Blending Object-Z and Timed CSP: An introduction to TCOZ

Brendan Mahony

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

Jin Song Dong

Mathematical and Information Sciences
C'wealth Science & Ind'l Research Organisation
jinsong.dong@cmis.csiro.au

ABSTRACT

Object-Z is an extension to the Z language designed to facilitate specification in an object-oriented style. It is an excellent tool for modeling data and algorithms, but its object semantics are single threaded and operations are atomic. Therefore, it is difficult to use Object-Z to capture the behaviour of concurrent real-time reactive systems. On the other hand, Timed CSP is good at modeling real-time concurrent behaviour, but has little support for modeling the state of a complex system. This paper introduces a blending of Object-Z and Timed CSP, known as TCOZ. The blended notation is particularly suited for specifying complex systems whose components have their own thread of control.

1 Introduction

A complex system often has an intricate system state and a control structure involving concurrency and real-time interaction.

Object-Z [4] is a variant of the Z formal specification language designed to facilitate the modular design of complex data and algorithms through the application of object oriented techniques. The main aim of Object-Z is to improve the clarity of large specifications through enhanced structuring. Object-Z has a single threaded semantics and operations are atomic. There is no notion of duration associated with operations, just the abstract notion of termination. Therefore, it is difficult to use Object-Z to model concurrent real-time reactive systems.

Timed CSP [15] is an extension of Hoare's Communicating Sequential Processes (CSP) process algebra notation. It builds on CSP's strengths in modeling process control issues, such as concurrency and synchronisation, by adding primitives for modeling real-time. 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 introduces a blending of Object-Z and Timed CSP, called Timed Communicating Object-Z (TCOZ). TCOZ builds on the respective strengths of both notations in order to provide a single elegant notation for modeling both state and process aspects of complex systems. The notion of blending Object-Z with CSP has been suggested independently by Fischer [6] and Smith [16]. TCOZ is novel in that it includes timing primitives and supports the modeling of true multi-threaded concurrency.

To facilitate our discussion the common example of a first-in/first-out queue is used. We first discuss the queue without and then with timing constraints. This example is used through out the paper to illustrate the differences between and to demonstrate the advantages and disadvantages of the Object-Z, Timed CSP, and TCOZ notations respectively. Modeling the same system in each of the notations allows the reader to concentrate more on understanding the relative merits of the notations and less on understanding the subtlties of systems being modeled. The main body of the paper thus consists of three sections of similar structure. In Section 2 the Object-Z notation is briefly introduced and the queue example used to demonstrate its weakness in modeling timing and concurrency issues. In Section 3 the Timed CSP notation is briefly introduced and the queue example used to demonstrate its weakness in the modeling of complex data and algorithms. In Section 4, the blended notation, TCOZ, is introduced and the queue example used to show how it uses the strengths of the individual notations to address their respective weaknesses. Finally, the multi-threaded capabilities of TCOZ are illustrated by a buffered comsumer/producer process example.

This paper is aimed at readers with some familiarity with mathematical approaches to the modeling of data, algorithms, and process control issues, though it should not be neccessary to have any detailed knowledge of the Object-Z and Timed CSP notations.

2 Object-Z

Object-Z is an object oriented extension of the Z formal specification language. It improves the clarity of large specifications through enhanced structuring.

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

2.1 A generic queue example

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

$Queue[X]$

$$items : \operatorname{seq} X$$
$$\Delta$$
$$h : X$$
$$\overline{\quad items \neq \langle\,\rangle \Rightarrow h = head(items) \quad}$$

$\textit{INIT}$

$$items = \langle\,\rangle$$

$Join$

$$\Delta(items)$$
$$item? : X$$
$$\overline{\quad items' = items \frown \langle item? \rangle \quad}$$

$Leave$

$$\Delta(items)$$
$$item! : X$$
$$\overline{\quad items \neq \langle\,\rangle \quad}$$
$$item! = h \wedge items' = tail(items)$$

The state schema is nameless and contains declarations (the attributes) above the short dividing line and a predicate (class invariant) below the line.

In this example, it has one (true) attribute *items* denoting a sequence of elements of the generic type X. The Z toolkit supports sequences with the following mechanisms: the set of finite sequences over type X is written $\operatorname{seq} X$; sequence constants are written as list of constants enclosed in angled brackets, eg. $\langle 2, 3, 3, 1 \rangle$ is an integer sequence, $\langle\,\rangle$ is the empty sequence; sequences may be joined using *concatenation*, eg. $s_1 \frown s_2$; the first element of a sequence is the *head* and rest of the sequence is its *tail*.

