Formal Aspects of Computing (2002) 13: 142–160
© 2002 BCS

Formal Aspects of Computing

Deep Semantic Links of TCSP and Object-Z: TCOZ Approach

Brendan Mahony[1] and Jin Song Dong[2]

[1] Information Technology Division, Defence Science and Technology Organisation of Australia, Edinburgh, South Australia, Australia
[2] Computer Science Department, School of Computing, National University of Singapore, Singapore

Abstract. Formal methods can be used in effective combination only if the semantic links between individual methods are clearly established. This paper discusses the semantic design of TCOZ, a language blended from Object-Z and TCSP. The semantic model adopted is the infinite timed failures model of TCSP, extended to include initial state and update events for modelling operations on internal state. An infinite trace model has been used so as to ensure proper account is taken of the potentially unbounded non-determinism allowed by Z schemas.

Keywords: TCSP; Object-Z; TCOZ; Integrated Formal Methods

1. Introduction

Hoare [Hoa99] observes that formal methods are increasingly being used in effective combinations. Ultimately, the effective combination of formal methods can only be achieved if the semantic links between individual methods are clearly established and (consequently) the semantics of those methods are well integrated.

A recent research focus in formal methods is the integration of Z/Object-Z and CSP/CCS [Fis00, Smi97, GaS97, Suh99, SmD01, TaA97, MaD98]. One such approach is the blending of Object-Z [DuR00, Smi00] and TCSP [ScD95], called Timed Communicating Object-Z (TCOZ) [MaD98]. TCOZ builds on the respective strengths of both notations in order to provide a single elegant notation for modelling both state and process aspects of complex systems. The notion of blending Object-Z with CSP has been suggested independently by Fischer/Wehrheim [Fis00, FiW99] and Smith/Derrick [Smi97, SmD01]. TCOZ is novel in that it includes timing primitives, it supports the modelling of true multi-threaded concurrency, it clearly separates the design of control logic from algorithm design, it clearly separates the communications interface of classes from their internal structure, and it integrates the notions of operational refinement and process refinement into a single notion of class refinement. All these features are based on two general semantic links:

- Object-Z operation schemas are semantically identified with terminating CSP processes (that perform only state update events).

Correspondence and offprint requests to: Brendan Mahony, Information Technology Division, Defence Science and Technology Organisation of Australia, PO Box 1500, Edinburgh, South Australia 5111, Australia. E-mail: Brendan.Mahony@dsto.defence.gov.au or dongjs@comp.nus.edu.sg

- The behaviour of active objects is semantically identified with non-terminating CSP processes (multi-threading).

We consider these two semantic links are 'deeper' than say Smith/Derrick's approach [SmD01] which semantically identifies Object-Z operations with CSP channel/events, and classes with CSP processes. Although Smith/Derrick's 'shallow' links are less disruptive of the individual notations' syntax/semantics, the Object-Z classes are treated as 'modules'; and the powerful composition mechanism, class instantiation, is lost. On the other hand, TCOZ links TCSP constructs deeply into the Object-Z class structure (operations) so that complex operations and active object behaviour can be constructed by TCSP process expressions. Class constructs can encapsulate not only data/state but also TCSP processes. In one sense, TCOZ provides mechanisms to structure TCSP models.

This paper presents detailed discussions of the semantic links between Object-Z and TCSP. The paper also summarises the essential process semantics aspects of TCOZ that appeared in an early conference version [MaD99]. The TCOZ semantics model is presented in a similar fashion to Smith's abstract semantic model for Object-Z [Smi95b], which ignores the Object-Z reference semantics (by Griffith [GrR95]).

The support of timing primitives in TCOZ is made possible through the adoption of Reed's timed-failures semantics for TCSP [ReR88, ScD95]. The timed-failures semantics model CSP processes in terms of timed-event traces and timed-event failures. This semantic model allows CSP to be extended with time-related primitives such as delays, timeouts, and clock-interrupts. In order to support objects with encapsulated state this model is extended to include an initial state and state-update events. Object-Z operations are modelled as terminating sequences of timed state-update events. State-update events are similar to *signal* events, as described by Davies [Dav91], in that their occurrence is determined solely by the process; there is no requirement to model state-failures. Since Z allows unbounded non-determinism, it is also necessary to adopt the infinite trace variant of the timed failures model, as described by Mislove et al. [MRS95].

The outline of the paper is as follows. From the semantics point of view, Section 2 gives introductions and discussions on Object-Z and TCSP. Section 3 presents the integration of Object-Z and Timed-CSP – TCOZ. Section 4 presents the summary of the abstract syntax and the semantics. Section 4.3 presents the semantic toolkit used to describe the semantics of TCOZ. Section 4.4 presents the TCOZ semantics. Section 5 concludes the paper.

2. Overview of Object-Z and TCSP

The TCOZ language is a blending of the Object-Z and TCSP languages, making use of the class structuring and algorithmic and data design facilities of Object-Z and the process design facilities of TCSP. This section introduces the basic TCOZ language by first considering the complementary strengths and weaknesses of the Object-Z and TCSP languages. The simple example of a first-in-first-out (FIFO) queue is used as a pedagogical device for demonstrating the features of each language. Particular attention is paid to the semantic issues raised by language features discussed.

2.1. Object-Z and State Transition Model

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's state schema and its individual state transitions conform to individual operations of the class. An object is said to be an instance of a class and to evolve according to the definitions of its class.

