Active Objects in TCOZ

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

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

Abstract

Active objects have their own thread of control and passive objects are controlled by others. In Object-Z, all objects are modelled as passive objects. Timed Communicating Object Z (TCOZ) extends the Object-Z notation with Timed CSP's process and timing constructs. The blending of the concepts of object and non-terminating process leads to the notion of active object in TCOZ. This paper considers the concept of active object, its role and implications in formal real-time object-oriented specification.

1. Introduction

Object-Z [5] is strong in modelling complex system states, but weak in modelling system behaviours and timing properties. All component objects are modelled as passive entities which are controlled by a system object, while the system object is controlled by an implicit user of the system. Timed Communicating Object Z (TCOZ) extends the Object-Z notation with process and timing constructs from Timed CSP [16] so that the state, behaviour and timing aspects of a system can be all captured within one notation. The integration of the concepts of object (from Object-Z) and non-terminating process (from CSP) leads to a powerful modelling approach in TCOZ, based on the notion of active object.

Active objects have their own thread of control, while passive objects are controlled by others. This allows active objects to behave in a truly concurrent manner. The state attributes and local operations of active objects are completely encapsulated, while passive objects must be explicitly referenced so that they can be sent messages. Communication between active objects is achieved through the use of global interfaces modelled as CSP channels or as sensors and actuators . This both increases the modularity of TCOZ specifications and reduces the coupling between process classes. The ability to model active objects is a powerful and novel aspect of TCOZ.

The basic TCOZ notation has been briefly described in an introductory paper [14]. The discussion of active objects in TCOZ is deliberately left out from [14] because we believe that the notion of active object provides a new way for modelling systems and deserves a further investigation. In this paper, we present detailed discussion and classification of active objects. Their role and implications in formal real-time object-oriented specification are analysed. From passive modelling to active modelling there is a cultural change. A number of small specification examples are presented in this paper with an aim to demonstrate how and when to use active modelling techniques.

It is assumed that the reader has some familiarity with both Object-Z [5] and CSP [10], since the mechanics of blending the two notations is considered only briefly in Section 2. A glossary of TCOZ language features may be found in Appendix A. Section 3 illustrates how to define and derive (via inheritance) active objects in TCOZ; how active objects are composed and interact with each other. Section 4 classifies the active objects according to independence and co-operativeness. Section 5 presents multiple views (passive/active) of a simple concurrent communication system and demonstrates the advantages of using active views to model such systems.

2. TCOZ features

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

In this section we briefly consider the aspects of TCOZ which help to bring the two notations together.

2.1. A model of time

In TCOZ, all timing information is represented as real valued measurements in *seconds*. 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 [8] are adopted. Under this convention, time quantities are represented by the type $\mathbb{R}\mathtt{s}$, where $\mathbb{R}$ represents the real numbers and $\mathtt{s}$ is the SI symbol for the standard unit of time. Time literals consist of a real number literal annotated with a symbol representing a unit of time. For example, $3\,\mu\mathtt{s}$ is a literal representing a period of three microseconds, that is three millionths of the standard time unit, the second. All the SI standard units symbols are supported and all the arithmetic operators are extended in the obvious way to allow calculations involving units of measurement.

2.2. Deadlines and delays

In order to describe the timing requirements of operations and sequences of operations, a deadline command inspired by Hayes and Utting [9] is introduced. If *OP* is an operation specification (defined through any combination of CSP process primitives and Object-Z operation schemas) then $OP \bullet \textsc{Deadline}\, t$ describes the process which has the same effect as *OP*, but is constrained to terminate no later than *t*. If it cannot terminate by time *t*, it deadlocks.

The WAITUNTIL operator is a dual to the deadline operator. The process $OP \bullet \textsc{WaitUntil}\, t$ performs *OP*, but will not terminate until at least time *t*. If it would normally terminate before time *t*, it idles.