The queue state also has *secondary* attribute h representing the head element of the queue, when it is non-empty. Syntactically, the declarations of secondary attributes appear below the Δ separator placed in the declaration section of the state schema. For a detailed discussion on secondary attributes see [3].

The initial state schema is distinguished by the keyword $\textit{INIT}$. The state schema is implicitly included in the initial state schema. In this example, an initialised queue contains no elements of X (i.e. *items* is the empty sequence).

The remaining two schemas are operation schemas. Operation schemas have a Δ-list of those attributes whose values may change. By convention, no Δ-list means no attribute changes value. Every operation schema implicitly includes the state schema in un-primed form (the state before the operation) and primed form (the state after the operation). Implicitly associated with each schema is a *precondition* which describes the initial states for which there exists some final state satisfying the schema predicate.

In this example, operation *Join* appends a given input *item*? to the existing sequence of items. Operation *Leave* outputs a value *item*! defined as the head of sequence *items* and reduces *items* to the tail of its original value. The precondition of *Join*, written pre *Join*, is `true` and pre *Leave* is $items \neq \langle\,\rangle$.

Note that the values of the secondary attributes are always subject to change. This is achieved by implicit inclusion of secondary attributes in the Δ-list of every operation. In this case, the value of h is always subject to change whenever operation *Join* or *Leave* is invoked.

Transition system interpretation

The standard behavioural interpretation of Object-Z objects is as transition systems [17]. A behaviour of a transition system consists of a series of state transitions each effected by one of the class operations. The *Queue* object starts with *items* empty then evolves by successively performing either *Join* or *Leave* operations. This is sometimes expression semi-formally by an equation such as

$$Qbeh \mathrel{\widehat{=}} (Join \mathbin{\Box} Leave);\ Qbeh.$$

Here *Qbeh* represents the behaviour of the *Queue*, $(\_ \mathbin{\Box} \_)$ repesents a choice between operations, and $(\_;\ \_)$ represents sequential composition[1]. Operations in Object-Z are atomic, only one may occur at each transition, and there is no notion of time or duration. It is difficult to use the standard Object-Z semantics to model a queue for which the Leave and Join operations have duration, let alone one for which those durations may overlap.

A crucial point is that the operations enabled at each point are those whose preconditions are satisfied by the current state. Thus if the *items* list is empty the *Leave* operation may not occur. This entwining of behavioural

[1] Note the clear influence of CSP.

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. It is important to note that this use of preconditions to control the sequencing of transitions is incompatable with standard algorithmic refinement techniques which would allow arbitrary weakening of preconditions, thus playing havoc with the delicate interplay of preconditions in the original specification.

2.2 Timed Object-Z

Suppose that the queue has the following timing properties. Firstly, that it takes a small but non-zero time (t_l and t_j respectively) to update the internal state during a *Leave* or *Join* operation. Secondly, that each element of the queue becomes *stale* if it is not passed on within C time units of being added to the queue and that stale elements should never be passed on.

Such issues cannot be adequately addressed in the standard interpretation of Object-Z, extended conventions for modeling time and environment are required. One such approach is the Timed Object-Z notation [2]. Here, various Z-based approaches for specifying real-time requirement [5, 12] are integrated into the Object-Z notation.

Firstly, let

$$\mathbb{T} \mathrel{\widehat{=}} \mathbb{R}$$

represent the (real) time domain.

Timed Object-Z extends Object-Z in two ways. Firstly, environmental factors are modeled as functions of time and are included in the system state. In the queue with timeouts the environment is modeled by functions env_j and env_l representing the willingness of the environment to participate in adding/removing a particular item to/from the queue respectively. The second extension is to include a global real-time clock, conventially repesented by a distinguished state attribute *now*. The clock may only be updated during an operation and the next operation must start immediately the last is finished.

Inheritance in Object-Z is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes. The timed queue is defined by inheriting the simple queue with re-definitions of the operations *Join* and *Leave* in order to reflect interactions with the environment and the passage of time. Since some operation must always be in action, a new operation *Drop* is introduced to timeout queue items and progress time to the next interesting action. The length of the *Drop* operation is determined by a secondary attribute *idle*, which computes the amount of time until the environment is ready to allow a *Join* or *Leave* operation.