Syntactically, a class definition is a named box, optionally with generic parameters. In this box the constituents of the class are defined and related. The main constituents are: a state schema, an initial state schema and a list of operation schemas.

ClassName[*generic parameters*]
state schema
initial state schema
operation schemas

The class structure makes use of the Z schema box construct in a hierarchy. In the general case, a schema box consists of a declaration part which describes the variables which the schema uses, called the *signature* of the schema, and predicate parts which describe some relationship which must be attained between the values of the signature variables. The interpretation of a Z schema is as a set of *bindings* of the signature variables to values, the set of such bindings for which the values of the variables satisfy the relationship described by the predicate part of the schema [Spi88]. Special syntactic conventions are used within the class schema to abbreviate and simplify the schema definitions which describe the class.

2.1.1. A Generic Queue Example

Consider the Object-Z specification of a generic FIFO queue. The queue structure may be represented as a class of objects, generic in the element type of the queue.

Queue[*X*]

$items : \mathrm{seq}\, X$
Δ
$h : X$

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

INIT
$items = \langle\, \rangle$

Join
$\Delta(items)$
$item? : X$

$items' = items \frown \langle item? \rangle$

Leave
$\Delta(items)$
$item! : X$

$items \neq \langle\, \rangle$
$item! = h$
$items' = \mathrm{tail}\, items$

The elements of the queue are modelled as a *primary* attribute (sequence of items), as described by the anonymous *state* schema. The head of the sequence is modelled as a *secondary* attribute, which appears below a Δ separator placed in the declaration section of the state schema. The secondary attributes are subject to change by every operation. 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.

The *initialisation* schema requires that the queue start operations with an empty list of items. The initialisation schema requires no declaration part because the state schema declaration is included by convention.

The rest of the class consists of a list of *operation* schemas. Two operations may be performed on queue objects: the *Join* operation adds an item to the back of the queue and the *Leave* operation removes an item from the head of the queue. Operation schemas include special conventions so that they may be interpreted as describing a relationship between the current and the next state of an object. Firstly, all the state variables of the object are included in the schema signature by default in each operation schema. Secondly, the Δ-list indicates the state variables for which a final value is calculated. Prime decorated variables with the identical base names are included in the schema signature for each variable in the Δ-list. The undecorated and primed state variables are used to describe properties of the current and next object states respectively. Finally, query and shriek decorated variables may be included to represent the values of any inputs and outputs, respectively, to the operation. Under all these conventions the schema may be seen to describe a relation between pairs of variable bindings, the first representing the current state and the inputs and the second representing the next state and the outputs.

In the general case an operation schema may not relate every possible current state to a next state (*partial definedness*) or else may relate some current states to more than one next state (*non-determinism*). In this sense the operation schema acts as a *specification* for a code routine which will be well-defined in the sense of identifying one (*total definedness*) and only one (*determinism*) next state for every current state. The process of transforming such a specification into a code routine is called *algorithmic refinement* and essentially corresponds to making the specification more defined and more deterministic.

2.1.2. Object-Z Semantic Issues

The standard behavioural semantics of Object-Z classes is presented as transition systems [Smi95b]. The transition system begins in a legal initial state and then evolves through a series of state transitions each effected by one of the class operations. The standard semantic model of transition system behaviour is a pair consisting of an initial state binding and a list of the operations invoked on the system in the order they are invoked. For example, the *Queue* object starts with *items* empty then evolves by successively performing either *Join* or *Leave* operations. A behaviour of this class may be modelled by a binding which gives *items* the value $\langle\ \rangle$ and a sequence recording the order in which the operations are performed.[1] The class as a whole may be modelled by the collection of behaviours allowed by the class.

If there is more than one possible behaviour for any given sequence of inputs, the class definition is said to be non-deterministic and may be viewed as a specification for a final program which must be deterministic. The process of transforming a class specification to a final program is again called refinement and in this case consists entirely of removing non-determinism.

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

Of greater interest from the semantic view point is the fact that this use of preconditions to control the sequencing of transitions is incompatible with standard algorithmic refinement. The process of removing non-determinism from operation specifications corresponds closely to that of removing non-determinism from the class specification, but the process of increasing the definedness of operation specifications will in the general case actually increase process non-determinism rather that reduce it. Thus algorithmic refinement of class operations can play havoc with the delicate interplay of preconditions in the original specification.

2.2. TCSP and Timed-Failures Model

TCSP [ScD95] extends the well-known CSP (Communicating Sequential Processes) notation of Hoare [Hoa85] with timing primitives. 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. TCSP 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 cooperative 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. The standard CSP semantics reflects this by modelling a process behaviour as a *failure* pair consisting of a sequence of events performed by the process (the *trace*) and a set of events that were subsequently offered by the environment but refused by the process (the *refusal*). A process is modelled by the collection of failures exhibited by the process.

The standard TCSP semantics enhances this failures semantics by recording the timing of each event in the trace and also recording a refusal set for every point in time. This powerful modelling technique is able to describe a range of real-time concepts such as delays, timeouts, and clock interrupts.

In the following, the core of the TCSP language is described informally in terms of this timed-failures model.

[1] Finer definition may be achieved by including information such as the inputs accepted, next state and outputs generated, and even the operations refused or enabled at each step.

2.2.1. Process Primitives

In TCSP, a distinguished event ✓ is used to represent and detect process termination.

