it from the collection. The $Purge$ operation is invoked when no communication is possible before the first item goes stale ($t_i \in \operatorname{dom} mems$). Each of these operations also implicitly updates the $idle$ attribute according to the requirement in the data invariant. The $idle$ attribute always records the amount of idling required before the next action is enabled and the $Wait$ operation simply consumes this idle time.

2.3.2 Transition System Interpretation

Object Z has a three-stage semantics. The various operations in a class are given a standard Z semantics, then this is used to develop a transition-system semantics, which then determines an event-based process semantics [48]. The Z operation semantics is best viewed as describing a relation between initial and final states for each operation. The operations of a given class thus form a named collection of relations, which determines a transition system in which a given operation may fire exactly when its Z precondition is satisfied. The Z precondition of an operation schema describes the initial states for which there exists some final state satisfying the schema predicate. The process model for the class consists of all the sequences of operations/events which can be performed by objects of the class.[2]

For example, the $TimedCollection$ object starts with $mems$ empty, then evolves by successively performing either Add, $Delete$, $Purge$, or $Wait$ operations. This is sometimes expressed semiformally by an equation, such as

$$TCbeh \triangleq (Add \;\Box\; Delete \;\Box\; Purge \;\Box\; Wait);\; TCbeh.$$

Here, $TCbeh$ represents the behavior of the $TimedCollection$, $(\_ \Box \_)$ is the Object Z choice between operations,[3] and $(\_;\ \_)$ is the Object Z sequential composition. The choice of which operations are enabled at each point is determined by which preconditions are satisfied by the current state. As an example of a precondition calculation, consider the $Purge$ operation. The Z precondition is defined to be

$$\begin{aligned}&\exists mems' : \mathbb{F}(\mathbb{T} \times X);\\ &now', t_i', idle' : \mathbb{T};\\ &(t, oldest) : \mathbb{T} \times X \bullet Purge.\end{aligned}$$

By expanding the predicate part of the $Purge$ schema and simplifying, it can be shown that this is equivalent to

$$now - t_i \in \operatorname{dom} mems.$$

Thus, the $Purge$ operation may only occur when the oldest member of the timed collection has expired.

This entwining of behavioral control matters with algorithmic matters creates unnecessary complexity in the design process and fails to promote a clear separation of concerns. For example, in order to ensure that operations occur in some desired order, the designer must painstakingly craft the preconditions of all the operations in a class so as to ensure the desired interactions, and may even need to add unnecessary process state in order to represent control state. Since there is no way to progress time except through the action of an operation, it is necessary to introduce pseudooperations, such as $Wait$, whose sole purpose is to make sure that there is something to do at each point in time. Deep reasoning is required to demonstrate that time does indeed always progress. Perhaps most inconvenient is the fact that this use of preconditions to control the sequencing of transitions is incompatible with aspects of Z algorithmic refinement. In particular, refinement by weakening an operation's precondition is disallowed in Object Z. Weakening an operation's pre–condition would result in it being enabled more often, thus playing havoc with the process control structure of the original specification. For example, if the precondition for the $Purge$ operation was weakened, it would be possible for it to occur either before or after the expiry time stamp of data element, a result completely at odds with the purpose of the time stamp.

2.3.3 Summary

The $TimedCollection$ class describes the data state ($mems$) and operations of the timed collection well. However, Timed Object Z requires interactions with the environment and the progress of time to be micromanaged in an intrusive manner. The proliferation of additional attributes required to deal with process control and time result in significant overspecification of the system. All too frequently, deep reasoning is required to comprehend the subtle and complex interplays between operations and environment.

2.4 Timed CSP

Timed CSP [46] extends the well-known CSP (Communicating Sequential Processes) notation of Hoare [29] with timing primitives. CSP is an *event*-based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization.

CSP adopts a symmetric view of process and environment. Events represent a cooperative synchronization between process and environment. Both process and environment may control the behavior of the other by *enabling* or *refusing* certain events or sequences of events. Although CSP semantics are symmetric with respect to process and environment, we find it helpful in the following to use the words *request* and *block* as synonyms for enable and refuse, respectively, when referring to the environment.

2.4.1 Process Primitives

A process which may participate in event a and then act according to process description P is written

$$a@t \rightarrow P(t).$$

The event a is initially enabled by the process and occurs as soon as it is requested by its environment; all other events initially are refused. The event a is sometimes referred to as the *guard* of the process. The (optional) timing parameter t records the time, relative to the start of the

[2] Smith's semantics also includes *ready* sets which record the enabled events at each step.

[3] Note the clear influence of CSP.

process,[4] at which the event a occurs and allows the subsequent behavior P to depend on its value.

The second form of sequencing is process sequencing. A distinguished event $\surd$ is used to represent and detect process termination. The sequential composition of P and Q, written $P;Q$, acts as P until P terminates by communicating $\surd$ and then proceeds to act as Q. The termination signal is hidden from the process environment and therefore occurs as soon as enabled by P. The process which may only terminate is written SKIP.

The parallel composition of processes P and Q, synchronized on event set X, is written

$$P|[X]|Q.$$

No event from X may occur in $P|[X]|Q$ unless enabled jointly by both P and Q. When events from X do occur, they occur in both P and Q simultaneously and are referred to as *synchronizations*. Events not from X may occur in either P or Q separately, but not jointly. For example, in the process described by

$$(a \to P)|[a]|(c \to a \to Q),$$

all a events must be synchronizations between the two processes. Since a is not enabled initially by the right-hand process, a cannot occur in the left-hand process until the right hand process has performed a c event and the a event becomes enabled in both processes.

In an asynchronous parallel combination

$$P \; ||| \; Q,$$

both components P and Q execute concurrently without any synchronizations.

Diversity of behavior is introduced through two choice operators. The external choice operator allows a process a choice of behavior according to what events are requested by its environment. The process

$$(a \to P) \Box (b \to Q)$$

begins with both a and b enabled. The environment chooses which event actually occurs by requesting one or the other first. Subsequent behavior is determined by the event which actually occurred, P after a and Q after b, respectively. When the range of choices is large (possibly infinite), external choice may be written in an intentional form,

$$\Box a : A \bullet P(a),$$

which allows the environment to choose any event a from a set A and subsequent behavior is determined by $P(a)$.

Internal choice represents variation in behavior determined by the internal state of the process. The process

$$a \to P \sqcap b \to Q$$

may initially enable either a, b, or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice. Again, an intentional form is allowed.

An important derived concept in CSP is the notion of *channel*. A channel is a collection of events of the form $c.n$: the prefix c is called the *channel name* and the collection of suffixes is called the *values* of the channel. When an event $c.n$ occurs, it is said that *the value n is communicated on channel c*. When the value of a communication on a channel is determined by the environment (external choice), it is called an *input* and when it is determined by the internal state of the process (internal choice) it is called an *output*. It is convenient to write $c?n : N \to P(n)$ to describe behavior over a range of allowed inputs instead of the longer $\Box n : N \bullet c.n \to P(n)$. Similarly, the notation $c!n : N \to P(n)$ is used instead of $\sqcap \; n : N \bullet c.n \to P(n)$ to represent a range of outputs. Expressions of the form $c?n$ and $c!n$ do not represent events, the actual event is $c.n$ in both cases.

The interrupt process $P_1 \bigtriangledown e \to P_2$ behaves as P_1 until the first occurrence of interrupt event e, then the control passes to P_2.

Recursion is used to given finite representations of nonterminating processes. The process expression

$$\mu P \bullet a?n : \mathbb{N} \to b!f(n) \to P$$

describes a process which repeatedly inputs a natural on channel a, calculates some function f of the input, and then outputs the result on channel b. CSP specifications are typically written as a sequence of simultaneous equations in a finite collection of process variables. Such a specification $\vec{X} \triangleq \vec{F}(\vec{X})$ is implicitly taken to describe the solution to the vector recursion $\mu\vec{X} \bullet \vec{F}(\vec{X})$.

In general, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state. The approach adopted by CSP is to allow a process definition to be parameterized by state variables. Thus, a definition of the form

$$P_{n:N} \triangleq Q(n)$$

represents a (possibly infinite) family of definitions, one for each possible value of n. There is no inherent notion of process state in CSP, but rather these annotations are a convenient way to provide a finite representation of an infinite family of process descriptions.

2.4.2 Timing Primitives

To the standard CSP process primitives, Timed CSP adds two time specific primitives, the delay and the timeout.

A process which allows no communications for period t then terminates is written WAIT t. The process WAIT t; P is used to represent P delayed by time t.

The timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline. The process

$$(a \to P) \triangleright [t] Q$$

will try to perform $a \to P$, but will pass control to Q if the a event has not occurred by time t, as measured from the invocation of the process.

[4] This may be nonzero because the process must wait until the event is requested by its environment.

$$
\begin{aligned}
& TC_{\{(t,a)\}\cup s} \mathrel{\widehat{=}} \\
& \qquad (\mathit{left}?e : X@t_i \rightarrow \textsc{Wait}\ t_a; \\
& \qquad TC_{ps(t_i+t_a,\{(t,a)\}\cup s)\cup\{(t_o,e)\}} \ \Box \\
& \qquad \mathit{right}!a@t_i \rightarrow \textsc{Wait}\ t_d; \\
& \qquad TC_{ps(t_i+t_d,s)}) \rhd \{t\} \\
& \qquad\qquad \textsc{Wait}\ t_p;\ TC_{ps(t+t_p,tl)}
\end{aligned}
$$

where $(t,a) = \mathit{find\_oldest}(\{(t,a)\}\cup s)$.

$$
\begin{aligned}
& [X] \\
& \mathit{find\_oldest} : \mathbb{P}_1(\mathbb{T}\times X) \rightarrow \\
& \qquad\qquad (\mathbb{T}\times X) \\
\hline
& \forall s : \mathbb{P}_1(\mathbb{T}\times X) \bullet \exists (t,e) : s \bullet \\
& \qquad t = \mathit{min}(\mathrm{dom}\, s) \\
& \qquad \mathit{find\_oldest}(s) = (t,e)
\end{aligned}
$$

Fig. 3. Behavior when nonempty.

2.4.3 The Timed Collection Example

The timed collection can be modeled as a process with two channels, $left$ and $right$, respectively. Elements are added to the collection through communications on the $left$ channel and removed through communications on the $right$ channel. The timing issues of the timed collection can be described using (Timed CSP's) delay and timeout constructs.

The initial state is represented by the empty set.

$$TimedCollection \mathrel{\hat{=}} TC_{\emptyset}.$$

When the first element joins the collection, it is stamped with a timeout and the time taken to update the process state is represented by a delay.

$$TC_{\emptyset} \mathrel{\hat{=}} left?e : X \rightarrow \textsc{Wait}\ t_a; TC_{\{(t_o,e)\}}.$$

When the collection is nonempty, the process is ready to accept $left$ or $right$ events. The staleness stamps are updated with each communication and state update delays are introduced. In the event of no communication occurring before the oldest element becomes stale, all stale elements are purged (see Fig. 3).