2.3. Declaring channels

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

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

2.4. Sensor and Actuators

Complementary to synchronous CSP channels, two continuous (asynchronous) interface mechanisms inspired by the control system architecture [17] are introduced in TCOZ [12]. The **sensor** provides a sampling channel linked to a global analogue variable. The **actuator** provides a local-variable linked to a global analogue variable. Sensors and activators may appear either at the system boundary (describing how global analogue variables are sampled from or generated by the digital subsystem) or else within the system (providing a convenient mechanism for describing local communications which do not require synchronisations).

2.5. Guards and preconditions

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

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

2.6. Complex network topologies

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

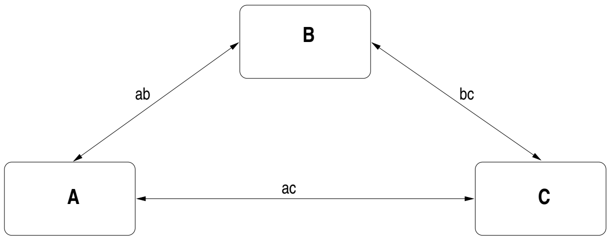

Figure 1. Two dimensional topology

bc. One CSP expression for such a network communication system is

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

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

The above expression not only suffers from syntactic clutter, but also serves to obscure the inherently simple network topology described so elegantly by Figure 1. In TCOZ, a graph-based approach is adopted to represent the network topology [11]. For example, the network topology of Figure 1 may be described by

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

2.7. Semantics of TCOZ

A separate paper details the blended state/event process model which forms the basis for the TCOZ semantics [13]. In brief, the semantic approach is to identify the notions of operation and process by providing a process interpretation of the Z operation schema construct. TCOZ differs from many other approaches to blending Object-Z with a process algebra in that it does not identify operations with events. Instead an unspecified, fine-grained, collection of state-update events is hypothesised. Operation schemas are modelled by the collection of those sequences of update events that achieve the state change described by the schema. This means that there is no semantic difference between a Z operation schema and a CSP process. It therefore makes sense to also identify their syntactic classes.

The process model used by TCOZ consists of sets of tuples consisting of: an *initial* state; a *trace* (a sequence of time stamped events, including update-events), a *refusal* (a record what and when events are refused by the process), and a *divergence* (a record of if and when the process diverged). The trace/refusal pair is called a *failure* and the overall model 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 ✓ event occurs), then the state at the time of termination is called the *final* state.

The process model of an operation schema consists of all initial states and update traces (terminated with a ✓) 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. An advantage of this semantics is that it allows CSP process refinement to agree with Z operation refinement.

The process model of an active class (see Section 3) encapsulates the object state by hiding all of the update-events. There can be no shared-variable contentions between active objects in TCOZ.

3. Defining active objects in TCOZ

In TCOZ, classes may be categorised as either *active* (having their own thread of control) or *passive* (under the control of another object). Active classes are distinguished by the presence of a MAIN process definition. The MAIN process is used to determine the process semantics of objects of the class. In the absence of a MAIN process definition, passive classes cannot be given process semantics and they are instead treated as libraries of state and process definitions which may be reused by other classes through inheritance.

3.1. Inheritance between active/passive classes

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

In TCOZ (like Object-Z), 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 initialisation schemas of inherited classes and those declared in the derived class are conjoined. Operation schemas with the same name are also conjoined.

The active object basic ingredients, '**chan**' and '**sensor**'/'**actuator**', are treated as state attributes of a class, therefore, they are pooled into the derived classes. Channel and sensor/actuator renaming is the same as the state attribute renaming. The non-terminating behaviour description 'MAIN' of an active class is novel to TCOZ. 'MAIN' distinguishes the definition of active and passive classes. Therefore the rules for (active and passive) class inheritance are mainly related to the reuse rules for 'MAIN':

- A new passive class can also be derived from an existing active class, in this case, the MAIN process of the existing class is implicitly removed in the definition of the passive class.