A (timed prefix sequencing) process which may participate in event a at time t then wait for δ time followed by process P is written

$$a@t \overset{\delta}{\rightarrow} P(t). \quad \text{or} \quad a@t \rightarrow \textsc{Wait}\,\delta;\ P(t)$$

The event a is initially enabled and occurs as soon as it is also enabled by its environment; that is, it may not appear in the refusal set until it appears in the event trace, in which it must be the first event. 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. This is the time at which a occurs as recorded in the event trace. Subsequent behaviour, described by $P(t)$, may depend on the value of t. 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 $t + \delta$. During stabilisation the process must refuse all events and may not perform any. That is, in terms of $\textsc{Wait}\,t$, a failure of $\textsc{Wait}\,t$ refuses all events until time t and must then not refuse a ✓ event until it has performed a ✓. If omitted the event stabilisation time is 0 s by convention.

The process sequencing, sequential composition of P and Q, written $P;\ Q$, acts as P until P terminates by communicating ✓ and then proceeds to act as Q; the termination signal is hidden from the process environment. This is modelled by cutting each P failure at the first occurrence of ✓, removing the ✓, and concatenating a Q failure appropriately shifted in time. The process which may only terminate is written $\textsc{Skip}$.

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. Thus a failure of $P \,|[X]|\, Q$ is a superposition of failures, from P and Q respectively, which agrees exactly when restricted to X. Synchronisation on the empty event set is usually written $P \,|||\, Q$.

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 occurred, P after a and Q after b respectively. A failure of $P \,\Box\, Q$ is a failure of either P or Q such that there is failure in the other process with the same refusal set whose first event occurs no earlier. Whilst there is considerable scope for the introduction of nondeterminism with external choice, for example where both processes want to perform their first event simultaneously, the actual mechanism of external choice can be regarded as deterministic because the refusal set determines the event trace of the process. 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. The failures of $P \sqcap Q$ consist of the combination of the failures of the individual processes. Internal choices are non-deterministic because the refusal set does not determine the behaviour of the process. 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.

2.2.2. The Queue Example

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; the failures model has no way of representing internal state. Rather this intentional form of expression is a convenient way to provide a finite representation of an infinite family of process descriptions.

An example of the use of this convention is the definition of the FIFO queue in TCSP. The activities of joining and leaving the queue are represented by communications of the *left* and *right* respectively. Clearly any value output on the *right* must have first been received on the *left* channel and moreover they must be output in the order they were received, but this is a very hard condition to describe without making use of internal state. Consider the simple case where only one value is permitted in the queue. The initial queue is

$$P = \mathit{left}?a \rightarrow Q$$

after which either a new item may be added or the current one removed,

$$Q = \mathit{left}?a \rightarrow R \;\Box\; \mathit{right}!a \rightarrow P$$

and so on. Since there is no limit on the number of items that may be in the queue at one time, even this simple case has no finite representation without a convention for describing internal state.

The method used is to define a family of process names which consist of a base name decorated with an expression representing the current value of the internal state. Initially the queue is empty.

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

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

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

where t_j and t_l represent the time delay (duration) for the join and leave operations. Even for such a simple example TCSP has advantages 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 modelling in the form of mathematical toolkits and libraries nor modular techniques for 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 with a new type constant in place of X.

2.2.3. Timeout and Clock Interrupt

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

$$a \rightarrow P \triangleright\{t\}\, Q$$

will pass control to Q if the a event has not occurred by time t, as measured from the invocation of the process. A failure of $P \rhd\{t\}\ Q$ is a failure of P whose first event occurs before t or else consists of a failure for $\textsc{Wait}\ t\,;\ Q$.

The clock interrupt passes control from one process to another at a specified time. A failure of $P \swarrow\{t\}\ Q$ consists of a failure of P up to time t followed by a delayed failure of Q.

2.2.4. The Queue with Timeouts

Suppose that a queue process has the timing property 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. This timeouts queue can be described using delays and timeouts.

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

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

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:

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

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 occurring before the head of the queue becomes stale, the stale element is dropped:

$$\begin{array}{l}
TQ_{\langle(h,t)\rangle\frown tl} \mathrel{\widehat{=}} \\
\quad (left\,?a : X\ @t_i \rightarrow \textsc{Wait}\ t_j\,; \\
\qquad TQ_{ds(t_1+t_j,\langle(h,t)\rangle\frown tl)\frown\langle(a,C)\rangle} \ \Box \\
\quad right\,!h\,@t_i \rightarrow \textsc{Wait}\ t_l\,;\ TQ_{ds(t_1+t_l,tl)}) \\
\qquad \rhd\{t\}\ \textsc{Wait}\ t_d\,;\ TQ_{ds(t+t_d,tl)}
\end{array}$$

where t_d represents the time delay (duration) of the timeout drop operation. and the generic function *ds* (drop stale) is defined as

$$\begin{array}{|l}
[X] \\
\hline
ds : (\mathbb{T} \times \mathrm{seq}(X \times \mathbb{T})) \rightarrow \mathrm{seq}(X \times \mathbb{T}) \\
\hline
\forall t : \mathbb{T};\ s : \mathrm{seq}(X \times \mathbb{T}) \bullet \\
\qquad ds(t,s) = \mathrm{squash}\{(i,(e,t_o)) : s \mid t_o > t \bullet (i,(e,t_o - t))\} \\
\hline
\end{array}$$

The advantages and disadvantages of TCSP are thrown into even sharper focus by this example. TCSP 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, because the process control logic is nothing at all like the simple transition system interpretation. Note that this example does not even make use of the multi-threading and synchronisation capabilities of TCSP, 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 elegant treatment of the delay and timeout issues. Although, for example, Roscoe's CSP$_M$ language [Ros97] includes some powerful data modelling primitives, CSP still has no standard support for state modelling in the form of mathematical toolkits and libraries nor are there modular techniques for constructing and reasoning about complex internal state.