TimedQueue$[X]$

$$Queue[X \times \mathbb{T}][\textbf{redef: } Join, Leave]$$

$$\begin{array}{ll}
env_j, env_l : \mathbb{T} \nrightarrow X & \\
\Delta & \\
now : \mathbb{T} & \text{[current time]} \\
idle : \mathbb{T} & \text{[time before next Join/Leave]} \\
\hline
\textbf{let } rm = \{ t : \mathrm{dom}(env_j \cup env_l) \mid t > now \} \bullet & \\
\quad rm \neq \varnothing \Rightarrow idle = min(rm) - now &
\end{array}$$

Join

$$\begin{array}{l}
\Delta(items) \\
item? : X \\
\hline
now \in \mathrm{dom}\, env_j \wedge env_j(now) = item? \\
now' - now = t_j \\
items' = ds(t_j, items) \frown \langle (item?, C) \rangle
\end{array}$$

Leave

$$\begin{array}{l}
\Delta(items) \\
item! : X \\
\hline
items \neq \langle\, \rangle \\
now \in \mathrm{dom}\, env_l \wedge env_l(now) = item! \\
now' - now = t_l \wedge item! = first(h) \\
items' = ds(t_l, tail(items))
\end{array}$$

Drop

$$\begin{array}{l}
\Delta(items) \\
\hline
now \notin \mathrm{dom}\, env_j \\
items \neq \varnothing \Rightarrow now \notin env_l \\
now' - now = idle \\
items' = ds(idle, items)
\end{array}$$

The Z toolkit supports tuples or vectors with the following mechanisms: the cross product of X and Y is written $X \times Y$, elements of which consist of pairs (x, y); the first element of tuple t is $first(t)$ and the second is $second(t)$. The generic function ds (drop stale) is defined as

$$
\begin{array}{l}
[X] \\
\hline
ds : (\mathbb{T} \times \mathrm{seq}(X \times \mathbb{T})) \rightarrow \mathrm{seq}(X \times \mathbb{T}) \\
\hline
\forall\, t, s : \mathbb{T};\ h : X;\ tl : \mathrm{seq}(X \times \mathbb{T}) \bullet \\
\quad ds(t, \langle\rangle) = \langle\rangle \\
\quad t < s \Rightarrow \\
\qquad ds(t, \langle (h, s) \frown tl \rangle) = ds(t, tl) \\
\quad t \geq s \Rightarrow \\
\qquad ds(t, \langle (h, s) \frown tl \rangle) = \langle (h, s - t) \rangle \frown ds(t, tl)
\end{array}
$$

for example $ds(2, \langle (a, 1), (b, 3), (c, 7) \rangle) = \langle (b, 1), (c, 5) \rangle$.

In this model, changes to data (*items*) are well captured. However, Timed Object-Z requires interactions with the environment and the progress of time must be micro-managed in a most intrusive manner. A proliferation of additional attributes are required to deal with process control and time resulting in significant over specification of the system.

3 Timed CSP

Timed CSP [15] extends the well-known CSP (Communicating Sequential Processes) notation of Hoare [8] with timing primitives. As indicated by its name, CSP is an *event* based notation primarily aimed at describing the sequencing of behaviour within a process and the synchronisation of behaviour (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to consider temporal aspects of sequencing and synchronisation.

CSP adopts a symmetric view of process and environment. Events represent a co-operative synchronisation between process and environment. Both process and environment may control the behaviour of the other by *enabling* or *refusing* certain events or sequences of events.

3.1 Process primitives

The primary building blocks for Timed CSP processes are sequencing, parallel composition, and choice.

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

$$a @ t \xrightarrow{\delta} P(t).$$

The event a is initially enabled and occurs as soon as it is also enabled by its environment. 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 process) at which the event a occurs and allows the subsequent behaviour, P, to depend on its value. The second (optional) timing parameter, δ, is the *stabilisation* time. It delays the commencement of $P(t)$ for at least time δ after the occurrence of the a event, that is $P(t)$ commences at relative time $t + \delta$. If omitted the event stabilisation time is 0 s by convention.

The second form of sequencing is process sequencing. A distinguished event $\checkmark$ 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 $\checkmark$ and then proceeds to act as Q; the termination signal is hidden from the process environment. The process which may only terminate is written $\mathit{S\small{KIP}}$.

The parallel evolution of processes P and Q, synchronised on event set X is written

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

No event from X is enabled in $P \,|[\, X \,]|\, Q$ unless enabled jointly by both P and Q. Other events occur in either P or Q separately.

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

$$a \rightarrow P \;\Box\; b \rightarrow Q$$

begins with both a and b enabled and performs the first to be enabled by its environment. Subsequent behaviour is determined by the event which actually occured, P after a and Q after b respectively. External choice may also be written in an intentional form,

$$\Box\; a : A \bullet P(a).$$

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

$$a \rightarrow P \sqcap b \rightarrow Q$$

may initially enable either a, or b, or both, as it wishes, but must act subsequently according to which event actually occurred. 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 the allowed *values* of the channel. When an event $c.n$ occurs it is said that *the value n is communicated on channel c*. By convention, 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 $c?n : N \rightarrow P(n)$ to describe behaviour over a range of allowed inputs instead of the longer $\Box\, n : N \bullet c.n \rightarrow P(n)$. Similarly the notation $c!n : N \rightarrow P(n)$ is used instead of $\bigsqcap n : N \bullet c.n \rightarrow P(n)$ 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.