- When a new active class is derived from an existing active class, the objects of the new class often have behaviour different from the objects of the existing class. As a consequence, the MAIN process is generally redefined explicitly. However, it may sometimes be convenient to inheret the parent class MAIN process definition. In this case, the MAIN is attached to the parent class name in the inheritance list of the child class. For example,

  B
  $A(\mathrm{MAIN})$
  $\ldots$
  $\ldots$

  indicates that a class B inherits the structure and the MAIN class definition of A.

- A new passive class can be derived from an existing passive class following the same rules as the standard Object-Z.

- A new active class can be derived from an existing passive class, in this case, a MAIN process definition needs to be added to the new class.

3.2. Composition and interaction of active objects

A composite object, say c, can passively access a component object, say ob, by invoking an operation (Op) of the component via the dot notation, e.g. $ob.Op$. However, as an active object has its own thread of control, its features must not be accessed in this way. For instance, the following composition of an object of A (supplier) in B (a client) is forbidden in TCOZ:

A
$v_1 : T_1$
$\ldots$
$v_m : T_m$
c_1 : **chan**
$\ldots$
$OpA_1 \mathrel{\widehat=} \ldots$
$\ldots$
$OpA_n \mathrel{\widehat=} \ldots$
$\mathrm{MAIN} \mathrel{\widehat=} \ldots$

B
$a : A$
$\ldots$
$\ldots$
$OpB_1 \mathrel{\widehat=} a.OpA_1$
$\ldots$

The above specification error can be detected syntactically. As the behaviour of an active object is completely described by MAIN, MAIN is the only operation that can be accessed by a client object. Other operations must be invisible to its client. The general rule is:

> In every active class, all operations, except MAIN, and all state attributes, except channels and analogue interfaces, are implicitly hidden from the environment by being placed in the class invisibility list[1] [3]. The various channel and analogue interfaces are global names and are not features of individual objects. They appear in neither the visibility nor the invisibility list of a class.

For example, the class A above is equivalent to the class A_I, which is A enhanced with a class invisibility-list.

A_I
$\not\upharpoonright(v_1, \ldots, v_m, OpA_1, \ldots, OpA_n)$
$\langle$Body of $A\rangle$

A composite object that contains active objects is also an active object. A MAIN definition must appear in the composite object class definition. As the MAIN operation implicitly includes the state initialisation INIT for every active object, the composite active objects will implicitly include their active component objects' initial states. No explicit initialisation is required, nor allowed, for active components.

Since the internal state of active components is encapsulated, they are essentially pure CSP processes. We adopt the syntactic convention of identifying the object name with its MAIN process. This allows active components to be composed using CSP operators. For example, if ob_1 and ob_2 are active object components, the independent parallel composition behaviour the two objects can be expressed as $ob_1 \mid\mid\mid ob_2$, which means $ob_1.\mathrm{MAIN} \mid\mid\mid ob_2.\mathrm{MAIN}$.

4. Classification of active objects

Active objects can be classified into three categories according to the nature of their interactions with their environment: synchronous, asynchronous and mixed active objects. In this section, the application domains of the various kinds of active objects are illustrated through a typical example of each kind of object.

4.1. Synchronous active objects

Synchronous active objects are highly co-operative. A hand-shake communication is required between such ob-

[1] In general, features (attributes and operations) of a class are accessible to its client via dot notation. The class invisibility-list [3], '$\not\upharpoonright(\ldots)$', lists features that cannot be accessed by its clients. It complements the class visibility-list.

jects. Their communication interfaces are discrete, synchronous channels. For example, a button that can be pushed 'on' and 'off' can be modelled as a synchronous active object.

Button

$$on, off : \textbf{chan} \qquad \text{[the button on/off]}$$

$$
\begin{array}{l}
On \mathrel{\widehat{=}} off \to Off \\
Off \mathrel{\widehat{=}} on \to On \\
\textsc{Main} \mathrel{\widehat{=}} Off
\end{array}
$$

4.2. Asynchronous active objects

Asynchronous active objects are highly independent. Their communication interfaces are analogue, asynchronous sensor/actuators. Since they do not synchronise with other objects they are usually characterised by some form of periodic behaviour.