3. Integrating TCSP and Object-Z: TCOZ

TCSP and Object-Z complement each other in their expressiveness. Object-Z has strong data and algorithm modelling capabilities. The Z mathematical toolkit is extended with object-oriented structuring techniques. TCSP has strong process control modelling 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 associations 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 [He89], 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 modelling real-time and/or concurrent systems. Fischer [Fis00] and Smith [SmD01] 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 multi-threading 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 [Gal96], based on Z and (value-passing) CCS. Their Z operation schemas do not appear as events, but instead appear as prefixes to parameterised 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. Whilst this approach effectively disentangles the communication interface from the operational structure, the need to associate every occurrence 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-update events; to identify (active) classes with non-terminating CSP processes; and to allow arbitrary (channel-based) communications interfaces between objects.

The syntactic implication of this approach is 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. 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 multi-threaded computation even at the operation level, something that would not be possible with the 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. 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 TCSP primitives in describing the process aspects of an operation's behaviour. All operation definitions in TCOZ are in fact TCSP process definitions, with operation schemas being given the syntactic status of terminating Timed CSP processes.

As an example, consider the specification of the *Join* operation in the timed-queue example, assuming the state of the object is modelled as

$$\begin{array}{l} items : \mathrm{seq}(X \times \mathbb{T}) \\ (h, t) == head(items) \end{array}$$

The timing characteristics of the *Join* operation are expressed by the condition $\delta = t_j$ in the state-update *Add* operation.

Add

$$\begin{array}{l} \Delta(items) \\ item? : X \\ t_i? : \mathbb{T} \\ \hline \delta = t_j \wedge items' = ds(t_i? + t_j, items) \frown \langle (item?, C) \rangle \end{array}$$

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 *Join* lies in the adoption of CSP primitives in its definition. Item inputs are communicated to the *Join* operation along a channel *left*.

$$Join \mathrel{\widehat=} [item : X \,;\, t_i : \mathbb{T}] \bullet left?item@t_i \rightarrow Add$$

This definition of *Join* says that after the parameter *item* has been input on channel *left* at time t_i, the state-calculation *Add* is performed.

The local name space may be changed either by a local block definition as above or else by the occurrence 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 *Leave* operation, the communication of the deleting element must precede the updating of the queue object 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 behaviour cannot be described using an output parameter. Instead, the update operation is described as a simple state update which removes the head item (and any others that become stale).

Delete

$$\begin{array}{l} \Delta(items) \\ t_i? : \mathbb{T} \\ \hline items \neq \langle\, \rangle \wedge \delta = t_d \wedge items' = ds(t_i? + t_d, tails(items)) \end{array}$$

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

$$Leave \mathrel{\widehat=} [t_i : \mathbb{T} \mid items \neq \langle\, \rangle] \bullet right!h@t_i \rightarrow Delete$$

The first part of the definition of *Delete* is a (novel) process control primitive known as a *state guard*. Although if-then style commands appear in several dialects of CSP, for example CSP$_M$ [Ros97], we believe that TCOZ is unique in adopting the state guard as a separate primitive in the style of Morgan's version of the guarded command language [Mor94].

The adoption of a state guard mechanism allows TCOZ to have a clear 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 queue is non-empty. For other states this process will *deadlock* or *block*, refusing any communication.

Note that the precondition requirement in the *Delete* 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 leads to unspecified subsequent behaviour. *Delete* 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 of 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).

3.2. Processes

A schema expression describes a relationship on or between process state(s), whilst a process expression describes the overall behaviour 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 non-determinism. TCSP 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 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. Firstly, a variable binding is added to represent the initial values of all the process attributes. Secondly, a new kind of event, referred to as an *update* event, 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 ✓ event occurs), then the state at the time of termination is called the *final* state. Finally, since the unbounded non-determinism 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. [MRS95], is adopted.

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. In the timed-failures model, divergence is represented by allowing arbitrary behaviour from the time of divergence.

The process model for the state guard consists of replacing the trace part of every behaviour 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.

Some existing Object-Z schema calculus operators, such as $\_ \Box \_$, $\_ \parallel \_$, and $\_;\ \_$, have namesakes with similar semantics in the CSP process calculus. The convention adopted in TCOZ is that the CSP operator is intended, 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 $\_ \parallel \_$ and $\_ ||| \_$, the designer is exposed to all the usual dangers of shared variable concurrency. The operation $OS_1 \parallel OS_2$, where OS_1 and OS_2 are operation schema, will synchronise on all state-update events on variables in the respective delta-lists. Thus $OS_1 \parallel OS_2$ will have much the same process properties as $OS_1 \bigwedge OS_2$, 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. We 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 utilised when properly protected by the object encapsulation mechanism.

3.3. Active Behaviour

The distinguished process name MAIN in a class indicates that the objects of the class have *active* behaviour after the initialisation. Initialisation 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. 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-queue example, the existence of environmental obligations and the need to drop stale elements means that the timed-queue class must have its own thread of control. Assuming that the class operations are defined as in Section 3.1, the timed-queue behaviour can be defined by a MAIN process similar to the TCSP version presented in Section 2.2.4. We will see this MAIN process defined below in the full class definition.