2.4.4 Summary

For such an example, Timed CSP is superior to Object Z as a means of describing process control. Timed CSP also handles the timing issues of delays and timeouts simply and elegantly. The allowed sequences of events are clearly and concisely determined by the CSP code, there is no need to calculate preconditions, nor is any other form of deep reasoning required to understand the ways in which the timed collection may evolve. The Timed Object Z approach results in a too complex model which overspecifies this simple system, even though the timed collection example does not make use of the multithreading and synchronization capabilities of Timed CSP which are clearly well beyond the scope of Object Z's atomic state transition semantics. On the other hand, the syntactic treatment of internal state in the above is complex and unwielding, distracting strongly from the basically elegant treatment of the delay and timeout issues. Although, for example, Roscoe's CSP$_M$ language [45] includes some powerful data modeling primitives, CSP still has no standard support for state modeling in the form of mathematical toolkits and libraries, nor are there modular techniques for constructing and reasoning about the complex internal state.

3 TCOZ

In many ways, Object Z and Timed CSP complement each other in their capabilities. Object Z has strong data and algorithm modeling capabilities. The Z mathematical toolkit is extended with object-oriented structuring techniques. Timed CSP has strong process control modeling capabilities. The multithreading and synchronization primitives of CSP are extended with timing primitives. Moreover, both formalisms are already strongly influenced by the other in their areas of weakness. Object Z supports a number of primitives which have been inspired by CSP notions such as external choice and synchronization. CSP practitioners tend to make use of notation inspired by the Z mathematical toolkit in the specification of processes with the internal state. This is not surprising, given their joint associations in the Programming Research Group in Oxford. Another important connection is the well-known duality between the state transition behavioral model and the event-based behavioral model [28], which makes it a simple matter to develop complementary semantics for the two languages.

Given these factors, it is natural to consider the possibility of blending the two notations into a more complete approach to modeling real-time and/or concurrent systems. Fischer [21] and Smith [49] have independently suggested CSP-style semantics for Object Z classes in which operation calls become CSP events. Operation names take on the role of CSP channels, with input and output parameters being passed down the operation channel as values. This view fits nicely with the Object Z interpretation of operations being atomic, but is not well-suited to considering multithreading and real-time. Restricting operations to atomic events collapses the spatial and temporal aspects of operations; everything happens at a single point and instantaneously. Identifying channel names with operation names creates unnecessary tensions between the data and process views of objects and considerably reduces the potential for reuse of operation definitions. Another approach is that taken by Galloway in his CCZ language [23], based on Z and (value-passing) CCS. There Z operation schemas do not appear as events, but instead appear as prefixes to parameterized CCS output processes. The effect of the operation schema is to restrict the allowed output values in the associated process and to update the values of the process state parameters. While this approach effectively disentangles the communication interface from the operational structure, the need to associate every occurence of an operation with a following output process is a major syntactic inconvenience.

The approach taken in the TCOZ notation is to identify operation schemas (both syntactically and semantically) with (terminating) CSP processes that perform only state

$$
\begin{array}{|l}
\textit{Add}_0 \\
\hline
\Delta(\textit{mems}) \\
e? : X \\
t_i? : \mathbb{T} \\
\hline
\delta = t_a \\
\textit{mems}' = \\
\qquad ps(t_i? + t_a, \textit{mems}) \cup \{(t_o, e?)\} \\
\hline
\\
\textit{Add} \mathrel{\widehat{=}} [e : X;\ t_i : \mathbb{T}] \bullet \\
\qquad\qquad \textit{left}?e@t_i \rightarrow \textit{Add}_0
\end{array}
$$

Fig. 4. Add operation.

update events; to identify (active) classes with nonterm–inating CSP processes; and to allow arbitrary (channel-based) communications interfaces between objects.

The syntactic implications of this approach are that the basic structure of a TCOZ document is the same as for Object Z. A document consists of a sequence of definitions, including type and constant definitions in the usual Z style. TCOZ varies from Object Z in the structure of class definitions, which may include CSP channel and processes definitions. In fact, all operation definitions in TCOZ are considered to define CSP processes. The CSP view of an operation schema is that it describes all the sequences of update events which change the system state as required by the schema predicate. The exact nature and granularity of these update events is left undetermined in TCOZ (at least at the syntactic level), but by allowing an operation to consist of a number of events, it becomes feasible to specify its temporal properties when describing the operation. Since operation schemas take on the syntactic role of CSP processes, they may be combined with other schemas and even CSP processes using the standard CSP process operators. Thus, it becomes possible to represent true multithreaded computation even at the operation level, something that would not be possible with CCZ approach. The Fischer/Smith approach of identifying operation names with CSP channels is not followed, channels are given an independent, first class role. This allows the communications and control topology of a network of objects to be designed orthogonally to their class structure. The CSP channel mechanism is the only (dynamic) way to pass information between objects as the state of objects is encapsulated by hiding all update events.

3.1 Defining Operations

The operation schema is the basic tool for describing state change in TCOZ. In order to allow treatment of timing issues in schema definitions, a distinguished identifier δ is introduced to represent the duration of the state calculations performed by the operation. When δ does not appear in the definition of an operation, the default interpretation is that there be no constraint on the duration of the operation, although individual specification documents may choose to adopt a different convention.

Although the schema is the basic tool, the true power of TCOZ comes from the ability to make use of Timed CSP primitives is describing the process aspects of an operation's behavior. All operation definitions in TCOZ are in fact Timed CSP process definitions, with operation schema being given the syntactic status of terminating Timed CSP processes.

As an example, consider the specification of the Add operation (see Fig. 4) of the timed collection example. The actual state-change allowed by the operation schema remains unchanged from the Timed Object Z version, but the timing characteristics of the operation are expressed by the condition $\delta = t_a$, rather than $now' - now = t_a$.

Since TCOZ operations are identified with terminating CSP processes, it is natural to allow their definition in terms of CSP primitives, such as event sequencing, as well as through the schema calculus. The novelty of the full TCOZ version of Add lies in the adoption of CSP primitives in its definition. Item inputs are communicated to the Add operation along a channel $left$. This definition of Add says that after the parameter e has been input on channel $left$ at time t_i, the state-calculation Add_0 is performed. Several aspects of TCOZ name-space conventions are raised by this definition.

First, observe that the parameters e and t_i occur in Add_0 with Z-style input decorations and in Add without them. In TCOZ, the convention is adopted that the true name of all parameters is the undecorated version. In an operation schema, the ? and ! decorations are used solely to distinguish between inputs and outputs for the purposes of defining the binding semantics of the operation. This is analogous to the convention of using primed and unprimed versions to indicate the final and initial values of a state attribute. In fact, all parameters are treated in the same way as state attributes, with the exception that state attributes are available in every name environment in a class definition. The reason for this convention may be clearly seen in the following example:

$$
\begin{aligned}
&[y! : \mathbb{N} \mid y! = f(x)]; c.y \rightarrow \\
&\qquad [\Delta(x); y? : \mathbb{N} | x' = g(y?)].
\end{aligned}
$$

Clearly, it does not make sense to decorate the y in $c.y$ with either the ! or the ?, any more than it would make sense to use a $'$ decoration on an attribute variable appearing in a communication.

This leads to the second observation that may be made of the Add definition. The $[e\ :\ X;\ t_i : \mathbb{T}] \bullet \_$ construct is a local block definition in the state guard style (state guards are explained further below). The other forms of local blocks are the intentionial forms of both internal and external choice, which use the usual Z-style schema-text conventions. For example,

$$\sqcap\, n :\ \mathbb{N}\ |n < 5 \bullet c!n \rightarrow P$$

or

$$\Box\ n : \mathbb{N} | n < 5 \bullet c?n \rightarrow P.$$

The state guard serves as an alternate form of external choice, so that the Add process is equivalent to

$$\Box\ e : X;\ t_i : \mathbb{T}\ \bullet\ left?e@t_i \rightarrow Add_0.$$

$$\begin{array}{l}
\underline{\quad Delete_0 \qquad\qquad\qquad} \\
\Delta(mems) \\
t_i? : \mathbb{T} \\
\hline
mems \neq \varnothing \wedge \delta = t_d \\
mems' = ps(t_i + t_d, \\
\qquad\qquad mems \setminus \{(t, oldest)\}) \\
\hline
\end{array}$$

$$\begin{array}{l}
Delete \mathrel{\widehat{=}} [t_i : \mathbb{T} \mid mems \neq \varnothing] \bullet \\
\qquad right!oldest@t_i \rightarrow Delete_0
\end{array}$$

Fig. 5. Delete operation.

In TCOZ, the internal and external choice operators must be used explicitly. Decorated communications of the form $c!n$ and $c?n$ have no conventional meaning in TCOZ, simply being syntactic sugar for $c.n$. They are allowed simply as a form of comment to emphasize the intended direction of communication.

In TCOZ, the local name space may be changed either by a local block definition as above or else by the occurence of an operation schema. An operation schema removes all its input parameters from scope and replaces them with its output parameters. The output parameters then become available for use in subsequent communication events or as inputs to subsequent operation schemas.

In the case of the $Delete$ operation (see Fig. 5), the communication of the deleting element must precede the updating of the collection state and in fact is the enabling event for the operation. Since the name convention is that outputs are only available to the right of a schema, this behavior cannot be described using an output parameter. Instead, the update operation is described as a simple state update which removes the oldest item (and any others that become stale). The overall delete operation consists of this schema guarded by a communication on the $right$ channel.

The first part of the definition of $Delete$ is a novel process control primitive known as a *state guard*.[5] The adoption of a state guard mechanism allows TCOZ to adopt a proper separation between algorithm and process design issues. The sequencing of activities in an object is controlled explicitly through state guards rather than implicitly through the operation preconditions. In this way, it becomes possible to reclaim the Z-style operation design and decomposition techniques abandoned by standard Object Z.

Every process definition has (at least) an initial state which may be addressed using schema notation. This is the function of the first part of the expression defining $Delete$. It is a schema-based method of restricting the action of the process to initial states for which the collection is nonempty. For other states this process will *deadlock* or *block*, refusing any communication.

Note that the precondition requirement in the $Delete_0$ schema, though identical, could not achieve the desired restriction on the behavior of $Delete$. Failure to satisfy a precondition when control is passed to an operation instead results in *divergence*, or unspecified subsequent behavior. $Delete_0$ places no restrictions at all on its behavior when the initial queue is empty. The precondition is the state-based equivalent of process divergence and the guard is the state-based equivalent to process deadlock.