A typical asynchronous object is a clock which displays year, month, date, hour, minute and second. The calendar-time is modelled as a six-tuple of integers.

$$
\begin{array}{l}
CalendarTime == \\
\quad \mathbb{N}\,\mathsf{yr} \times \mathbb{N}\,\mathsf{mn} \times \mathbb{N}\,\mathsf{dy} \times \mathbb{N}\,\mathsf{hr} \times \mathbb{N}\,\mathsf{min} \times \mathbb{N}\,\mathsf{s}\,.
\end{array}
$$

The clock stores the total elapsed time since some reference date and the function

$$
\begin{array}{l}
Convert : \mathbb{N}\,\mathsf{s} \to CalendarTime \\
\hline
\ldots \qquad \text{[detail of function omitted]}
\end{array}
$$

is used to convert this elapsed to a calendar-time.

The elapsed time counter must be incremented every second and the display screen updated in less than 50 milliseconds.

Clock

$$
\begin{array}{l}
per == 1\,\mathsf{s} \\
gain == 50\,\mathsf{ms}
\end{array}
$$

$$
\begin{array}{l}
total : \mathbb{N}\,\mathsf{s} \\
\Delta \\
display : CalendarTime\ \textbf{actuator} \\
\hline
display = Convert(total)
\end{array}
$$

Inc

$$
\begin{array}{l}
\Delta(total) \\
\hline
total' = total + 1
\end{array}
$$

$$
\begin{array}{l}
\textsc{Main} \mathrel{\widehat{=}} \mu\, C \bullet Inc \\
\quad \bullet \textsc{Deadline}\ gain \bullet \textsc{WaitUntil}\ per;\ C
\end{array}
$$

The actuator *display* is modelled as a secondary attribute [4] whose value depends on the value of the primary attribute *total*.

An active object with strictly periodic behaviour can only conveniently be represented using sensor/actuators. Channels are not the appropriate mechanism for modelling periodic objects because channels involve synchronisations and are therefore subject to arbitrary delay by an uncooperative environment.

4.3. Mixed active objects

Active objects may have both synchronous and asynchronous interfaces. Such mixed active objects are both independent and co-operative. For example, consider a security system built in a house. The security device can be turned 'off' (when the owner is home) and turned 'on' (when the owner is away). If the security system is turned on, then a sensor will detect any intrusion into the house. If an intrusion is detected, the alarm will be switched on within one second and will be continuously making a alarm for up to 30 minutes if not turned off.

There are three components, detector, speaker and on/off button in a security system. The detector can be modelled as a sensor, the speaker can be modelled as an actuator and the on/off buttons can be modelled as channels.

$$
\begin{array}{l}
Condition ::= normal \mid intrusion \\
Reaction ::= silence \mid noise
\end{array}
$$

SecSys

$$
\begin{array}{l}
detector : Condition\ \textbf{sensor} \qquad \text{[intruder]} \\
speaker : Reaction\ \textbf{actuator} \qquad \text{[alarm]} \\
on, off : \textbf{chan} \qquad \text{[system activation]}
\end{array}
$$

$\textsc{Init}$

$$speaker = silence$$

Alarm

$$
\begin{array}{l}
\Delta(speaker) \\
\hline
speaker' = noise
\end{array}
$$

Quiet

$$
\begin{array}{l}
\Delta(speaker) \\
\hline
speaker' = silence
\end{array}
$$

$$
\begin{array}{l}
Ready \mathrel{\widehat{=}} \\
\quad (detector?intrusion \to (Alarm \bullet \\
\quad \textsc{Deadline}\ 1\,\mathsf{s} \bullet \textsc{WaitUntil}\ 30\,\mathsf{min}); \\
\quad Quiet;\ Ready) \mathbin{\nabla} off \to Quiet;\ Rest \\
Rest \mathrel{\widehat{=}} on \to Ready \\
\textsc{Main} \mathrel{\widehat{=}} Rest
\end{array}
$$

In the definition of the operation *Ready*, '∇' is an interrupt operator. In general, $P \mathbin{\nabla} e \to Q$ denotes a process that performs P until exception e, then transfer control to Q. Note that the objects of *SecSys* are non-periodic objects.