In general, the behaviour 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 intentionally parameterised by state variables. Thus a definition of the form

$$P_{n:N} \mathrel{\widehat{=}} Q(n)$$

represents a (possibly infinite) family of definitions, one for each possible value of n. It is important to note that there is no inherent notion of process state in CSP, but rather that this intentional form of expression is a convenient way to provide a finite representation of an infinite family of process descriptions.

3.2 The queue example

A queue is a process with two channels, *left* and *right* respectively. Elements are added to the queue through communications on the *left* channel and removed through communications on the *right* channel. The sequence of communications on the *right* channel must be a subsequence of those on the *left* channel at all times. The difference between the two sequences is loosely referred to as *the queue* and constitutes the internal state of the queue process.

Initially the queue is empty.

$$Queue \mathrel{\widehat{=}} Q_{\langle\,\rangle}$$

At any stage a new element may enter the queue or the head of the queue (if any) may leave.

$$\begin{aligned} Q_{\langle\,\rangle} &\mathrel{\widehat{=}} \mathit{left}?a : X \xrightarrow{t_j} Q_{\langle a\rangle} \\ Q_{\langle h\rangle ^\frown tl} &\mathrel{\widehat{=}} \mathit{left}?a : X \xrightarrow{t_j} Q_{\langle h\rangle ^\frown tl ^\frown \langle a\rangle} \;\Box \\ &\qquad \mathit{right}!h \xrightarrow{t_l} Q_{tl} \end{aligned}$$

Even for such a simple example CSP scores over Object-Z as a means of describing process control. 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 queue may evolve. On the other hand the state annotations are quite cumbersome, even in this example. There is no standard support for state modeling in the form of mathematical toolkits and libraries nor modular techniques to constructing and reasoning about complex internal state. For example, the CSP queue is not truly generic. Queues with various base types can only be specified by repeating the queue definition for each type.

3.3 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 $\mathit{WAIT}\ t$. The process

$$\mathit{WAIT}\ t;\ P$$

is used to represent P delayed by time t.

Current Timed CSP practice is to prefer the use of explicit waits over the use of event stabilisation constants because the latter notation is considered too terse for easy comprehension. For example, an expression of the form $a \rightarrow \mathit{WAIT}\ \delta;\ P$ is generally used in preference to $a \xrightarrow{\delta} P$, especially in complex expressions.

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

$$a \rightarrow P \rhd\{t\}\ Q$$

will pass control to Q if the a event has not occured by time t, as measured from the invocation of the process.

3.4 The queue with timeouts

The timeouts queue can be described using delays and timeouts.

Once again the initial state is represented by the empty sequence.

$$\mathit{TimedQueue} \mathrel{\widehat{=}} TQ_{\langle\,\rangle}$$

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

$$TQ_{\langle\,\rangle} \mathrel{\widehat{=}} \mathit{left}?a : X \rightarrow \mathit{WAIT}\ t_j;\ TQ_{\langle (a,C)\rangle}$$

When the queue is non-empty the process is ready to accept *left* or *right* events as per the untimed queue with the exception that staleness stamps are updated with each communication and state update delays are introduced. In the event of no communication occuring before the head of the queue becomes stale, the stale element is dropped.

$$\begin{aligned} &TQ_{\langle (h,t)\rangle ^\frown tl} \mathrel{\widehat{=}} \\ &\qquad (\mathit{left}?a : X\ @t_1 \rightarrow \mathit{WAIT}\ t_j; \\ &\qquad\qquad TQ_{ds(t_1+t_j, \langle (h,t)\rangle ^\frown tl) ^\frown \langle (a,C)\rangle} \;\Box \\ &\qquad \mathit{right}!h\ @t_1 \rightarrow \mathit{WAIT}\ t_l;\ TQ_{ds(t_1+t_l, tl)}) \\ &\qquad\qquad \rhd\{t\}\ \mathit{WAIT}\ t_d;\ TQ_{ds(t,tl)} \end{aligned}$$

The advantages and disadvantages of Timed CSP are thrown into even sharper focus by this example. Timed

CSP handles the issues of delays and timeouts simply and elegantly. It is difficult to see how the timeout issue could be treated at all in the standard Object-Z. The Timed Object-Z approach results in a too complex model which over specifies this simple system. Note that this example does not even make use of the multi-threading and synchronisation capabilities of Timed CSP which are clearly well beyond the scope of Object-Z's atomic state transition semantics. On the other hand, the treatment of internal state in the above has become intolerably complex, distracting strongly from the basically elegant treatment of the delay and timeout issues.