For every operation P (even those constructed using the process calculus), the collection of initial states for which the process will not diverge is called its *precondition* (written $\mathrm{pre}\ P$) and the collection of states for which it will not deadlock is called its *guard* (written $\mathrm{grd}\ P$).

3.2 Schemas and Processes

A schema expression describes a relationship on or between process state/s, while a process expression describes the overall behavior or evolution of a process. The Z semantic model for operation schemas consists of sets of variable *bindings*, mappings from variable names to values. An important point is that these sets may be infinite when the operation allows unbounded nondeterminism. Timed CSP has a number of semantic models, but the most common consists of sets of tuples consisting of a *timed trace* (a sequence of time stamped events) and a *refusal* (a record of when events are refused by the process). The trace/refusal pair is called a *failure* and the model the timed failures model. The basic approach taken in the TCOZ semantics is to adopt the timed failures semantic model and to provide an interpretation of the Z semantic model in terms of failures and divergences, though some variations are required to make this possible. First, a variable binding is added to represent the initial values of all the process attributes. Second, a new class of events, referred to as *update* events, is introduced to represent changes to the process attributes. The resulting model is called the state/failures/divergences model. The state of the process at any given time is the initial state updated by all of the updates that have occurred up to that time. If an event trace terminates (that is, if a $\surd$ event occurs), then the state at the time of termination is called the *final* state. Finally, since the unbounded nondeterminism potentially present in Z schemas cannot be treated properly using finite-traces, an infinite-trace variation of the timed-failures model, due to Mislove et al. [43], is adopted.

The process model of an operation schema consists of all initial states and update traces (terminated with a $\surd$) such that the initial state and the final state satisfy the relation described by the schema. If no legal final state exists for a given initial state, the operation diverges immediately. In the timed-failures model, divergence is represented by allowing arbitrary behavior from the time of divergence.

The process model for the state guard consists of replacing the trace part of every behavior of the guarded process whose initial state does not satisfy the state guard with the empty trace. The empty event trace describes the process being blocked by the failure of the state guard. In addition, divergence cannot occur if the state guard is not satisfied.

[5] Although if-then style commands appear in several dialects of CSP, for example CSP$_M$ [45], we believe that TCOZ is unique in adoptiong the state guard as a separate primitive in the style of Morgan's version of the guarded command language [44].

Since schema calculus operators cannot sensibly be applied to arbitrary CSP processes, it is necessary to strictly distinguish the schema calculus from the process calculus. The two exceptions to this are the typecasting of operation schema expressions as terminating processes and of initial state schema expressions as state guards. In all other circumstances, the schema and process calculi are separate and distinct. For example, if P and Q are operations schema expressions, the expression $a \rightarrow (P \wedge Q)$ is legal while the expression $(a \rightarrow P) \wedge Q$ is not. The full power of the schema calculus may be used to construct schema expressions, but once a schema expression has been cast into a process-like role, it may no longer act in a schema-like role.

Some existing Object Z schema calculus operators, such as $\_\Box\_$, $\_|\_$, and $\_;\ \_$, have name-sakes with similar semantics in the CSP process calculus. The convention adopted in TCOZ is that the CSP operator is intended, and only "pure logic" schema calculus operators are allowed in TCOZ. This is justified by the superior algebraic properties of the CSP operators.

When operations are combined using the concurrency primitives $\_|\_$ and $\_|||\_$, the designer is exposed to all the usual dangers of shared variable concurrency. The operation $OS_1||OS_2$, where OS_1 and OS_2 are operation schema, will synchronize on all state update events on variables in the respective delta-lists. Thus, $OS_1||OS_2$ will have much the same process properties as $OS_1 \wedge OS_2$, with the exception that when the operations are inconsistent for a given initial state, the concurrent composition will deadlock while the logical composition will diverge. For example, consider

$$SQRT \triangleq [x, x' : \mathbb{N} | x > 0 \wedge x'^2 = x]$$

$$HALF \triangleq [x, x' : \mathbb{N} | x' * 2 = x].$$

The operations $SQRT$ and $HALF$ can agree only in the case where $x = 4$. When either of the operations is undefined (for example, when $x = 2$, $SQRT$ is undefined), $SQRT||HALF$ will diverge. When both are defined but in disagreement (for example, when $x = 16$), $SQRT||HALF$ will deadlock at some unspecified time. The process $SQRT \wedge HALF$ is just $[x, x' : \mathbb{N} | x = 4 \wedge x' = 2]$ and will never deadlock. The concurrent composition $OS_1|||OS_2$ is even less well-behaved, and every variable may be updated in any way allowed by either OS_1 or OS_2. Such a situation is likely to be very difficult to analyze. We strongly recommend that concurrent composition of operations be used sparingly, preferably only in cases where the operations have disjoint delta-lists. Shared data structures should only be utilized when properly protected by the object encapsulation mechanism.

3.3 Active and Passive Objects

The definition in a class of the distinguished process name MAIN indicates that the class is being defined as *active*. As in CCZ [23], the MAIN process is used to determine the behavior of objects of an active class after initialization. Initialization is treated in the usual way through the INIT schema. Active objects have their own thread of control and their mutable state attributes and operation definitions are fully encapsulated (update events are hidden). Distinct objects, even of the same class, share no data and can experience no shared variable interference. Other objects can neither reference an active object's state attributes nor invoke any of its local operations. Only local constants, such as the object identity attribute $self$, may be accessed by other classes. All dynamic interactions with an active object must take place through the CSP channel communication mechanism. Active objects are considered to have the syntactic properties of process identifiers and may be composed using CSP operators.

The MAIN operation is optional in a class definition. If a class is defined without a MAIN process it is called a *passive* class. Passive objects are controlled by other objects in a system and their state and operations are fully available to the controlling object (unless explicitly hidden). The appearance of MAIN clearly distinguishes the definition of active objects and passive objects in a system.

Returning to the timed collection example, the existence of environmental obligations and the need to purge stale elements means that the timed collection class must have its own thread of control. Assuming that the class operations are defined as in Section 3.1, the timed collection behavior is defined by a MAIN process similar to the Timed CSP version presented in Section 2.4.3.

$$\begin{aligned}\text{MAIN} \triangleq \mu TC \bullet [mems = \emptyset] \bullet Add;\ TC \Box \\ [mems \neq \emptyset] \bullet ((Add \Box Delete) \rhd [t]Purge); TC.\end{aligned}$$

The most striking difference lies in the use of the operation schemas to subsume the role of the complex annotations present in the Timed CSP version. This represents a clearer and more structured presentation of the basic control logic. A second difference lies in the use of the state guard construct to distinguish between the empty and nonempty behaviors of the timed collection, thus saving the need to define separate empty and nonempty processes.

3.4 Communication Channels

The class 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.

We have decided on this convention in TCOZ, primarily because we see no compelling reason to associate types with channels and can see some minor advantages in not doing so. Certainly, the timed failures semantics does not require that channels be typed. Channel communication events are simply pairs consisting of the channel name and the value communicated. Neither does the Z bindings semantics requires the value be typed, the Z semantics is modeled in untyped set theory. The main argument in favor of extending the typing conventions to channels is that it provides redundancy which will guard against silly errors, such as trying to read an input of one type when the output was of another type. Balanced against this, we believe that the ability to send many forms of data over a channel plays a vital role in lowering the complexity of class interfaces and this lower complexity also reduces the likelihood of

$$\begin{array}{l}
mems : \mathbb{F}(\mathbb{T} \times X) \\
left, right : \mathbf{chan} \\
\Delta \\
t : \mathbb{T} \\
oldest : X \\
\hline
mems \neq \varnothing \Rightarrow \\
\qquad (t, oldest) \in mems \\
\qquad t = min \operatorname{dom} mems
\end{array}$$

Fig. 6. State schema.

errors such as listening for an input on the wrong channel. Furthermore, we believe it is more instructive to group logically related communications (such as those pertaining to a particular protocol) than to group communications with identical type but logically unrelated function. Thus, we currently adopt the untyped convention with the intention of evaluating our position again as our experience using the notation increases. An interesting analysis of the general benefits of typed and untyped syntax has been made by Lamport and Paulson [36].

Contrary to the conventions adopted for internal state variables, channels are viewed as global rather than as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects. In the situation of multiple instances of objects of the same class in a system, those objects will all share the same channel. For example, if $\mathcal{O}$ is a sequence of objects of class C with channel c, then in the process

$$||| \; i : \operatorname{dom} \mathcal{O} \bullet \mathcal{O}(i),$$

each of the objects $\mathcal{O}(i)$ communicates with the environment by sharing channel c with every other object. In the general case, there is no way for the environment to know which of the objects it is communicating with when using channel c. If it is necessary to know which object the environment is communicating with, the object identity attribute $self$ [17] can be included in the communication, for example,

$$c.(self,\ message).$$

(This technique is used frequently in the lift case study.) The introduction of channels to TCOZ reduces the need to reference other classes in class definitions, thereby further enhancing the modularity of system specifications.

Consider once again the timed collection example. Using the TCOZ conventions, the class state can be significantly simplified from the Timed Object Z version (see Fig. 6). The sole remaining primary class attribute is the actual collection itself, and none of the timing attributes are required. In addition to the list of $mems$, the state schema must declare channels $left$ and $right$. These channels serve much the same role as the corresponding environment variables in the Timed Object Z version, but here that role is better defined in terms of the CSP process model. The secondary attributes $oldest$ and t remain useful in simplifying the operation definitions and are retained.

3.5 The Timed Collection

Bringing together the various aspects of the TCOZ timed collection introduced above, we are able to present the entire class definition in Fig. 7.

This specification represents a more concise, flexible, and scalable treatment of both process and state than is possible in either Object Z or Timed CSP. The structure of the process' internal state and communications interfaces are prominently documented. The structured schema-based approach to describing state transitions, supported as it is by the full power of the Z toolkit and the schema calculus, is better able to handle large and complex process state than the essentially ad hoc state annotation conventions of CSP. Making use of the Timed CSP process definition conventions removes the need to consider process control matters in operation schemas. There is a clear separation of process control and algorithmic matters, which simplifies the description of both.

3.6 Composing Classes

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. Essentially, all definitions are pooled with the following provisions: Inherited type and constant definitions and those declared in the derived class are merged. The state and initialization schemas of inherited classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.

TCOZ extends Object Z with two new class constructs, channels **chan** and main behavior MAIN. Channels are treated as normal state constant attributes; therefore, they are pooled into the derived classes. Channel renaming is the same as state attribute renaming. Since new classes will generally have new behaviors, the MAIN class definition is never inherited. As for all other class definitions, a class extension must include a MAIN definition if the class is to be active. The rules for (active and passive) class inheritance are:

- A new active class may be derived from an existing active class by defining a new MAIN process.
- A new active class may be derived from an existing passive class by defining a MAIN process.
- A new passive class can also be derived from an existing active class by not defining a new MAIN process.
- A new passive class can be derived from an existing passive class by not defining a MAIN process.