5. From passive to active views

As TCOZ preserves Object-Z as a proper sub-language, the Object-Z flavoured passive view and the CSP flavoured active view can be both supported in TCOZ. For the same system requirement, differing abstract design-views can produce differing specification/design representations. It has been demonstrated by many Object-Z specifications that the passive view is appropriate for modelling many systems. We believe that the active view is more appropriate for modelling concurrent reactive systems. In this section, we use the simple example of an N-place buffer to demonstrate some of the limitations of the passive view and therefore promote the active view. In addition, we present an incremental way of migrating from the passive view to the active view by gradually introducing active features.

5.1. Two communicating buffers

Consider two communicating N-place buffers, the left-hand and the right-hand buffers respectively. Each buffer stores a sequence of elements ordered by their time of arrival. Elements join the system on the left-hand-side, work their way to the right-hand buffer, and eventually leave the system on the right-hand-side. We require that each buffer take no more than t_j μs for adding a new element and t_l μs for removing an existing element.

5.2. The passive view

We first consider a passive model of an N-place buffer. In the pure passive-view the buffer is described solely by the operations which the buffer may perform.

$Buffer_p[X]$

$N : \mathbb{N}$
$t_j, t_l : \mathbb{R}\,\mu\mathrm{s}$

$Q : \mathrm{seq}\,X$
$\#Q \leq N$

INIT
$Q = \langle\,\rangle$

Add
$\Delta(Q)$
$i? : X$
$Q' = \langle i? \rangle \frown Q$

Remove
$\Delta(Q)$
$i! : X$
$Q' = front(Q)$
$i! = last(Q)$

$Join \mathrel{\widehat=} [\#Q < N] \bullet (Add \bullet \mathrm{DEADLINE}\; t_j)$

$Leave \mathrel{\widehat=} [Q \neq \langle\,\rangle] \bullet (Remove \bullet \mathrm{DEADLINE}\; t_l)$

Although we have used DEADLINE commands to limit the processing time of the operations, this model offers a purely passive view of the buffer. There are no communications interfaces and no MAIN process to determine the behaviour of the object. Nevertheless, this model is a reasonably satisfying one, since it gives the reader a good intuition as to how a buffer will behave.

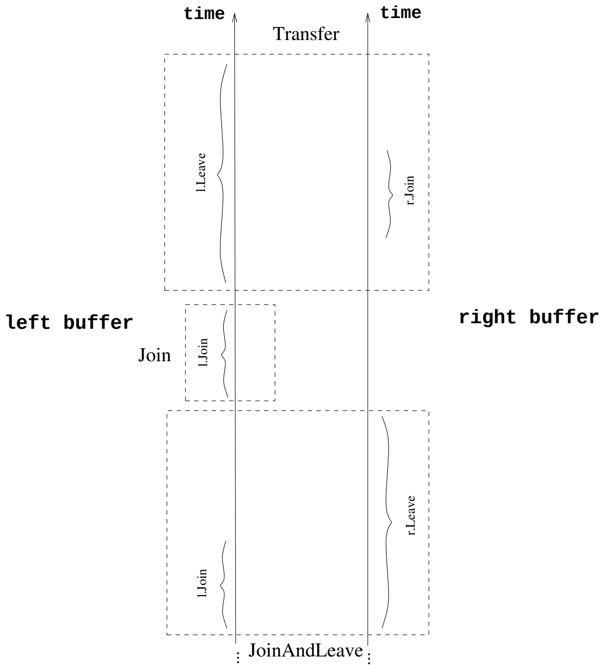

Figure 2. Event diagram for the passive model

A passive view of the two-buffer system may be constructed using the passive buffers as components.