4 Blending Object-Z and Timed CSP

Object-Z and Timed CSP seem in many ways to 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 multi-threading and synchronisation 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 synchronisation. CSP practitioners tend to make use of notation inspired by the Z mathematical toolkit in the specification of processes with internal state. This is not surprising given their joint origins in the Programming Research Group, Oxford. Another important connection is the well-known duality between the state transition behavioural model and the event based behavioural model [7] which makes it a simple matter to develop complementary semantics for the two langauges.

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 [6] and Smith [16] 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 therefore not well suited to considering multi-threading and real-time. Restricting operations to atomic events completely collapses the concurrent and temporal aspects of operations, everything happens at a single point and instantaneously.

The approach taken in the TCOZ notation is to model operations as terminating CSP processes and to model objects as non-terminating CSP processes. A simple operation consists of accepting a number of inputs, performing a calculation on the object state, performing a number of outputs, and then terminating. Each input and output communication is represented by a separate event and occurs on a CSP-style channel. Since the atomic operation's input and output communications are separated in time by calculations on the local state, it becomes feasible to specify the temporal properties of this calculation when describing the operation. Note that the temporal properties of an entire operation are not so easily specified. CSP communications require the co-operation of the environment and may in general be deferred indefinitely by a malicious environment. Another advantage of this approach is that by combining simple operations with CSP operators it becomes possible to represent true multi-threaded computation even at the operation level. 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.

4.1 Extensions to operation definition

The Object-Z operation definition conventions are extended in two ways.

The first extension is to the variable declaration conventions used in the operation schemas. In order to support fully the CSP channel a new class of declaration is introduced to Object-Z. An input or output parameter may be declared to have occurred on a particular channel. For example, a declaration of the form

$$x? : X \ \mathbf{on}\ c$$

within an operation schema not only declares an input parameter $x?$ of type X, but also requires that an input communication $c?x$ must occur before the operation commences state update calculations. Output communications may be declared in a similar manner, except that output communications are not enabled until after the completion of any state update calculations. A variant form of input communication declaration, analogous to the general form of Timed CSP event sequencing, is allowed. A declaration of the form

$$x? : X \ \mathbf{on}\ c@t$$

declares a time variable t in addition to the input variable $x?$. The time variable t represents the time, relative to the start of the operation, at which the input occurred. For several reasons, including backward compatiblity, it is convenient to retain the standard input and output decorations. Variables of the form $x?$ and $x!$ not assigned to channels are called *simple parameters* and do not correspond to any CSP event activity.

In order to allow consideration of the temporal properties of an operation, a distinguished identifier δ is introduced to represent the duration of the state calculations performed by the operation. This time is calculated starting from the time of occurence of the last input communication required by the operation. This convention is adopted because the semantics of Timed CSP mean that communications events may be delayed indefinitely by a malicious environment, so that it is generally not possible to put a bound on the duration of the overall operation. Only the actual state update calculations may be bounded. If the total elapsed time is required in a specification, it may be obtained by placing timing variables on the input communications as described above.

As an example, consider the specification of the *Join* operation of the simple queue example making use of these new conventions. The CSP view of the new item input may now be taken, that is that it is communicated to the queue object along a channel *left*. This is indicated by declaring the *item*? parameter to be T **on** *left*. In order to determine the timing charactistics of the operation the condition $\delta = t_j$ is added to the predicate part of the operation schema. Otherwise the schema remains unchanged.

Join

$\Delta(items)$
$item? : X$ **on** *left*

$\delta = t_j \wedge items' = items \frown \langle item? \rangle$

This operation schema is given a Timed CSP interpretation as a terminating process. Its CSP-style behaviour (ignoring state effects) is similar to $left?item \to WAIT\ t_j$.

The second extension is a more radical (at least at first sight) departure from existing Object-Z techniques. Since operations are identified with terminating CSP processes, it is natural to allow their definition in terms CSP primitives, such as event sequencing, as well as through the schema calculus. In the case of the *Leave* operation, the communication of the leaving item must precede the updating of the queue state. Since the schema convention is for outputs to follow state calculations, this behaviour cannot be described using a simple schema. Instead, the update operation is described using a schema

Remove

$\Delta(items)$

$items \neq \langle\,\rangle \wedge \delta = t_l \wedge items' = tail\ items$

and the overall leave operation consists of this operation guarded by a communication on the *right* channel.

$$Leave \mathrel{\widehat=} [items \neq \langle\,\rangle] \bullet right!h \to Remove$$

The first part of the definition of *Leave* is a novel process control primitive known as a *state guard*.