A composite object which contains active objects is also an active object (a MAIN definition must appear in the composite object class). Active objects are responsible for their own intialization, so a composite object will often not require an explicit INIT schema. An example of this is the Buffered-Consumer/producer class in Section 3.7. Since it has no passive internal state, it requires no explicit initialization. Another result of the encapsulation of local state in active objects is the inability of a composite class to refer to the local state attributes and operations of a

$TimedCollection[X]$

$mems : \mathbb{P}(\mathbb{T} \times X)$
$left, right : \mathbf{chan}$
Δ
$t : \mathbb{T}$
$oldest : X$

$mems \neq \varnothing \Rightarrow (t, oldest) \in mems \wedge t = min\,\mathrm{dom}\,mems$

INIT

$mems = \varnothing$

Add_0

$\Delta(mems)$
$e? : X$
$t_i? : \mathbb{T}$

$\delta = t_a \wedge$
$mems' = ps(t_i? + t_a, mems)$
$\quad \cup\{(t_o, e?)\}$

$Delete_0$

$\Delta(mems)$
$t_i : \mathbb{T}$

$mems \neq \varnothing \wedge \delta = t_d$
$mems' = ps(t_i + t_d,$
$\quad (mems \setminus \{(t, oldest)\})$

$Add \mathrel{\widehat=} [e : X;\ t_i : \mathbb{T}] \bullet left?e@t_i \rightarrow Add_0$
$Delete \mathrel{\widehat=} [t_i : \mathbb{T} \mid mems \neq \varnothing] \bullet right!oldest@t_i \rightarrow Delete_0$

$Purge$

$\Delta(mems)$

$\delta = t_p \wedge mems' = ps(t + t_p, mems)$

$\mathrm{MAIN} \mathrel{\widehat=} \mu\, TC \bullet [mems = \varnothing] \bullet Add;\ TC \;\Box$
$\qquad [mems \neq \varnothing] \bullet ((Add \;\Box\; Delete) \rhd\{t\}\ Purge);\ TC$

Fig. 7. Timed Object Z model of the Timed Collection.

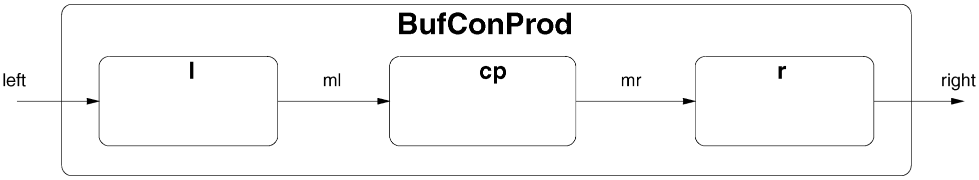

Fig. 8. Buffered consumer producer.

component object. Only class constants such as the object identity $self$ may be accessed.

3.7 A Multithreaded Example

The timed collection example made no use of the multithreaded capabilities of the TCOZ notation. In this section, a specification of a standard buffered consumer/producer process is presented (see Fig. 9) as a demonstration these aspects of TCOZ.

A simple consumer/producer process accepts inputs of type L on its $left$ channel, calculates some function $\mid f : L \rightarrow R$ of the inputs, and outputs the result on its $right$ channel.

To ensure that all inputs are accepted and outputs are received, the process is buffered left and right with timed collection processes (see Fig. 10). The buffered process consists of a left buffer, a right buffer, and an internal consumer/producer, as depicted in Fig. 8.

Correct hookup of the timed collection buffers to the consumer/producer is achieved by *renaming* the various internal channels. The renaming convention is the same as for Object Z and Timed CSP, that is, $P[a/b]$ is P with all occurrences of b replaced by a. Intermediate channels ml and mr are introduced as the internal interfaces to the left and right buffers, respectively. In order to retain a consumer-/producer-like interface, the $right$ channel of the left buffer is renamed to ml, the $left$ channel of the right buffer renamed to mr and the $left$ and $right$ channels of the internal consumer/producer are renamed to ml and mr, respectively.

The internal interfaces are protected from environmental interference by *hiding* them. The hiding notation is the same as for both Object Z and Timed CSP, that is, $(P \setminus c)$ is P with c protected from external influence. In the case where P is process-like and c is a channel, this has the important result of freeing communications on c from the requirement of synchronizing with the environment. Thus, communications on mr and ml occur as soon as the local processes are ready and cannot be blocked by any other entity.

The $BufConProd$ class definition allows true multithreading of the two buffers and the consumer/producer.

ConProd

$$f : L \rightarrow R$$

$$\mathit{left}, \mathit{right} : \textbf{chan}$$

Process

$$i? : L$$
$$o! : R$$

$$\delta = t_p \wedge o! = f(i?)$$

$$\textsc{Main} \mathrel{\widehat=} \mu\, C \bullet [i : L] \bullet \\ \mathit{left}?i \rightarrow \mathit{Process}; \\ \mathit{right}!o \rightarrow C$$

Fig. 9. Consumer/producer class.

For example, the left buffer may be accepting a new item at the same time as the consumer/producer is processing another item, at the same time as the right buffer is releasing yet another item to the environment. This concurrency is coerced into smooth cooperation through the requirement for synchronization between the processes when communication occurs on the internal channels ml and mr.

The addition of these CSP process structuring features represents a significant advance in the scope and size of systems which may be addressed by the Object Z approach.

3.8 Complex Network Topologies

The syntactic structure of the CSP synchronization operator is convenient only in the case of pipeline-like communication topologies. Expressing more complex communication topologies generally results in unacceptably complicated expressions. For example, consider the communication topology shown in Fig. 11, where 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

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

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

BufConProd

$$l : \mathit{TimedCollection}[L][ml/\mathit{right}]$$
$$r : \mathit{TimedCollection}[R][mr/\mathit{left}]$$
$$cp : \mathit{ConProd}[ml/\mathit{left}, mr/\mathit{right}]$$

$$\textsc{Main} \mathrel{\widehat=} \\ l\,|[\,ml\,]|\,cp\,|[\,mr\,]|\,r \setminus ml, mr$$

Fig. 10. Buffered cons/prod model.

The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology described so elegantly by Fig. 11. One reason for this is that it artificially suggests a dominant role for process A. Equivalent, but superficially very different, formulations of the network could be used to assign this "dominant" role to either B or C. We believe that network topologies can be better described by adopting a notation inspired by the graph-based approach embodied in Fig. 11.

A *network topology abstraction* is an expression of the form (Fig. 12).

The formal network parameters are abstract names that represent the process that form the nodes of the network topology. It is necessary to use abstract names to allow separate incarnations of processes with identical definition. The formal network topology describes a finite graph in which each edge or *network connection* generally represents a private channel connection between network nodes. Multiple connections between processes and connections between multiple processes over a single channel are represented by multiple connection expressions. In this case, the channel becomes a party line between the various participating processes.

A network topology abstraction describes a function, which we call a *network constructor*, that builds a process network using its process arguments as nodes. The network constructor associated with a given network abstraction is built by processing the formal parameters sequentially. At each stage there is a list of remaining formal parameters. Suppose v is the next remaining formal parameter. The local channels on which v is not synchronizing are decorated to avoid unwanted synchronizations. The local channels on which it is synchronizing with any of the remaining parameters are used to create a synchronization with the remaining network. Then the rest of the formal parameters

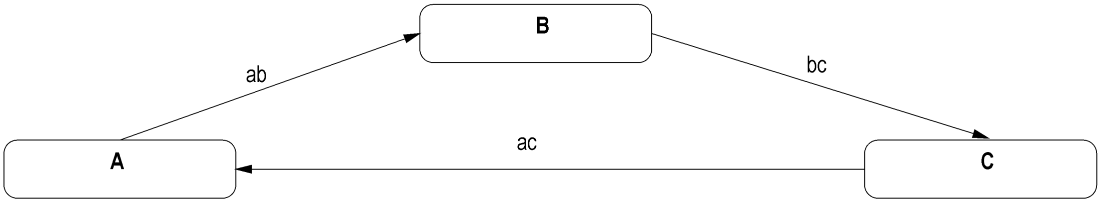

Fig. 11. Two-dimensional communication topology.

$$\begin{array}{l} (\| \; v_1, v_2, v_3 \ldots \bullet \\ \quad v_1 \xleftrightarrow{ch_{12}} v_2; \\ \quad v_2 \xleftrightarrow{ch_{23}} v_3; \\ \quad v_3 \xleftrightarrow{ch_{13}} v_1; \; \ldots). \end{array}$$

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

$$\begin{array}{l} v_1 \xleftrightarrow{ch_{12}} v_2; \\ \quad v_2 \xleftrightarrow{ch_{23}} v_3; \\ \qquad v_3 \xleftrightarrow{ch_{13}} v_1; \; \ldots \end{array}$$

are called the *formal network topology*.

Fig. 12. Network topology abstraction.

are treated one at a time to construct an expression for that remaining network. Finally, all the local channels are hidden and any decorations removed.

The above network topology abstraction thus describes the following network constructor (Fig. 13).

The system in Fig. 11 can be described by applying a suitable network topology abstraction to the processes A, B, and C.

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

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 Fig. 11 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,

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

As a further example of the use of the network topology convention, note that the behavior of the $BufConProd$ object can also be expressed by

$$\textsc{Main} \triangleq || \; (l \xleftrightarrow{ml} cp \xleftrightarrow{mr} r).$$

The network topology convention is particularly useful for describing the interactions between floors (requests), lift-controller, and elevators in the lift system.

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

Fig. 13. Network topology constructor.

We believe this TCOZ network topology convention to be a novel and potentially useful addition to the basic CSP notation. For example, the directed parallel operator in the CSP_M language, while adopting similar syntax, addresses the different problem of synchronizing on differently named channels without the burdensome use of renaming.

4 The Lift Case Study

The multilift system is a standard example used to demonstrate the expressive power of various specification techniques in modeling concurrent reactive systems.[6] 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 [51]. 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 [47] 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 [15] 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, centralized control-model because of the Object Z's single thread semantics. Neither the CSP nor 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 behavior of the lift system;
- a true multithreaded 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.

[6] Many formal notations have been applied to the lift system. For example, the UNITY model [7], [9], Raddle [22], and Constraint Nets [52].

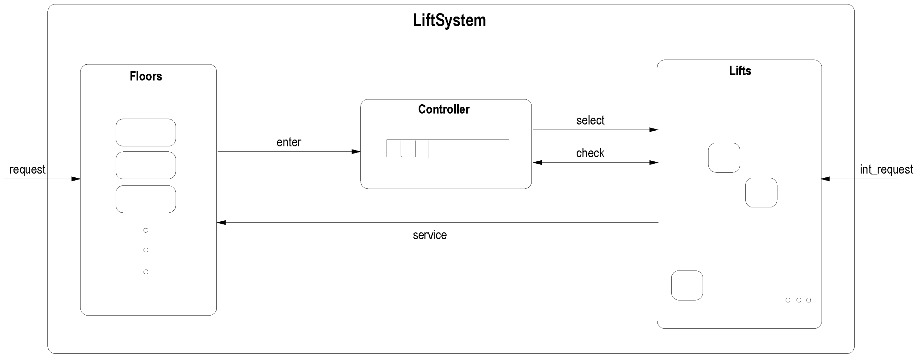

Fig. 14. Lift system communication diagram.