3.4. Channels Together

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. Contrary to the conventions adopted for internal state variables, channels are viewed as shared 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 further enhancing the modularity of system specifications.

Consider once again the timed-queue example: in addition to the list of *items*, the state schema must declare channels *left* and *right*, and the entire class definition of the timed-queue can be presented as:

$TimedQ\,[X]$

$items : \mathrm{seq}(X \times \mathbb{T})$
$left, right : \mathrm{chan}$
Δ
$(h, t) : X \times \mathbb{T}$
$items \neq \langle\,\rangle \Rightarrow (h, t) = head(items)$

$\textsc{Init}$
$items = \langle\,\rangle$

Add
$\Delta(items)$
$item? : X$
$t_i? : \mathbb{T}$
$\delta = t_j \wedge items' = ds(t_i? + t_j, items) \frown \langle (item?, C) \rangle$

$Delete$
$\Delta(items)$
$t_i? : \mathbb{T}$
$items \neq \langle\,\rangle \wedge \delta = t_d \wedge items' = ds(t_i? + t_d, tails(items))$

$Join \mathrel{\widehat{=}} [item : X\,;\ t_i : \mathbb{T}] \bullet left?item@t_i \rightarrow Add$
$Leave \mathrel{\widehat{=}} [t_i : \mathbb{T} \mid items \neq \langle\,\rangle] \bullet right!h@t_i \rightarrow Delete$
$Drop \mathrel{\widehat{=}} [\Delta(items) \mid \delta = t_d \wedge items' = ds(t + t_d, items)]$
$\textsc{Main} \mathrel{\widehat{=}} \mu\, TQ \bullet [items = \langle\,\rangle] \bullet Join;\ TQ\ \Box$
$\qquad [items \neq \langle\,\rangle] \bullet ((Join \,\Box\, Leave) \rhd \{t\}\ Drop);\ TQ$

This model represents a more concise, flexible, and scalable treatment of both process and state than is possible in either Object-Z or TCSP. Process's internal state and communications interfaces are tightly coupled with the behaviour part of the model in a single class construct. 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 TCSP 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.

4. Summary of TCOZ Abstract Syntax and Semantics

This section summarises the abstract syntax and semantics of TCOZ (for detailed discussion/explanation, see [MaD99]).

4.1. Abstract Syntax

Firstly, the abstract syntax of Z schemas and expressions is presented by the given sets $[ZS, ZE]$.

The abstract syntax of TCOZ process expressions is given in the form of a free type definition, in a style similar to that adopted by Schneider and Davies [ScD95] with extra state-related primitives.

In the following let $[\Sigma]$ represent the set of all possible communications channels:

$$
\begin{array}{ll}
TZE ::= & ref\langle\langle NAME\rangle\rangle \mid \textsc{Stop} \mid \textsc{Chaos} \mid \textsc{Skip} \mid \textsc{Wait}\,\langle\langle ZE\rangle\rangle \mid \\
& op\langle\langle ZS\rangle\rangle \mid (\_\bullet\_)\langle\langle ZS \times TZE\rangle\rangle \mid \\
& (\_\rightarrow\_)\langle\langle \Sigma \times TZE\rangle\rangle \mid (\_.\_\rightarrow\_)\langle\langle \Sigma \times ZE \times TZE\rangle\rangle \mid \\
& (\_\sqcap\_)\langle\langle TZE \times TZE\rangle\rangle \mid (\_\Box\_)\langle\langle TZE \times TZE\rangle\rangle \mid \\
& (\_\parallel\_)\langle\langle TZE \times TZE\rangle\rangle \mid (\_|[\_]|\_)\langle\langle TZE \times \mathbb{P}\Sigma \times TZE\rangle\rangle \mid \\
& (\_|||\_)\langle\langle TZE \times TZE\rangle\rangle \mid (\_;\_)\langle\langle TZE \times TZE\rangle\rangle \mid \\
& (\_[\_/\_])\langle\langle TZE \times \Sigma \times \Sigma\rangle\rangle \mid (\_\backslash\_)\langle\langle TZE \times \mathbb{P}\Sigma \mid \\
& (\mu\_\bullet\_)\langle\langle NAME \times TZE\rangle\rangle
\end{array}
$$

where $[ZS, ZE]$ represents the abstract syntax of Z schemas and expressions and $[\Sigma]$ represents the set of all possible communications channels.

The body of a TCOZ class is essentially a system of simultaneous equations defining (possibly recursively) a collection of (stateful) operations and processes. Each equation consists of a name $[NAME]$ and a TCOZ expression. The collection of names referenced in an expression, *tze*, is called the process signature of *tze* and written $\mathrm{sig}\,tze$. The process signature of an expression may be determined by a straightforward recursive search of the expression structure:

$$
\begin{array}{l}
TZB == \\
\quad \{\mathscr{C} : NAME \twoheadrightarrow TZE \mid \\
\qquad \forall\, tze : \mathrm{ran}\,\mathscr{C};\ N : NAME \bullet \\
\qquad\quad N \in \mathrm{sig}\,tze \Rightarrow N \in \mathrm{dom}\,\mathscr{C}\}
\end{array}
$$

In TCOZ, classes fall into two categories: passive and active. A passive class consists of an initialisation and a class body, while an active TCOZ class definition contains an additional main process expression, which determines the overall behaviour of the class.

$$
\begin{array}{|l}
\hline TZC_p \\
init : ZS \\
\mathscr{C} : TZB \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline TZC_a \\
TZC_p \\
main : TZE \\
\hline
\mathrm{sig}\,main \subseteq \mathrm{dom}\,\mathscr{C} \\
\hline
\end{array}
$$

4.2. TCOZ Semantic Models Overview

The state of an active TCOZ object is fully encapsulated; the only way to interact with an active object is via its communications interface. Active objects are modelled as pure (non-terminating) CSP processes, using the basic infinite timed-failures semantics.