$TwoBuffers_p[X]$

$l, r : Buffer_p[X]$
$l \neq r$

INIT
$l.\mathrm{INIT} \wedge r.\mathrm{INIT}$

$Join \mathrel{\widehat=} l.Join$
$Leave \mathrel{\widehat=} r.Leave$
$Transfer \mathrel{\widehat=} l.Leave;\; r.Join$
$JoinAndLeave \mathrel{\widehat=} Join \;|||\; Leave$

If the basic operations were left at *Join*, *Leave*, and *Transfer*; the $TwoBuffers_p$ definition would also be reasonably satisfying. However, that would ignore the obvious possibilities for parallelism inherent in the use of two buffer components. In order to exploit this capability to a limited extent we have defined a 'parallel' operation *JoinAndLeave*. This operation performs a *Join* on the left buffer and a *Leave* on the right buffer at the same time. One possible event trace for this model is illustrated in Figure 2. However, more arbitrary concurrent event traces which allow total and/or partial overlapping between *l.Join* and *r.Leave* in time duration (illustrated in Figure 3) are dissallowed by this model. Each possible concurrent situation would have to be explicitly modelled as a separate possible operation.

5.3. Towards the active view

Once the buffer is modelled as a purely passive object without even communication interfaces, the composite $TwoBuffers_p$ cannot use its components in any active mode. The active mode requires the use of communication interfaces since these are the only method of accessing an active object's internal features. We now move toward a partially active view by extending the buffer class with input/output channels, using the inheritance mechanism.

$$
\begin{array}{l}
\underline{\mathit{Buffer}_t[X]} \\
\mathit{Buffer}_p[X][\ \mathbf{redef} : \mathit{Join}, \mathit{Leave}] \\
\hline
\quad \mathit{left}, \mathit{right} : \mathbf{chan} \\
\hline
\mathit{Join} \mathrel{\widehat{=}} [i : X \mid \#Q < N] \bullet \\
\qquad \mathit{left}?i \rightarrow (\mathit{Add} \bullet \mathrm{DEADLINE}\ t_j) \\
\mathit{Leave} \mathrel{\widehat{=}} [Q \neq \langle\, \rangle] \bullet \\
\qquad \mathit{right}!\mathit{last}(Q) \rightarrow (\mathit{Remove} \bullet \mathrm{DEADLINE}\ t_l)
\end{array}
$$

As the class $Buffer_t$ has an active communications interface, its operations (as opposed to the buffer itself) may be used in an active mode. We demonstrate this by describing an active model of the two-buffer system that inherits the $Buffer_t$ class.

$$
\begin{array}{l}
\underline{\mathit{TwoBuffers}_t[X]} \\
\hline
\quad l : \mathit{Buffer}_t[X][\mathit{middle}/\mathit{right}] \\
\quad r : \mathit{Buffer}_t[X][\mathit{middle}/\mathit{left}] \\
\hline
\mathit{Join} \mathrel{\widehat{=}} l.\mathit{Join} \\
\mathit{Leave} \mathrel{\widehat{=}} r.\mathit{Leave} \\
\mathit{Transfer} \mathrel{\widehat{=}} l.\mathit{leave} \mid[\, \mathit{middle} \,]\mid r.\mathit{Join} \\
\mathrm{MAIN} \mathrel{\widehat{=}} (\mu\, T \bullet \\
\qquad \mathit{Transfer} \;\Box\; (\mathit{Join} \mid\mid\mid \mathit{Leave}) \;\Box \\
\qquad \mathit{Join} \;\Box\; \mathit{Leave};\ T) \setminus \mathit{middle}
\end{array}
$$

However, because the component objects remain passive, truly concurrent event traces such as in Figure 3 are still disallowed in this model. Any concurrency must still be explicitly modelled as in the *Join* ||| *Leave* alternative.

5.4. The active view

We can develop the partially active buffer $Buffer_t$ by defining a default active behaviour for the buffers.