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.

4.1 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.

4.2 Specification Structure

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).

Fig. 14 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 request entered into its itinerary. While 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 behavior 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.

4.3 Buttons

A basic component of the lift system is the button panels on the floors or inside the lifts. Buttons have a common behavior; they can be pushed "On" by people and turned "Off" by the system.

$$ButtonStatus ::= On | Off.$$

Buttons are modeled using a simple passive class (Fig. 15), which records their current state and provides operations for turning them on and off.

4.4 The Building

Our model of the building concentrates on the behavior of the service-request panels on each floor, as depicted in Fig. 16; other floor-related issues are elided.

Button

$state : ButtonStatus$

INIT

$state = Off$

TurnOn

$\Delta(state)$

$state' = On$

TurnOff

$\Delta(state)$

$state' = Off$

Fig. 15. Button model.

4.4.1 Floor Panels

Floors may be divided into two classes, those from which it is possible to travel upward and those from which it is possible to travel downward.

$$MoveDir ::= Up | Down.$$

The *TopFloor* (Fig. 17) is a floor from which only downward travel is possible, and the *BottomFloor* (Fig. 18) is a floor from which only upward travel is possible.

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 (Fig 19).

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 behavior to the objects of the existing class. As a consequence, the MAIN process must always be redefined explicitly.

The subscript “$_{\copyright}$” (object containment [7] [13]) indicates that the button objects are contained in their corresponding floor object.

As a floor can be either top-floor, bottom-floor, or middle-floor, the general type of a floor is defined as a class-union [11].

4.4.2 The Building

The building is modeled as an aggregate of active floor objects (Fig. 20). 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, $\_|||\_$.

4.5 Lifts

A lift consists of four parts, as depicted in Fig. 21, 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 behavior 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.

4.5.1 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

$$| \; t_o : \mathbb{T} \qquad \texttt{open time}$$

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 (Fig. 22).

In the *CycleDoor* process, we make use of the CSP interrupt primitive, which allows an exception handling behavior 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 restarted 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.

4.5.2 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 (e.g., 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

[7] 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 [13].

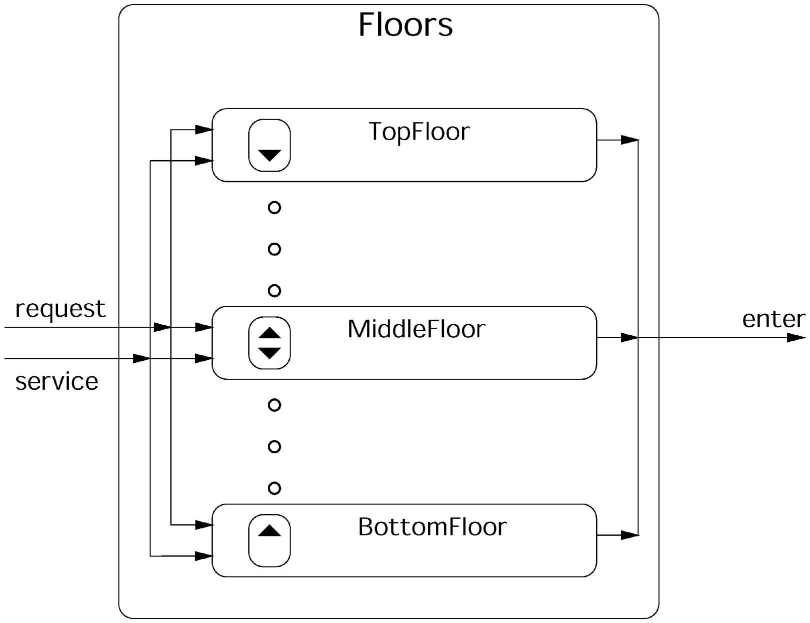

Fig. 16. External service request panels.

arrived at the destination floor. The timing properties of lift movement are described by two time constants:

- maximum time to move one floor Up or Down

  $| \; \bar{t} : \mathbb{T}$

- acceleration and braking delay

  $| \; delay : \mathbb{T}$

TopFloor

$| \; num : \mathbb{N}$

$downbutton : Button_{\copyright}$
$request, enter, service : \mathbf{chan}$

INIT
$downbutton.\mathrm{INIT}$

$$\begin{array}{l} PressDown \mathrel{\widehat{=}} \\ \quad request?(num, Down) \rightarrow \\ \quad [downbutton.state = Off] \bullet \\ \quad downbutton.TurnOn; \\ \quad enter!(num, Down) \rightarrow \mathrm{SKIP} \; \Box \\ \quad [downbutton.state = On] \bullet \mathrm{SKIP} \\ DownOff \mathrel{\widehat{=}} service?(num, Down) \\ \quad \rightarrow downbutton.TurnOff \\ \mathrm{MAIN} \mathrel{\widehat{=}} \mu \, T \bullet \\ \quad (PressDown \; \Box \; DownOff); \; T \end{array}$$

Fig. 17. Top floor model.

The maximum 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. The shaft model is captured in Fig. 23.

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 favor, it must be pointed out that from the point of view of the lift controller the

BottomFloor

$upbutton : Button_{\copyright}$
$request, enter, service : \mathbf{chan}$

$| \; num : \mathbb{N}$

INIT
$upbutton.\mathrm{INIT}$

$$\begin{array}{l} PressUp \mathrel{\widehat{=}} \\ \quad request?(num, Up) \rightarrow \\ \quad [upbutton.state = Off] \bullet \\ \quad upbutton.TurnOn; \\ \quad enter!(num, Up) \rightarrow \mathrm{SKIP} \; \Box \\ \quad [upbutton.state = On] \bullet \mathrm{SKIP} \\ UpOff \mathrel{\widehat{=}} service?(num, Up) \\ \quad \quad \rightarrow upbutton.TurnOff \\ \mathrm{MAIN} \mathrel{\widehat{=}} \mu \, B \bullet \\ \quad (PressUp \; \Box \; UpOff); \; B \end{array}$$

Fig. 18. Bottom floor model.

$$\begin{array}{l}
\underline{\mathit{MiddleFloor}} \\
\mathit{TopFloor}, \mathit{BottomFloor} \\
\hline
\mathit{downbutton} \neq \mathit{upbutton} \\
\hline
\textsc{Main} \mathrel{\widehat{=}} \mu M \bullet \\
\quad (\mathit{PressDown} \mathbin{\Box} \\
\quad \mathit{DownOff} \mathbin{\Box} \\
\quad \mathit{PressUp} \mathbin{\Box} \mathit{UpOff});\ M
\end{array}$$

Fig. 19. Middle floor model.

matters of essential interest are precisely when the movement commences and when it finishes.

4.5.3 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 (Fig. 24). A dependent (secondary) variable 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 $md?$ 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.

4.5.4 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 three modes of behavior. It is modeled in Fig. 25.

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

$$\begin{array}{|l}
t_p : \mathbb{T} \\
\quad [\texttt{wait for passenger input}].
\end{array}$$

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 lift services 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.

$$\begin{array}{l}
\underline{\mathit{Building}} \\
\mid\ \mathit{height} : \mathbb{N} \\
\hline
\mathit{floors} : \mathrm{seq}\, \mathit{Floor}_{\copyright} \\
\hline
\mathit{height} = \#\mathit{floors} \\
\forall\, i : \mathrm{dom}\, \mathit{floors} \bullet \\
\quad \mathit{floors}(i).\mathit{num} = i \\
\mathit{floors}(1) \in \mathit{BottomFloor} \\
\mathit{floors}(\mathit{height}) \in \mathit{TopFloor} \\
\forall\, i : (2 \mathinner{..} (\mathit{height} - 1)) \bullet \\
\quad \mathit{floors}(i) \in \mathit{MiddleFloor} \\
\hline
\textsc{Main} \mathrel{\widehat{=}} |||\, f : \mathrm{ran}\, \mathit{floors}
\end{array}$$

Fig. 20. Building model.

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 (Fig. 26).

4.5.5 The Bank of Lifts

Each lift consists of a door, a shaft, and a controller (Fig. 26).

The collection of all the lifts (Fig. 27) in the system is modeled as an aggregate of the individual lifts acting autonomously.

4.6 The Central Controller

The responsibility of the central controller is to dispatch floor requests to idle lifts (modeled in Fig. 28). It consists of a request queue with channels that connect the floors and the lifts. The network topology is described graphically in Fig. 14.

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 while moving in the floor required direction, the request is removed from the queue ($CheckServ$ operation).

4.7 The Lift System

The lift system consists of the floors of the building, the bank of lifts, and the central controller. The number of

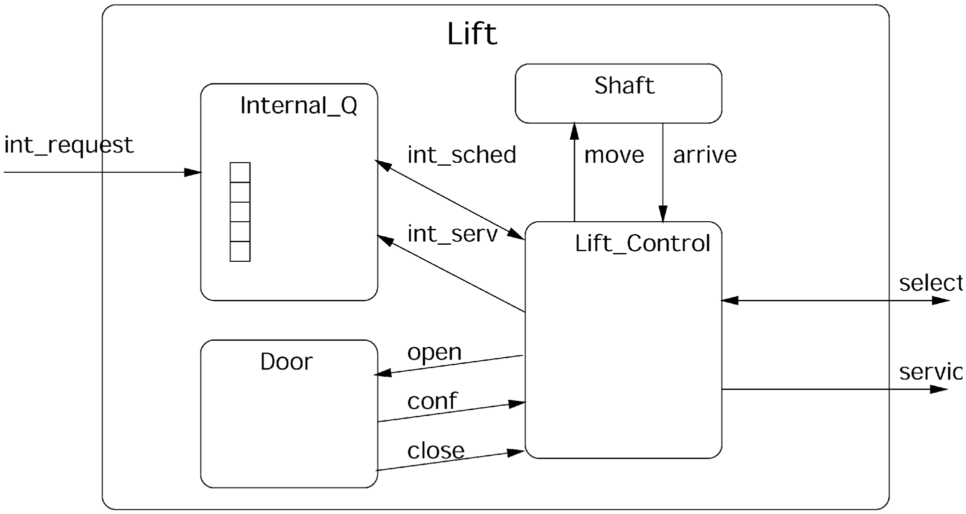

Fig. 21. Internal lift communication diagram.

floor-service buttons in each lift must be exactly the number of floors in the building. It is modeled in Fig. 29.