Passive classes are treated essentially as specification libraries. They have no process semantics and all of their internal components (attributes and operations) may be accessed directly. The operations defined in classes (both active and passive) have explicit state. This is modelled by extending the infinite timed-failures model with an initial state component and by introducing a special class of events which update system state.

Thus we utilise two semantic models: the basic infinite timed-failures model and the novel infinite timed-states model.

4.3. Semantic Toolkit

In this section the mathematical models used in describing the TCOZ semantics are developed. The data-related aspects of the language are modelled using state bindings and the process-related aspects of the language are modelled using event traces and refusals.

The notation of the basic Z mathematical toolkit [Spi88] is used to develop the various models, with the exception that an enhanced convention for treating schemas and schema bindings as first-class objects is adopted. This builds on the enhancements to the Z toolkit suggested by Valentine [Val95]. This convention explicitly recognises schema types, such as $[a, b : \mathbb{N}]$, as sets of variable bindings.

4.3.1. Schemas and State Bindings

In this section, the notions of state variables, bindings, and schemas are modelled and the interpretation of schemas as state guards, initialisations, and operations briefly described. These notions have been treated in considerable detail by the Z community so the development presented here is deliberately abstract, describing only the facets of Z's state semantics which are of direct relevance to the novel aspects of the semantics of TCOZ.

Variables are modelled as

$$\mathsf{VAR} ::= (\_)\langle\!\langle \Pi \rangle\!\rangle \mid (\_')\langle\!\langle \Pi \rangle\!\rangle \mid (\_?)\langle\!\langle \Pi \rangle\!\rangle \mid (\_!)\langle\!\langle \Pi \rangle\!\rangle \mid \delta$$

where $[\Pi]$ denotes the collection of all legal Object-Z state attribute (pre, post, input and output) base names and a distinguished variable δ used to describe operation timing.

The notational convenience of writing, V for $(\_)(\!|V|\!)$, V' for $(\_')(\!|V|\!)$, etc., is adopted throughout this paper. A collection of variables is called a *signature*.

The collection of value constants is represented by the abstract set [VAL]. The syntax of Z (and hence of TCOZ) is designed so as to ensure that VAL may always be represented by a proper set [Spi88], built up inductively from the given sets of a Z document.

A state binding is a partial function from variables to values:

$$Bind \mathrel{\widehat{=}} \mathsf{VAR} \mathrel{+\!\!\!\!\!\rightarrow} \mathsf{VAL}$$

The domain of a binding is called its signature.

The semantics of Z expressions is modelled using a record consisting of a signature and a binding from states to values:

Exp
$$\begin{array}{|l}
\Gamma : \mathbb{P}\,\mathsf{VAR} \\
v : Bind \mathrel{+\!\!\!\!\!\rightarrow} \mathsf{VAL} \\
\hline
\operatorname{dom} v = \{\gamma : Bind \mid \operatorname{dom}\gamma \supseteq \Gamma\}
\end{array}$$

The semantic function on Z expressions is represented by $\mathscr{F}_Z[\![\_]\!]$; that is, the meaning of the expression *ze* is $\mathscr{F}_Z[\![ze]\!]$.

The semantic space for Z schemas consists of a signature Γ and a collection of states ϕ, each with signature Γ:

Schema
$$\begin{array}{|l}
\Gamma : \mathbb{P}\,\mathsf{VAR} \\
\phi : \mathbb{P}\,Bind \\
\hline
\forall \gamma : \phi \bullet \operatorname{dom}\gamma = \Gamma
\end{array}$$

The semantic mapping on Z schemas is represented by $\mathscr{F}_Z[\![\_]\!]$.

4.3.2. Operations

An operation is represented by an initial-state signature Γ_i, a final-state signature Γ_f, and a collection of transitions (known as the *transition relation*, which consists of a pair of Γ states, the *initial state* and the *final* state, and a duration):

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\,Opr\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Gamma_i : \mathbb{P}\,\Pi \\
\Gamma_f : \mathbb{P}\,\Pi \\
X : \mathbb{P}\,Act \\
\hline
\forall \chi : X \bullet \\
\quad \mathrm{dom}\,\chi.i = \Gamma_i \land \\
\quad \mathrm{dom}\,\chi.f = \Gamma_f \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\_\,Act\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
i : \Pi \rightarrow\!\!\!\!\!\!+\;\, \mathsf{VAL} \\
f : \Pi \rightarrow\!\!\!\!\!\!+\;\, \mathsf{VAL} \\
d : Time \\
\hline
\end{array}$$

The *precondition* of the operation is the collection of initial states which are associated with a final state:

$$\mathrm{pre}\,\_ == \lambda\, O : Opr \bullet i(\!|O.X|\!)$$

An operation is said to be *non-deterministic* when there is some initial state which is related to more than one final state. An operation O_2 is said to be a *refinement* of another operation O_1, written $O_1 \sqsubseteq O_2$, when they have the same signature but O_2 has a weaker precondition and is less non-deterministic than O_1:

$$\begin{array}{l}
(\_ \sqsubseteq \_) == \\
\quad \{O_1, O_2 : Opr \mid \\
\qquad O_1.\Gamma_i = O_2.\Gamma_i \land O_1.\Gamma_f = O_2.\Gamma_f \land \mathrm{pre}\,O_1 \subseteq \mathrm{pre}\,O_2 \land \\
\qquad \forall \chi : O_2.X \bullet \chi.i \in \mathrm{pre}\,O_1 \Rightarrow \chi \in O_1.X\}
\end{array}$$