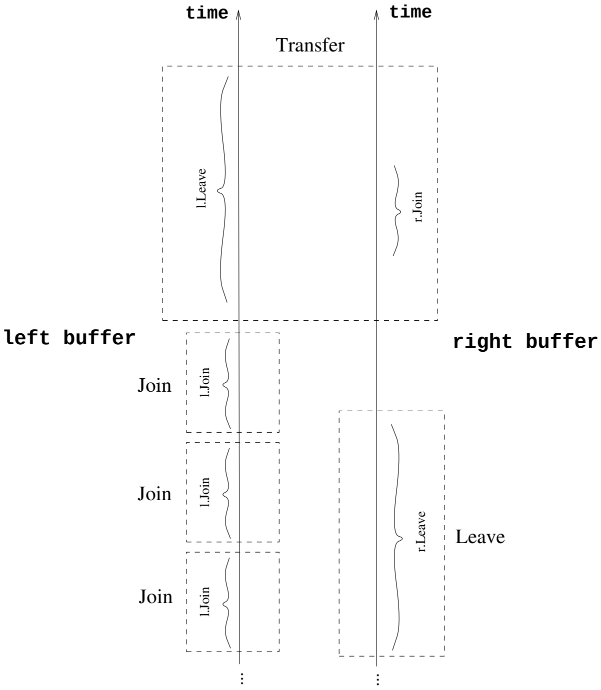

Figure 3. Event diagram for the active model

$$
\begin{array}{l}
\underline{\mathit{Buffer}_s[X]} \\
\mathit{Buffer}_t[X] \\
\mathrm{MAIN} \mathrel{\widehat{=}} \mu\, B \bullet (\mathit{Join} \;\Box\; \mathit{Leave});\ B
\end{array}
$$

Since $Buffer_s$ is a fully active class, the left and right buffers may be composed using CSP process operators.

$$
\begin{array}{l}
\underline{\mathit{TwoBuffers}_s[X]} \\
\hline
\quad l : \mathit{Buffer}_s[X][\mathit{middle}/\mathit{right}] \\
\quad r : \mathit{Buffer}_s[X][\mathit{middle}/\mathit{left}] \\
\hline
\mathrm{MAIN} \mathrel{\widehat{=}} \big\|(l \xleftrightarrow{\mathit{middle}} r)
\end{array}
$$

Once given their own independent threads of control, the two buffers are able to fully exploit the system's potential for parallelism. The truly concurrent event trace illustrated in Figure 3 is allowed in this model. Moreover, since active objects fully encapsulate their internal features, there is no need to, nor is it possible to, explicitly consider the possible interleavings and overlappings of operations in the left and right buffer. All this detail has been treated on a more local level.

6. Conclusion

The integration of the concepts of object (from Object-Z) and process (from CSP) leads to a powerful modelling approach in TCOZ, based on the notion of the active object. In an active object, internal structure is completely encapsulated and global channels and sensors/actuators are used to provide the communication interface for active objects. This encapsulation enhances the modularity of TCOZ specifications and the adoption of global communications mechanisms reduces design coupling between TCOZ classes. Passive object-oriented designs typically have high degrees of coupling because of the need to know the names of class features in order to access them. The behaviour of an active object is defined by a non-terminating MAIN process. Since active objects are *pure* processes, they may be composed in any way that processes may be composed and thus active objects provide the ideal tool for describing complex, concurrent systems.

Recently there has been active investigation of the integration of state modelling techniques with process description languages. The ability to distinguish active and passive views and to adopt fully active object models is a novel feature of TCOZ, that distinguishes it from other similar approaches which blend Z/Object-Z with either CSP [18, 6] or CCS [7, 19].

In this paper, we have presented detailed discussions of the concept, classification, and role of active objects in formal real-time object-oriented specification. Although passive modelling in TCOZ is still allowed, we believe that in the modelling of large systems, whether concurrent or not, the use of active objects will play a dominate role. From passive modelling to active modelling there is a cultural change for the specifier/designer. We believe this change is towards more modular, more object-oriented modelling because we can isolate each component and concentrate on its behaviour and interface without explicitly handcrafting each cross reference in order to send messages to other objects.