The lift system behavior 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.

4.8 Discussion

The application of TCOZ to the multilift 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 multilift 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 [15]. 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

Door

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

$OpenDoor \mathrel{\widehat{=}}$
$\quad servo!(self, ToOpen) \rightarrow$
$\quad sensor?(self, Opened) \rightarrow \mathrm{SKIP}$

$CloseDoor \mathrel{\widehat{=}}$
$\quad servo!(self, ToClose) \rightarrow$
$\quad sensor?(self, Closed) \rightarrow \mathrm{SKIP}$

$CycleDoor \mathrel{\widehat{=}}$
$\quad OpenDoor;\ conf \rightarrow$
$\quad (\mu\, CD \bullet \mathrm{WAIT}\, t_o;\ CloseDoor$
$\quad \nabla\ (sensor?(self, Interrupt)$
$\qquad \rightarrow OpenDoor;\ CD))$

$\mathrm{MAIN} \mathrel{\widehat{=}} \mu\, D \bullet open \rightarrow$
$\qquad CycleDoor;\ close \rightarrow D$

Fig. 22. Door model.

Shaft

$move, arrive : \mathbf{chan}$

$\mathrm{MAIN} \mathrel{\widehat{=}} \mu\, S \bullet$
$\quad [n \neq 0] \bullet move?n \rightarrow$
$\quad [0 < \delta \leq |n| * \bar{t} + delay];$
$\qquad arrive \rightarrow S$

Fig. 23. Shaft model.

$Internal\_Q$

$$height : \mathbb{N}$$

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

$NextUp$

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

$NextDown$

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

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

Fig. 24. Internal queue model.

to represent "process" behavior 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. While 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 behavior. The specification is thus, strictly speaking, not of a system for moving people between floors of a building, but 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 [47] and Object Z [15]. Apart from the ability of TCOZ to describe timing aspects not addressed by either of these specifications, such as the correct behavior of the door opening cycle, TCOZ represents an improvement in expressibility, modularity, and reusability over both existing specifications.

The CSP "lift" model [47] 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

LiftControl

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

SetFloor

$$\begin{array}{l}
\Delta(fl) \\
dest? : \mathbb{N} \\
\hline
fl' = dest?
\end{array}$$

SetDir

$$\begin{array}{l}
\Delta(md) \\
dir? : MoveDirection \\
\hline
md' = dir?
\end{array}$$

CalDir

$$\begin{array}{l}
dest? : \mathbb{N} \\
dir! : MoveDiection \\
\hline
dest? = fl \land dir! = md \lor \\
dest? > fl \land dir! = Up \lor \\
dest? < fl \land dir! = Down
\end{array}$$

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

Fig. 25. Individual lift controller.

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 synchronization operator ($\_||\_$) 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 (see p. 26 of [47]) 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 behavior 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 [15] 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, centralized 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 $LiftSystem$ (pp. 146-147 of [15]) became very complex (and lengthy) because the global ordering of synchronizations 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 synchronization 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.

$$
\begin{array}{l}
\mathit{Lift} \\
\hline
\quad \mathit{iq} : \mathit{Internal\_Q}_{\copyright} \\
\quad \mathit{lc} : \mathit{LiftControl}_{\copyright} \\
\quad s : \mathit{Shaft}_{\copyright} \\
\quad d : \mathit{Door}_{\copyright} \\
\hline
\textsc{Main} \mathrel{\widehat{=}} \parallel(\\
t1lc \xleftrightarrow{move,arrive} s; \\
t1lc \xleftrightarrow{open,close,conf} d; \\
t1lc \xleftrightarrow{int\_sched,int\_serv} iq)
\end{array}
$$

Fig. 26. Shaft model.

5 Related Work

The basis for the successful blending of the Timed CSP and Object Z notations is the duality between state transition semantics and event semantics. This has long been recognized [28], [34], [4], [6], and has undoubtedly helped shape the development of Object Z's behavioral semantics. Perhaps the most mature formalism based on this duality is Butler's [6] blending of CSP with Back's and von Smith's Action Systems [3]. 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 the usual Z precondition refinement techniques with the default behavioral semantics. The adoption of a distinct notion of state guards in TCOZ makes possible a

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

Fig. 27. Shaft model.

$$
\begin{array}{l}
\mathit{Controller} \\
\hline
\quad \mathit{reqQ} : \mathrm{seq}(\mathbb{N} \times \mathit{MoveDirection}) \\
\quad \mathit{enter}, \mathit{select}, \mathit{check}, \mathit{service} : \mathbf{chan} \\
\quad \Delta \\
\quad \mathit{req} : \mathbb{N} \leftrightarrow \mathit{MoveDirection} \\
\quad \mathit{ups}, \mathit{dns} : \mathbb{N} \times \mathbb{N} \to \mathbb{P}\,\mathbb{N} \\
\hline
\quad \mathit{req} = \mathrm{ran}\,\mathit{reqQ} \wedge \forall f_1, f_2 : \mathbb{N} \bullet \\
\qquad \mathit{ups}(f_1, f_2) = \{f : \mathbb{N} \mid f\ \underline{\mathit{req}}\ \mathit{Up} \wedge f_1 \leq f \leq f2\} \\
\qquad \mathit{dns}(f_1, f_2) = \{f : \mathbb{N} \mid f\ \underline{\mathit{req}}\ \mathit{Down} \wedge f_1 \leq f \leq f2\} \\
\\
\quad \textsc{Init} \\
\hline
\quad \mathit{reqQ} = \langle\, \rangle \\
\\
\quad \mathit{Join} \\
\hline
\quad \Delta(\mathit{reqQ}) \\
\quad \mathit{req}? : (\mathbb{N} \times \mathit{MoveDirection}) \\
\qquad \mathbf{on}\ \mathit{enter} \\
\hline
\quad \mathit{reqQ}' = \mathit{reqQ} \frown \langle \mathit{req}? \rangle \\
\\
\quad \mathit{Remove} \\
\hline
\quad \Delta(\mathit{reqQ}) \\
\quad (\mathit{dest}?, \mathit{md}?) : \mathbb{N} \times \mathit{MoveDirection} \\
\hline
\quad \mathit{reqQ}' = \mathrm{squash}(\\
\qquad \mathit{reqQ} \rhd \{(\mathit{dest}?, \mathit{md}?)\}) \\
\\
\quad \mathit{Find} \\
\hline
\quad \mathit{fl}?, d? : \mathbb{N} \\
\quad \mathit{md}? : \mathit{MoveDirection} \\
\quad \mathit{dest}! : \mathbb{N} \\
\quad \mathit{suc}! : \mathbb{B} \\
\hline
\quad \mathit{md}? = \mathit{Up} \wedge \mathit{suc}! = (\mathit{ups}(\mathit{fl}?, d?) \neq \varnothing) \wedge \\
\quad \mathit{suc}! \Rightarrow \mathit{dest}! = \mathit{min}\,\mathit{ups}(\mathit{fl}?, d?) \\
\quad \vee \\
\quad \mathit{md}? = \mathit{Down} \wedge \\
\quad \mathit{suc}! = (\mathit{dns}(\mathit{fl}?, d?) \neq \varnothing) \wedge \\
\quad \mathit{suc}! \Rightarrow \mathit{dest}! = \mathit{min}\,\mathit{dns}(\mathit{fl}?, d?) \\
\\
\quad \mathit{Dispatch} \mathrel{\widehat{=}} \\
\qquad [(\mathit{dest}, \mathit{md}) : \mathbb{N} \times \mathit{MoveDirection} \mid \mathit{reqQ} \neq \langle\, \rangle \wedge (\mathit{dest}, \mathit{md}) = \mathrm{head}\,\mathit{reqQ}] \bullet \\
\qquad\quad \mathit{select}!(\mathit{dest}, \mathit{md}) \to \mathit{Remove} \\
\quad \mathit{CheckServ} \mathrel{\widehat{=}} \\
\qquad [\mathit{fl}, d : \mathbb{N};\ \mathit{md} : \mathit{MoveDirection}] \bullet \mathit{check}?(\mathit{fl}, d, \mathit{md}) \to \mathit{Find}; \\
\qquad\quad [\mathit{suc}] \bullet \mathit{check}!\mathit{dest} \to \mathit{Remove}\ \Box\ [\neg\, \mathit{suc}] \bullet \mathit{check} \to \textsc{Skip} \\
\quad \textsc{Main} \mathrel{\widehat{=}} \mu\, C \bullet (\mathit{Join}\ \Box\ \mathit{Dispatch}\ \Box\ \mathit{CheckServ});\ C
\end{array}
$$

Fig. 28. Lift system controller mode.

LiftSystem

$bd : Building$
$ls : Lifts$
$contr : Controller$

$\forall\, l : ls.lifts \bullet$
$\quad l.iq.height = bd.height$

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

Fig. 29. Final lift system model.

full blending of the Z-style algorithm and the CSP-style process refinement.

The notion of blending the untimed CSP and Object Z has been proposed independently by Fischer [21] and Smith [51]. 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, while 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 thorough 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 precondition 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. 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 behavior 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 widely recognized. 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 [32], ESTEREL [5], RAISE [25]) and those that adopt a transition system style (UNITY [7], Action Systems [3], TLA [35]). We consider two real-time specific languages, E-LOTOS [31] and ASTRAL [8], as being representative of their respective classes and having similar aims to TCOZ.

The LOTOS specification language [32] is very similar in approach to TCOZ, blending CSP-like process primitives with a declarative-style data-specification language. E-LOTOS [31] is a recently developed real-time extension to LOTOS. The process and real-time primitives of E-LOTOS are influenced by Timed-CSP, therefore these aspects of E-LOTOS are similar to TCOZ. The major differences lie in the data-modeling and structuring aspects of the two formalisms. The data modeling language of E-LOTOS is an algebraic/functional hybrid, while 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).

ASTRAL is developed based on the ideas of RT-ASLAN [2] and TRIO [26]. ASTRAL also has a module construct that encapsulates the variables and transactions. The ASTRAL module has the two dimensions of reuse, importing and type instantiation. Therefore, the ASTRAL module is very close to the TCOZ class construct except that the names of instances of a module are modeled explicitly, while the object identity is implicitly included in the TCOZ class semantics (a class is a collection of object identities). In ASTRAL, object composition is generally modeled by the (programming-language-flavored) list type construct, *array ...of...*. In TCOZ, object composition may have various (mathematics-flavored) abstract forms, i.e., collections "$\mathbb{P}$..." and list "seq...". The timing aspects are similar to the TLA approach [1] and the Timed Object Z approach [11]. That is, a global clock *NOW* is introduced and system environments are modeled explicitly. Therefore, sequential real-time systems can be captured well by ASTRAL, while truly concurrent active systems are difficult to describe in ASTRAL.