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 *Leave*. It is a schema-based method of restricting the action of the process to initial states for which the queue is non-empty. For other states this process will *deadlock* or *block*, which is to say refuse any communication.

Note that the precondition requirement in the *Remove* schema, though identical, could not achieve the desired restriction on the behaviour of *Leave*. Failure to satisfy a precondition when control is passed to an operation instead results in *divergence*, which is to say unspecified subsequent behaviour. *Remove* places no restrictions at all on its behaviour 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 pre P) and the collection of states for which it will not deadlock is called its *guard* (written grd P).

Schemas vs Processes

A schema expression describes a relationship on or between process state/s. A process expression describes the overall behaviour or evolution of a process. These views are not in general compatible and it is necessary to strictly distinguish the schema calculus from the process calculus. The two exceptions to this are the interpretation of operation schema expressions as terminating processes and the use of initial state schema expressions as state guards. In all other circumstance the schema and process calculi are separate and distinct. For example, if P and Q are operations schema expressions, the expression $a \to (P \wedge Q)$ is legal whilst the expression $(a \to P) \wedge Q$ 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 $\_ \,\Box\, \_$, $\_ \parallel \_$, and $\_ ; \_$, have name-sakes with similar semantics in the CSP process calculus. The convention is adopted that it is always the CSP operator that is intended. In most cases, such as $\_ \,\Box\, \_$ and $\_ \parallel \_$, this is justified by the fact that the schema operator is inspired by the corresponding process operator and that a satisfactory treatment of the operator is only possible in the process semantics. For example, the schema operator

$\_ \Box \_$ does not respect operation refinement, whilst the process operator $\_ \Box \_$ does respect process refinement. In the case of $\_;\ \_$, the state-based and process-based semantics are fully compatible.

4.2 Extensions to classes

The extensions made to classes derive for the most part from the extensions made to operation definitions.

Firstly, the state schema convention is extended to allow the declaration of object communications channels. If c is to be used as a communications 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 hetrogeneous and may carry communications of any type. Contrary to the conventions adopted for internal state variables, channels are viewed as global rather then as encapsulated entities. This is an essential consequence of their role as communications interfaces *between* objects.

The second extension is that as well as operations (terminating processes), non-terminating processes may also be described. A distinguished identifier MAIN is used to determine the default behaviour of objects of a given class.

Consider once again the queue example (operations with duration). In addition to the list of *items*, the state schema must declare channels *left* and *right*. The class operations are described as in Section 4.1 and are then used to define a MAIN process similar to that described in Section 3.2.

$Queue[X]$

$items : \mathrm{seq}\, X$
$left, right : \mathbf{chan}$
Δ
$h : X$

$items \neq \langle\,\rangle \Rightarrow h = head(items)$

INIT

$items = \langle\,\rangle$

$Join$

$\Delta(items)$
$item? : X$ **on** $left$

$items' = items \frown \langle item? \rangle \wedge \delta = t_j$

$Remove$

$\Delta(items)$

$items \neq \langle\,\rangle \wedge \delta = t_l \wedge items' = tail\ items$

$Leave \mathrel{\widehat=} [items \neq \langle\,\rangle] \bullet right!h \to Remove$

$Q \mathrel{\widehat=} (Join \Box Leave);\ Q$
$\mathit{MAIN} \mathrel{\widehat=} \mathit{INIT} \bullet Q$

If this description is less concise than the standard CSP description of Section 3.2 (though it does avoid the need for the separate "empty queue" process definition), it at least represents a more robust and significantly more scalable treatment of process state. 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 standard CSP.

The conciseness of this description is essentially comparable to the the standard Object-Z description of Section 2.1. Additional aspects of this specification represent elegant descriptions of matters not treated at all in that specification. The standard Object-Z specification merely described a data structure and a collection of operations applicable to that data. The TCOZ specification describes the behaviour of an object evolving through time and interacting with its environment.

Using simple parameters

As noted in Section 4.1, operations may have simple parameters which are not assigned to any communications channel. In order to make use of such operations in the definition of process behaviour it is necessary to "produce" values for the input parameters and "consume" values from the outputs as part of the process definition. The general method of consuming outputs is to compose an operation with other operations in such a way that its outputs become their inputs. Much of the production problem may of course be addressed in the complementary manner, but it is necessary to have some method for starting off such a chain in the first place.

This is achieved through the generalisation of the state guard construct to allow the declaration of new variables. If an operation O with simple input parameter $i? : X$ is to be called with value $x : X$ then this may described by $[i : X \mid i = x] \bullet Op$. Note that the declared variable does not have the "?" decoration, the convention is that such decorations occur only within operation schemas. Simple input parameters are decorated at the commencement of the operation and simple output parameters are undecorated at the termination of the operation. If pre, post, input, or output decorations occur in a schema definition it is considered to be an operation schema and cannot appear in a state guard expression.