This paper only considered the notion of active object in the formal specification context. It will be very interesting to investigate the connection to current trends in object-oriented programming. As further work, we are interested in developing refinement/prototyping rules from TCOZ to concurrent object-oriented programming languages, such as Pool [1], Hybrid [15] and Eiffel// [2].

Acknowledgements

This work is supported in part by the DSTO/CSIRO Fellowship programme. We would like to thank anonymous referees for many useful comments.

References

[1] P. America, J. de Bakker, J. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. 13th ACM Symp. on Princ. Prog. Lang.*, pages 194–208, 1986.

[2] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9), 1993.

[3] J.S. Dong and R. Duke. Exclusive Control within Object Oriented Systems. In *The 18th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'Pacific95)*, pages 123–132. Prentice-Hall, November 1995.

[4] 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.

[5] 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.

[6] C. Fisher and G. Smith. Combining CSP and Object-Z: Finite or Infinite Trace Semantics? In *IFIP International Conference on Formal Description Techniques and Protocal Specification, Testing and Verification*, pages 503–518. Chapman & Hall, November 1997.

[7] A. J. Galloway and W. J. Stoddart. An operational semantics for ZCCS. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 272–282, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

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

[9] I. J. Hayes and M. Utting. Coercing real-time refinement: A transmitter. In D. J. Duke and A. S. Evans, editors, *BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer Verlag, 1997.

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

[11] B. Mahony and J.S. Dong. Network Topology and a Case Study in TCOZ. In J. Bowen, A. Fett, and M. Hinchey, editors, *The 11th International Conference of Z Users*, volume 1493 of *Lecture Notes in Computer Science*, pages 308–327, Berlin, Germany, September 1998. Springer-Verlag.

[12] B. Mahony and J.S. Dong. Sensors and Actuators: Process Control in TCOZ. Technical Report 98-38, Mathematical and Information Sciences, Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia, 1998.

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

[14] B. P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Computer Society Press.

[15] O. Nierstrasz. Active objects in Hybrid. In *Proc. 2nd ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'87)*, 1987.

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

[17] M. Shaw. Beyond objects. *ACM Software Engineering Notes*, 20(1), January 1995.

[18] 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.

[19] K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

A. TCOZ glossary

Notation	Explanation
c : **chan**	declare c to be a channel
$\perp$	divergent process
$\textsc{Stop}$	deadlocked process
$\textsc{Skip}$	terminate immediately
$\textsc{Wait}\ t$	delay termination by t
$a \rightarrow P$	communicate a then do P
$a@t \rightarrow P$	communicate a at time t
$c.a$	communicate a on channel c
$c?a$	input a on channel c
$c!a$	output a from channel c
$[b] \bullet P$	enable P only if b
$P;\ Q$	perform P until termination, then perform Q
$P \Box Q$	perform the first enabled of P and Q
$[i : I] \bullet P$	perform P with first enabled value of i (indexed external choice)
$P \sqcap Q$	perform either of P and Q
$[i! : I];\ P$	perform P with any value of i (indexed internal choice)
$v := e$	syntactic sugar for $[\Delta v \mid v' = e]$
$P \setminus A$	hide the events A from the environment of P
$P \parallel Q$	synchronise P and Q on all events
$P \mid[A]\mid Q$	synchronise P and Q on events from A
$(\Vert p_1, \ldots, p_n \bullet \ldots;\ p_i \xleftrightarrow{A} p_j;\ \ldots)$	network topology abstraction with parameters $p_1, \ldots, p_n$ and network connections including p_i communicating with p_j on private channels from A
$P \mid\mid\mid Q$	P and Q running without sychronisations
$P \triangleright \{t\}\ Q$	if P does not begin by time t, perform Q instead
$P \swarrow \{t\}\ Q$	perform P until time t, then transfer control to Q
$P \nabla e \rightarrow Q$	perform P until exception e, then transfer control to Q
$P \bullet \textsc{Deadline}\ t$	behaviours of P which terminate before time t
$P \bullet \textsc{WaitUntil}\ t$	after P idle until time t