A state guard is essentially a predicate on the state variables. It consists of a signature and a collection of states.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\_\,Grd\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Gamma : \mathbb{P}\,\Pi \\
\phi : \mathbb{P}\,Bind \\
\hline
\forall \gamma : \phi \bullet \mathrm{dom}\,\gamma = \Gamma \\
\hline
\end{array}$$

Again a state guard is expressed using a schema. The semantic function on guards, *mk_Grd*, is a simple injection.

$$mk\_Grd == \{Schema; Grd \bullet \theta Schema \mapsto \theta Grd\}$$

A state initialisation *Init* is similarly defined.

4.3.3. Events, Traces and Refusals

In this section a toolkit for describing infinite timed failures is developed. The general notational approach of Mislove et al. [MRS95] is followed fairly closely, though the Z syntactic conventions are exploited fully.

A TCOZ *event* may be either an update event, a simple synchronisation, a channel communication, or a termination event.

$$\begin{array}{l}
U == [v : \Pi;\ d : \mathsf{VAL}] \\
S == [c : \Sigma] \\
C == [c : \Sigma;\ d : \mathsf{VAL}] \\
Event == update\langle\!\langle U \rangle\!\rangle \mid sync\langle\!\langle S \rangle\!\rangle \mid com\langle\!\langle C \rangle\!\rangle \mid \checkmark
\end{array}$$

Updates record the variable changed and the new value. Synchronisations simply record the channel used, while channel communications also record a data value:

$$\begin{array}{l}
\mathbb{U} == update(\!|U|\!) \\
\mathbb{S} == syn(\!|S|\!) \\
\mathbb{C} == com(\!|C|\!)
\end{array}$$

The update events are distinguished from the other events in that they do not require the cooperation of the environment. The events which do require the cooperation of the environment are called environment events:

$$\mathbb{E} == \mathbb{S} \cup \mathbb{C} \cup \{\checkmark\}$$

Finally we overload the field selectors of the schema types U, S, C to act on events in the obvious way. For example, $update(\langle\!|v \mathrel{\widehat{=}} v, d \mathrel{\widehat{=}} \alpha|\!\rangle).v = v$.

A timed *trace* is a (possibly infinite) sequence of time stamped events. The time stamps on the records in a timed trace must be non-decreasing:

$$\begin{array}{l} Stamp == [\tau : Time;\ \sigma : Event] \\ Trace == \{t : \mathrm{seq}_\infty\, Stamp \mid \\ \qquad\qquad \forall\, n, m : \mathrm{dom}\, t \bullet n \leqslant m \Rightarrow t(n).\tau \leqslant t(m).\tau\} \end{array}$$

A finite trace is a trace with finite elements:

$$Fin\_Trace == \{t : Trace \mid \exists\, n : \mathbb{N} \bullet \mathrm{dom}\, t = 0 \mathinner{..} n\}$$

The *begin* time of a (non-empty) trace is the time stamp of its first element:

$$\mathrm{begin} == \lambda\, t : Trace \bullet (\mathrm{head}\, t).\tau$$

The *end* time of a (non-empty) finite trace is the time stamp of its last element:

$$\mathrm{end} == \lambda\, t : Fin\_Trace \bullet (\mathrm{last}\, t).\tau$$

A trace may be delayed in time using the delay operator:

$$del == \lambda\, t : Trace;\ \delta : \mathbb{R} \bullet \lambda\, i : \mathrm{dom}(t) \bullet \langle\!|\tau \mathrel{\widehat{=}} t(i).\tau + \delta, \sigma \mathrel{\widehat{=}} t(i).\sigma|\!\rangle$$

A trace may be shifted forward in time by composing it with a negative delay and restricting it to positive times:

$$(\_ \mathbin{\dot{-}} \_) == \lambda\, t : Trace;\ D : Time \bullet del(t, -D) \upharpoonright_\tau Time$$

where the traces filtering function is defined as

$$\begin{array}{|l} [Y] \\ \hline \_ \upharpoonright\!\_\ \_ : Trace \times (Stamp \rightarrowtail\!\!\!\!\to Y) \times \mathbb{P}\, Y \to Trace \\ \hline \forall\, t : Trace;\ \pi : Stamp \rightarrowtail\!\!\!\!\to Y;\ F : \mathbb{P}\, Y \bullet \\ \qquad t \upharpoonright_\pi F = \mathrm{squash}\{n : \mathbb{N};\ s : Stamp \mid s \in \pi^{-1}(\!|F|\!)\} \\ \hline \end{array}$$

The final state constructed by a finite trace is the final value assigned to each of the variables updated by the trace:

$$\begin{array}{l} \mathit{final} == \lambda\, t : Fin\_Trace \bullet \\ \qquad \{v : \Pi;\ d : \mathsf{VAL} \mid \\ \qquad\qquad \exists\, n : \mathrm{dom}\, t \bullet t(n).\sigma = update(\theta U) \wedge \\ \qquad\qquad\qquad \forall\, m : \mathrm{dom}\, t \mid m > n \bullet t(m).\sigma.v \neq v\} \end{array}$$