The state guard with local variable takes the place of CSP's intentional external choice. Thus $[i : X] \bullet P(i)$ is used instead of $\Box\, i : X \bullet P(i)$ and $[i : X] \bullet c.i \rightarrow P(i)$ instead of $c?i : X \rightarrow P(i)$.

The intentional form of internal choice is not used. Instead the outputs of operations are used. Thus $[n! : N];\ P(n)$ is used instead of $\sqcap\, n : N \bullet P(n)$ and $[n! : N];\ c.n \rightarrow P(n)$ instead of $c!n : N \bullet P(n)$.

4.3 The queue with timeouts

Some glimpse of the scalibility of the TCOZ notation may be obtained by considering the queue with timeouts example. The time queue model can be defined by inheriting the TCOZ *Queue* class.

$$
\begin{array}{l}
\textbf{class } \mathit{TimedQueue}[X] \\
\quad \mathit{Queue}[X \times \mathbb{T}][\textbf{redef} : \mathit{Join}, \mathit{Remove}, \mathit{Leave}] \\
\quad \mathit{Join} \\
\qquad \Delta(\mathit{items}) \\
\qquad \mathit{item}? : X \textbf{ on } \mathit{left} @ t_1 \\
\qquad \hline
\qquad \delta = t_j \\
\qquad \mathit{items}' = \mathit{ds}(t_1 + t_j, \mathit{items}) \frown \langle (\mathit{item}?, C) \rangle \\
\quad \mathit{Remove} \\
\qquad \Delta(\mathit{items}) \\
\qquad t_1? : \mathbb{T} \\
\qquad \hline
\qquad \mathit{items} \neq \langle\, \rangle \wedge \delta = t_l \\
\qquad \mathit{items}' = \mathit{ds}(t_1 + t_l, \mathit{tail}(\mathit{items})) \\
\quad \mathit{Leave} \mathrel{\widehat=} [\mathit{items} \neq \langle\, \rangle] \bullet \\
\qquad\qquad \mathit{right}!\mathit{first}(h) @ t_1 \rightarrow \mathit{Remove} \\
\quad \mathit{Drop} \\
\qquad \Delta(\mathit{items}) \\
\qquad \hline
\qquad \delta = t_d \\
\qquad \mathit{items}' = \mathit{ds}(\mathit{second}(h) + t_d, \mathit{tail}(\mathit{items})) \\
\quad \mathit{MT} \mathrel{\widehat=} [\mathit{items} = \langle\, \rangle] \bullet \mathit{Join} \\
\quad \mathit{NMT} \mathrel{\widehat=} [\mathit{items} \neq \langle\, \rangle] \bullet \\
\qquad\qquad ((\mathit{Join} \,\Box\, \mathit{Leave}) \\
\qquad\qquad\quad \triangleright\{\mathit{second}(h)\}\ \mathit{Drop}) \\
\quad \mathit{TQ} \mathrel{\widehat=} (\mathit{MT} \,\Box\, \mathit{NMT});\ \mathit{TQ} \\
\quad \textsc{Main} \mathrel{\widehat=} \textsc{Init} \bullet \mathit{TQ}
\end{array}
$$

The TCOZ specification of this system is of similar complexity to the simple queue specification in TCOZ. The *Join* and *Leave* operations are modified to ensure that stale items are removed and the *Drop* operation is added to be invoked when the head of the queue becomes stale. Each of the operation and process specifications remain simple and elegant. The comparison with the relative jumps in complexity in the Object-Z and Timed CSP specifications would seem to suggest a higher degree of scalability in TCOZ, blending as it does Object-Z's capabilities for handling data complexity with Timed CSP's capabilities for handling process complexity.

4.4 A multithreaded example

In order to give a simple example of the the multithreaded capabilities of the TCOZ notation, a buffered consumer/producer process is considered.

A simple consumer/producer process accepts inputs of type L on its *left* channel, calculates some function

$$| \ f : L \rightarrow R$$

of the inputs, and outputs the result on its *right* channel.

$$
\begin{array}{l}
\textbf{class } \mathit{ConProd} \\
\quad \mathit{left}, \mathit{right} : \textbf{chan} \\
\quad \mathit{Process} \\
\qquad i? : L \textbf{ on } \mathit{left} \\
\qquad o! : R \textbf{ on } \mathit{right} \\
\qquad \hline
\qquad \delta = t_p \wedge o! = f(i?) \\
\quad \textsc{Main} \mathrel{\widehat=} \mathit{Process};\ \textsc{Main}
\end{array}
$$

In order to ensure that all inputs are accepted and outputs are received, the process is buffered left and right with queue processes. The buffered process consists of a left buffer, a right buffer, and an internal consumer/producer.