6 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 flavor 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, timing, and communication concerns in the CSP style. The notion of active and passive objects are clearly distinguished in the TCOZ model. This powerful modeling combination, TCOZ, has been successfully applied to a comprehensive case study on specifying a real-time multilift system. In comparison to the CSP model [47] and the standard Object Z model [15] of the lift system, the TCOZ model not only describes the complex system state and behavior within a cleaner and less coupled structure, but also captures 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 [37], [41], which allow convincing descriptions of continuously varying real-world observables. TCOZ preserves, in a 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. These benefits might include: the full application of the Z schema calculus, structured design, and refinement techniques; the application of Timed CSP process equivalence and refinement techniques, especially the Timed CSP model hierarchy for moving between timed and untimed models; the extension of existing Z, Object Z, and CSP tools and model checkers. For example, we are currently planning a project to construct a parser/type-checker for TCOZ based on the Object Z parser of Johnson [33].

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [38]. 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 [46]; it is hoped that this might also form a valuable addition to the TCOZ notation.

Appendix A
TCOZ Concrete Syntax

The TCOZ syntax is based on the Object-Z syntax [16], [18].

A.1 Notation

The syntax is described in an extended BNF with the following metasymbols:

$::=$	produces
$\vert$	altnerative
$[x]$	optional x
$\{x\}$	zero or more $x's$
$\{x\}_1$	one or more $x's$.

Nonterminals ending in $List$ have productions according to:

$$xList ::= x\{,x\}.$$

Nonterminal names are typically compound and abbreviated, each part commencing with an uppercase letter, e.g., $OpExpDef$. Abbreviations are listed below.

Terminal symbols are shown directly as they appear in TCOZ. Metasymbols are larger than terminal symbols of the same shape as seen by:

$$::=\ {\scriptstyle ::=},\ \Big|\ {\scriptstyle |},\ \Big[\,\Big]\ {\scriptstyle [\,]},\ \text{and}\ \Big\{\,\Big\}\ {\scriptstyle \{\,\}}.$$

A.2 Abbreviations

The following abbreviations are used in the productions listed in Section A.3.

Abb	Abbreviation	Gen	Generic
Ax	Axiomatic	Inv	Invariant
Bool	Boolean	Op	Operation
Chan	Channel	Pred	Predicate
Dec	Declaration	Prim	Primary
Def	Definition	Sec	Secondary
Des	Designator	Sep	Separator
Exp	Expression	Var	Variable

A.3 Productions

The order of presentation of productions is top down with definitions for nonterminals appearing after their last application, except for recursive definitions.

Specification ::=
  Def $\{$*Sep Def*$\}$

Def ::=
  ClassDef | *NonClassDef*

ClassDef ::=

```
┌─ ClassHeading ─────
│ [Visibility]
│ [Inheritance]
│ [LocalDefs]
│ [StateSchema]
│ [InitialSchema]
│ [Operations]
└────────────────────
```

ClassHeading ::=
  ClassName [*GenFormals*]

Visibility ::= $\upharpoonright$ (*NameList*)

NameList ::=
  Name $\{$, *Name*$\}$

Inheritance ::=
  ClassDes $\{$*Sep ClassDes*$\}$

LocalDefs ::=
  NonClassDef $\{$*Sep NonClassDef*$\}$

NonClassDef ::=
  GivenTypeDef | *FreeTypeDef* |
  AbbDef | *AxDef* | *GenDef*

GivenTypeDef ::= [*NameList*]

FreeTypeDef ::=
  Name ::= *Branch* $\{$ | *Branch*$\}$

Branch ::=
  Name [$\langle\langle$*Exp*$\rangle\rangle$]

AbbDef ::=
  Name [*GenFormals*] == *Exp*

AxDef ::=

```
│ Decs
├──────────
│ [Pred]
```

GenDef ::=

```
┌═[GenFormals] ═════
│ Decs
├──────────────
│ [Pred]
└═══════════════════
```

GenFormals ::= [*NameList*]

StateSchema ::=

```
┌──────────────────
│ [ChanDecs]
│ [PrimVarDecs]
│ [Δ
│  SecVarDecs]
├──────────────
│ [ClassInv]
└──────────────────
```

ChanDecs ::=
  ChanDec $\{$; *ChanDec*$\}$

ChanDec ::= *NameList* : **chan**

PrimVarDecs ::= *Decs*

SecVarDecs ::= *Decs*

ClassInv ::= *Pred*

InitialSchema ::=

```
┌─ INIT ─────────────
│ Pred
└────────────────────
```

Operations ::=
  OpDef $\{$*Sep OpDef*$\}$

OpDef ::=
  OpSchemaDef | *OpExpDef*

OpSchemaDef ::=

```
┌─ OpName ──────────
│ [Deltalist]
│ [Decs]
├──────────────
│ [Pred]
└───────────────────
```

OpExpDef ::=
  OpName $\widehat{=}$ *OpExp*

OpExp ::=
  $\sqcap$ *SchemaText* • *OpExp*
  | $\Box$ *SchemaText* • *OpExp*
  | $|||$ *SchemaText* • *OpExp*

$\mid \mathit{NetworkTopology}$

$\mid [\mathit{SchemaText}] \bullet \mathit{OpExp}$

– – – – – – – – – – –

$\mid \mathit{OpExp};\ \mathit{OpExp}$ L

– – – – – – – – – – –

$\mid \mathit{OpExp} \Box \mathit{OpExp}$ L

$\mid \mathit{OpExp} \sqcap \mathit{OpExp}$ L

– – – – – – – – – – –

$\mid \mathit{OpExp} \parallel\mid \mathit{OpExp}$ L

$\mid \mathit{OpExp} \mid[\, \mathit{Exp} \,]\mid \mathit{OpExp}$ L

$\mid \mathit{OpExp} \parallel \mathit{OpExp}$ L

$\mid \mathit{OpExp} \rhd \{\mathit{Exp}\}\ \mathit{OpExp}$ L

$\mid \mathit{OpExp} \swarrow \{\mathit{Event}\}\ \mathit{OpExp}$ L

$\mid \mathit{OpExp} \nabla \mathit{OpExp}$ L

$\mid \mathit{OpExp} \bullet \textsc{Deadline}\, \mathit{Exp}$

$\mid \mathit{OpExp} \bullet \textsc{WaitUntil}\, \mathit{Exp}$

$\mid \textsc{Wait}\, \mathit{Exp}$

$\mid \mathit{Event}\big[@\mathit{Exp}\big] \to \mathit{OpExp}$

$\mid \mu\, \mathit{OpName} \bullet \mathit{OpExp}$

– – – – – – – – – – –

$\mid \mathit{OpExp}\ \mathit{Renaming}$

$\mid \mathit{OpExp} \setminus (\mathit{NameList})$

– – – – – – – – – – –

$\mid \mathit{OpSchemaExp}$

$\mid (\mathit{OpExp})$

$\mid \big[\mathit{Exp}\,.\big]\ \mathit{OpName}$

$\mid \perp \mid \textsc{Stop} \mid \textsc{Skip}$

$\mathit{OpSchemaExp} ::=$

$\bigwedge \mathit{SchemaText} \bullet \mathit{OpExp}$

$\mid \bigvee \mathit{SchemaText} \bullet \mathit{OpExp}$

– – – – – – – – – – –

$\mid \mathit{OpExp} \wedge \mathit{OpExp}$ L

– – – – – – – – – – –

$\mid \mathit{OpExp} \vee \mathit{OpExp}$ L

– – – – – – – – – – –

$\mid [\ \big[\mathit{Deltalist}\big]\ \big[\mathit{Decs}\big]\ \big[\mid \mathit{Pred}\big]\ ]$

$\mid \big[\mathit{Exp}\,.\big]\ \mathit{OpName}$

$\mid (\mathit{OpExp})$

$\mathit{NetworkTopology} ::=$

$\|(\mathit{Connection} \big\{;\ \mathit{Connection}\big\}\ )$

$\mathit{Connection} ::=$

$\mathit{NameList} \xleftrightarrow{\mathit{NameList}} \mathit{NameList}$

$\mathit{Event} ::= \mathit{Name}\big[.\mathit{Exp}\big]$

Productions are in equal-precedence groups (separated by – – – –), and the precedence of groups increases down the page. The "L/R" indication determines left or right associativity of binary operators.

$\mathit{OpName} ::= \mathit{Name} \mid \textsc{Main}$

$\mathit{Deltalist} ::= \Delta(\mathit{NameList})$

$\mathit{SchemaText} ::=$

$\mathit{Decs}\ \big[\mid \mathit{Pred}\big]$

$\mathit{Decs} ::=$

$\mathit{Dec}\ \big\{;\ \mathit{Dec}\big\}$

$\mathit{Dec} ::= \mathit{NameList} : \mathit{Exp}$

$\mathit{Pred} ::=$

$\forall\, \mathit{SchemaText} \bullet \mathit{Pred}$

$\mid \exists\, \mathit{SchemaText} \bullet \mathit{Pred}$

$\mid \exists_1\, \mathit{SchemaText} \bullet \mathit{Pred}$

$\mid \mathrm{let}\ \mathit{LetDefs} \bullet \mathit{Pred}$

– – – – – – – – – – –

$\mid \mathit{Pred} \Leftrightarrow \mathit{Pred}$ L

– – – – – – – – – – –

| *Pred* ⇒ *Pred* R

– – – – – – – – – – –

| *Pred* ∨ *Pred* L

– – – – – – – – – – –

| *Pred* ∧ *Pred* L

– – – – – – – – – – –

| ¬ *Pred*

| *Name* . INIT

| true | false

| *BoolExp*

| (*Pred*)

The nonterminal *Deltalist* cannot be written *DeltaList*, as the construct represented does not conform to the *List* convention.

Iff (⟺), disjunction (∨) and conjunction (∧) are associative. "R" indicates right-to-left association.

BoolExp ::= *Exp*

Exp ::=

μ *SchemaText* [• *Exp*]

| λ *SchemaText* • *Exp*

| let *LetDefs* • *Exp*

| if *Pred* then *Exp* else *Exp*

– – – – – – – – – – –

| *Exp* $\{ \times Exp \}_1$

– – – – – – – – – – –

| $\mathbb{P}$ *Exp*

– – – – – – – – – – –

| *Exp Infix Exp* L

– – – – – – – – – – –

| *Exp Exp* L

– – – – – – – – – – –

| *Prefix Exp*

– – – – – – – – – – –

| *Exp Postfix*

– – – – – – – – – – –

| *ClassHierarchy*

| *Exp* . *Name*

| *Name* [*GenActuals*]

| { [*ExpList*] }

| {*SchemaText* [• *Exp*] }

| ∅

| (*Exp* $\{, Exp\}_1$)

| ⟨ [*ExpList*] ⟩

| *Number*

| self

| (*Exp*)

LetDefs ::=

LetDef {; *LetDef*}

LetDef ::= *Name* == *Exp*

ClassHierarchy ::=

[↓] *ClassDes*

ClassDes ::=

ClassName [*GenActuals*] [*Renaming*]

ClassName ::= *Name*

GenActuals ::= [*ExpList*]

Renaming ::= [*RenameList*]

Rename ::= *Name*/*Name*

The syntax does not define the details of the productions for

Infix, *Prefix*, *Postfix*, *Number*, *Name*, *Sep*.