In order to hookup the buffer queues to the consumer/producer correctly it is necessary to rename the various channels. The renaming convention is essentially the same as for Object-Z and Timed CSP, that is $P[a/b]$ is P with all occurances of b replaced by a. 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.

In order to protect the internal interfaces from environmental interference they are hidden. The hiding notation is again essentially 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 synchronising 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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BufConProd}} \\
\hline
l : Queue[L][ml/right] \\
r : Queue[R][mr/left] \\
cp : ConProd[ml/left, mr/right] \\
\hline
\textit{MAIN} \mathrel{\widehat=} (l \,|[\, ml \,]|\, cp \,|[\, mr \,]|\, r \setminus ml, mr) \\
\hline
\end{array}
$$

The *BufConProd* class definition allows true multithreading of the two buffers and the consumer/producer. 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 co-operation through the requirement for synchronisation between the processes when communication occurs on the internal channels ml and mr.

5 Discussion

Many established formal specification and design notations have tended to concentrate either on data modeling and algorithmic concerns (eg. Z, VDM, Object-Z, etc.) or else on process control concerns (eg. CSP, CCS, StateCharts, etc.). In order to address the needs of complex interactive systems it is necessary to be able to address both in a smoothly integrated, but also highly structured and modular, manner. In this regard, Timed CSP and Object-Z complement each other not only in their expressive capabilities, but also in their underlying semantics. In addition, the object oriented flavour of Object Z provides an ideal foundation for promoting modularity and separation of concerns in system design. The intention of the proposed blending of these notations, TCOZ, is to treat data and algorithmic concerns in the Object-Z style and to treat process control concerns in the CSP style. Timing concerns are distributed according to their status as relating to algorithm or control. TCOZ preserves in large part both the syntax and semantics of the individual notations and hence can potentially benefit from the large body of experience developed in the use of and tool support for the individual notations and their parents, such as verification and refinement techniques and tools, model checkers, etc..

The basis for the successful blending of the two notations is the duality between state transition semantics and event semantics. This has long been recognised [7, 9, 13] and has undoubtedly helped shape the development of Object-Z's behavioural semantics. Perhaps the most mature formalism based on this duality is Butler's blending of CSP with Back's Action Systems [1]. An important lesson from this work is the need to distinguish strongly between the notions of guards and precondition in the state-transition view. The failure to do so in Object-Z has made it impossible to reconcile algorithmic refinement with behavioural semantics, greatly reducing the value of refinement in Object-Z. The adoption of a distinct notion of state guards in TCOZ makes possible a full blending of Z-style algorithm and CSP-style process refinement.

The notion of blending the untimed CSP and Object-Z has been proposed independently by Fischer [6] and Smith [16]. Both take the approach of identifying the notion of channel with that of operation, operation invocations becoming atomic events. Not only does this prevent the modeling of timing and concurrency at the operation level, it also forces an unnatural class structure since in order to communicate, objects must have like named operations. Neither formalism makes a full distinction between preconditions and guards[2] so that refinement is greatly restricted in both.

This paper serves merely as brief introduction to the TCOZ notation. Seperate papers describe the blended state/event process model which forms the basis for the TCOZ semantics [11] and a case study in using TCOZ to specify a real-time multi-lift system [10]. 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 [14], it is hoped that this might also form a valuable addition to the TCOZ notation.

We believe that TCOZ to be a good candidate for specifying complex parallel and distributed real-time systems.

Acknowledgements
We would like to thank John Colton, Neale Fulton and Lin Zucconi for many helpful comments on this work. This work is supported in part by the DSTO/CSIRO Fellowship programme.

REFERENCES

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

[2] J.S. Dong, J. Colton, and L. Zucconi. A formal object approach to real-time specification. In *the 3rd Asia-Pacific Software Engineering Conference*

[2] Fischer does allow a distinction between guards and preconditions, but the default is to identify them.

(APSEC'96), Seoul, Korea, December 1996. IEEE Computer Society Press.

[3] J.S. Dong, G. Rose, and R. Duke. The role of secondary attributes in formal object modelling. In Alex Stoyenko, editor, *The First IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)*, pages 31–38, Ft. Lauderdale, USA, November 1995. IEEE Computer Society Press.

[4] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

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

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

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

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

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

[10] B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: Applying TCOZ. Technical Report 97-23, Mathematical and Information Sciences, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia, 1997.

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

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

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

[14] 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.

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

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

[17] Graeme Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.