Elision of end-of-line semicolons, end-of-line conjunction symbols, and "such that" bars in 2D structures without predicates is not shown in the syntax.

ACKNOWLEDGMENTS

We would like to thank John Colton, Ian Hayes, Keith Gallagher, Neale Fulton, and the anonymous referees for many useful comments. This work has been supported in part by the DSTO/CSIRO Fellowship program.

References

[1] M. Abadi and L. Lamport, “An Old-Fashioned Recipe for Real Time,” *ACM Trans. Programming Languages,* vol. 15, no. 5, pp. 1,543-1571, 1991.

[2] B. Auernheimer and R.A. Kemmerer, “Rt-aslan: A Specification Language for Real-Time Systems,” *IEEE Trans. Software Eng.,* vol. 12, no. 9, Sept. 1986.

[3] R.J.R. Back and J. von Wright, “Refinement Calculus, Part II: Parallel and Reactive Programs,” *Stepwise Refinement.* J.W. de Bakker, W.P. de Roever, and G. Rozenberg, eds., Springer Verlag, 1990.

[4] M. Benjamin, “A Message Passing System: An Example of Combining CSP and Z,” *Z User Workshop: Proc. Fourth Ann. Z User Meeting,* pp. 221–228, Dec. 1989

[5] G. Berry and G. Gonthier, “The Esterel Synchronous Programming Language: Design, Semantics, Implementation,” *Science of Computer Programming,* vol. 19, no. 2, pp. 87–152, Nov. 1992.

[6] M.J. Butler, “A CSP Approach to Action Systems,” PhD thesis, Wolfson College, Oxford University, 1992.

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

[8] A. Coen-Porisini, C. Ghezzi, and R. Kemmerer, “Specification of Real-Time Systems Using ASTRAL,” Technical Report 96-30, Computer Science Dept., Univ. of California, Santa Barbara, Jan. 1997.

[9] H.C. Cunningham, V.R. Shan, and S. Shen, “Devising a Formal Specification for an Elevator Controller,” Technical Report TR 94-10, Department of Computer and Information Science, Univ. of Mississippi, 1994.

[10] J.S. Dong, “Living with Free Type and Class Union,” *Proc. 1995 Asia-Pacific Software Eng. Conf. (APSEC '95),* pp. 304–312, Dec. 1995.

[11] J.S. Dong, J. Colton, and L. Zucconi, “A Formal Object Approach to Real-Time Specification,” *Proc. Third Asia-Pacific Software Eng. Conf. (APSEC '96),* Dec. 1996.

[12] J. S. Dong and R. Duke, “The Geometry of Object Containment,” *Object-Oriented Systems,* vol. 2, no. 1, pp. 41–63, Mar. 1995.

[13] J.S. Dong and B.P. Mahony, “Active Objects in TCOZ,” *Proc. Second Int'l Conf. Formal Eng. Methods (ICFEM '98),* pp. 16–25, 1998.

[14] J.S. Dong, G. Rose, and R. Duke, “The Role of Secondary Attributes in Formal Object Modeling” *Proc. First IEEE Int'l Conf. Eng. Complex Computer Systems (ICECCS '95),* pp. 31–38, Nov. 1995.

[15] J.S. Dong, L. Zucconi, and R. Duke, “Specifying Parallel and Distributed Systems in Object Z,” *Second Int'l Workshop Software Eng. for Parallel and Distributed Systems,* pp. 140–149, 1997.

[16] R. Duke, P. King, G. Rose, and G. Smith, “The Object Z Specification Language: Version 1,” Technical Report 91-1, Software Verification Research Centre, Australia, 1991.

[17] R. Duke and G. Rose, “Modeling Object Identity,” *Proc. 16th Australian Computer Science Conf. (ACSC-16),* pp. 93–100, Feb. 1993.

[18] R. Duke and G. Rose, *Formal Object-Oriented Specification.* Academic Press, 1998.

[19] R. Duke, G. Rose, and G. Smith, “Object Z: A Specification Language Advocated for the Description of Standards,” *Computer Standards and Interfaces,* vol. 17, no. 511–533, 1995.

[20] C. Fidge, P. Kearney, and M. Utting, “A Formal Method for Building Concurrent Real-Time Software,” *IEEE Software,* vol. 14, no. 2, 1997.

[21] C. Fischer, “CSP-OZ: A Combination of Object Z and CSP,” *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97),* vol. 2, pp. 423–438, 1997.

[22] I. R. Forman, “Design by Decomposition of Multiparty Interactions in Raddle87,” *Proc. Fifth IEEE Int'l Workshop Software Specification and Design (IWSSD'89),* pp. 2–10, 1989.

[23] A.J. Galloway, “Integrated Formal Methods with Richer Methodological Profiles for the Development of Multiperspective Systems,” PhD thesis, Univ. of Teesside, School of Computing and Mathematics, Aug. 1996.

[24] A.J. Galloway and W.J. Stoddart, “An Operational Semantics for ZCCS,” *Proc. IEEE Int'l Conf. Formal Eng. Methods (ICFEM '97),* pp. 272–282, Nov. 1997.

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

[26] C. Ghezzi, D. Mandrioli, and A. Morzenti, “Trio: A Logic Language for Executable Specifications of Real-Time System,” *J. Systems and Software,* June 1990.

[27] I.J. Hayes and B.P. Mahony, “Using Units of Measurement in Formal Specifications,” *Formal Aspects of Computing* vol. 7, no. 3, 1995.

[28] J. He, “Process Simulation and Refinement,” *Formal Aspects of Computing,* vol. 1, no. 3, pp. 229–241, 1989.

[29] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[30] *Units of Measurement: Handbook on Int'l Standards for Units of Measurement.* Geneva: Int'l Organization for Standardization, 1979.

[31] “ISO,” SC21/WG7 Working Draft on Enhancements to LOTOS, ISO Working Group 7, Dec. 1997.

[32] “ISO 8807,” *LOTOS–A Formal Description Technique Based on the Temporal Ordering of Observational Behavior,* 1989.

[33] W. Johnston, “A Type Checker for Object Z,” Technical Report 96-24, Software Verification Research Centre, School of Information Technology, Univ. of Queensland, Brisbane, Australia, July 1996.

[34] M.B. Josephs, “A State-Based Approach to Communicating Processes,” *Distributed Computing,* vol. 3, pp. 9–18, 1988.

[35] L. Lamport, “The Temporal Logic of Actions,” “ACM Trans. Programming Languages and Systems,” vol. 16, pp. 872–923, 1994.

[36] L. Lamport and L.C. Paulson, “Should Your Specification Language Be Typed?” Technical Report 147, Systems Research Center, 1997.

[37] B.P. Mahony and J.S. Dong, “Overview of the Semantics of TCOZ,” *Proc. IFM '99: Integrated Formal Methods,* pp. 66–85, June 1999.

[38] B.P. Mahony, “Networks of Predicate Transformers,” Technical Report 95-05, Software Verification Research Centre, Department of Computer Science, Univ. of Queensland, St. Lucia, Australia, Feb. 1995.

[39] B.P. Mahony and J. S. Dong, “Blending Object Z and Timed CSP: An Introduction to TCOZ,” *Proc. 20th Int'l Conf. Software Eng. (ICSE'98),* Apr. 1998.

[40] B.P. Mahony and J.S. Dong, “Network Topology and a Case Study in TCOZ,” *Proc. ZUM '98 11th Int'l Conf. Z Users,,* vol. 1,493, Sept. 1998.

[41] B.P. Mahony and I.J. Hayes, “A Case Study in Timed Refinement: A Mine Pump,” *IEEE Trans. Software Eng.,* vol. 18, no. 9, pp. 817–826, Sept. 1992.

[42] M. Mislove, A. Roscoe, and S. Schneider, “Fixed Points without Completeness,” *Theoretical Computer Science,* vol. 138, pp. 273–314, 1995.

[43] C.C. Morgan, *Programming from Specifications,* second ed. Prentice Hall, 1994.

[44] A.W. Roscoe, *Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[45] 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.

[46] S. Schneider and J. Davies, “A Brief History of Timed CSP,” *Theoretical Computer Science,* vol. 138, 1995.

[47] M.D. Schwartz and N.M. Delisle, “Specifying a Lift Control System with CSP,” *Proc. Fourth IEEE Int'l Workshop Software Specification and Design (IWSSD '87),* pp. 21–27, Apr. 1987.

[48] G. Smith, “A Fully Abstract Semantics of Classes for Object Z,” *Formal Aspects of Computing,* vol. 7, no. 3, pp. 289–313, 1995.

[49] G. Smith, “A Semantic Integration of Object Z and CSP for the Specification of Concurrent Systems,” *Proc. FME '97: Industrial Benefit of Formal Methods,* Sept. 1997.

[50] K. Taguchi and K. Araki, “The State-Based CCS Semantics for Concurrent Z Specification,” *Proc. IEEE Int'l Conf. Formal Eng. Methods (ICFEM '97),* pp. 283–292, Nov. 1997.

[51] J.C.P. Woodcock, S. King, and I.H. Sorensen, “Mathematics for Specification and Design: The Problem with Lifts,” *Proc. Fourth IEEE Int'l Workshop Software Specification and Design (IWSSD '87),* pp. 265–268, 1987.

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

Brendan Mahony received the BSc degree from the Australian National University in 1984 and the BA degree with first class honors from the University of Queensland in 1987. In 1991, he was awarded a PhD at the University of Queensland, after studying techniques for the specification and design of real-time systems. He continued these studies as a research fellow at the University of Queensland. Since 1995, he has been employed by the Defence Science and Technology Organisation, researching the application of state machine formalisms to the analysis of safety and security critical systems. In 1997-1998, he spent a year with CSIRO's Mathematical and Information Sciences in Canberra, as the CSIRO/DSTO fellow.

Jin Song Dong received the BInfTech degree (with first class honors) and the PhD degree from the University of Queensland in 1992 and 1995, respectively. From 1995-1998, he was a research scientist and then a senior research scientist at the CSIRO Mathematical and Information Sciences in Australia. In 1998, he joined the faculty of the Computer Science Department of the National University of Singapore. His research interests include formal methods, real-time specification, object orientation, programming language semantics, safety critical systems, and open distributed systems. He is member of the IEEE. His recent papers can be found at http://www.comp.nus.edu.sg/~dongjs/.