Refusals represent the failure of a process to engage in events offered by the environment. We define them in the usual way [ScD95, MRS95]. The basic timed refusal set or *refusal token* ($RTOK$) consists of a collection of (environment) events refused for some (half-open) interval of time ($RINT$). Update events do not appear in refusals, since they do not require the cooperation of the environment. A *finite* refusal ($RSET$) consists of a finite number of refusal tokens. An infinite refusal ($IRSET$) consists of any collection of refusal tokens which is finite up to any finite time:

$$\begin{array}{l} RINT == \{\alpha, \omega : Time \bullet [\tau : Time \mid \tau \in [\alpha \ldots \omega)]\} \\ RTOK == \{I : RINT \bullet I \times [\sigma : \mathbb{E}]\} \\ RSET == \{R : \mathbb{F}\, RTOK \bullet \bigcup R\} \\ IRSET == \{R : \mathbb{P}\, RTOK \mid (\forall\, D : Time \bullet (\bigcup R \upharpoonright_\tau [0 \ldots D)) \in RSET) \bullet \bigcup R\} \end{array}$$

4.3.4. *The Infinite Timed-States Model for TCOZ*

TCOZ operations and processes are interpreted in the *infinite timed-states model*. This model extends the infinite timed-failures model for TCSP [MRS95], to allow treatment of local state. On the other hand, since the local state of active objects is encapsulated, they are interpreted in the basic infinite timed-failures model. The use of an infinite trace model is mandated by the presence of unbounded non-determinism in Z schemas, but it does introduce some complications in the treatment of recursion which have been discussed in detail in [MaD99] (and omitted in this paper).

The denotation of each TCOZ process $\mathscr{B}_{TIS}$ consists of a collection of behaviours. Each such behaviour consists of an initial state, an event trace, and a timed refusal. A basic infinite timed failure $\mathscr{B}_{TIF}$ does not contain an initial state and restricts the trace to environment events:

$$\begin{array}{|l}\hline \mathscr{B}_{TIS} \\ \hline \imath : \Pi \to \mathsf{VAL} \\ s : \mathit{Trace} \\ \aleph : \mathit{RSET} \\ \hline \end{array} \qquad \begin{array}{|l}\hline \mathscr{B}_{TIF} \\ \hline s : \mathit{Trace} \\ \aleph : \mathit{RSET} \\ \hline s = s \restriction_{\sigma} \mathbb{E} \\ \hline \end{array}$$

The infinite timed-failures model consists of a collection of infinite timed-failures behaviours which satisfy the five healthiness criteria, $\mathscr{H}$, described by Mislove *et al* [MRS95]:

$$\begin{array}{ll} \mathscr{M}_{TIF} == \{S : \mathbb{P}\,\mathscr{B}_{TIF} \mid & \\ \quad \mathscr{H}(S) & \text{[Healthiness criteria]} \\ \quad \} & \end{array}$$

Processes in the infinite timed-states model consist of $\mathscr{B}_{TIS}$ behaviours which satisfy the $\mathscr{M}_{TIF}$ healthiness criteria for each possible initial state:

$$\mathscr{M}_{TIS} == \{S : \mathbb{P}\,\mathscr{B}_{TIS} \mid \forall\, i : \Pi \to \mathsf{VAL} \bullet \mathscr{H}(\{\mathscr{B}_{TIS} \mid \imath = i \bullet \theta\mathscr{B}_{TIF}\})\}$$

The filter operator is extended to act on $\mathscr{M}_{TIF}$ and $\mathscr{M}_{TIS}$ in the obvious way.

We define refinement in the infinite-timed states model to be reverse subset inclusion on processes.

4.4. The Infinite Timed-States Semantics for TCOZ

In this section a semantics is presented for the TCOZ abstract syntax presented in Section 4.1. The semantic operator for the infinite timed-states model is represented by $\mathscr{F}_{TIS}\,[\![\_]\!]$.

4.4.1. *The Semantics of Ground Terms*

The first step in developing the semantics is to give a meaning to each of the ground constructors of the TCOZ abstract syntax.

The Stop process deadlocks immediately:

$$\mathscr{F}_{TIS}\,[\![\mathrm{S{\scriptsize TOP}}]\!] == \{\mathscr{B}_{TIS} \mid s = \langle\,\rangle \bullet \theta\mathscr{B}_{TIS}\}$$

The Wait process terminates after a set time. For $t : \mathit{Time}$,

$$\begin{array}{l} \mathscr{F}_{TIS}\,[\![\mathrm{W{\scriptsize AIT}}\,\_]\!](t) == \\ \quad \{\mathscr{B}_{TIS} \mid s = \langle\,\rangle \wedge \checkmark \notin \sigma(\!|\aleph \restriction_{\tau} [\,t \ldots \infty)\,|\!) \bullet \theta\mathscr{B}_{TIS}\} \cup \\ \quad \{\mathscr{B}_{TIS}\,;\ t' : \mathit{Time} \mid t' \geqslant t \wedge s = \langle (\!| \tau \mathrel{\widehat{=}} t', \sigma \mathrel{\widehat{=}} \checkmark |\!) \rangle \wedge \\ \qquad \checkmark \notin \sigma(\!|\aleph \restriction_{\tau} [\,t \ldots t')\,|\!) \bullet \theta\mathscr{B}_{TIS}\} \end{array}$$

An operation performs a sequence of state updates so as to ensure a desired relationship between initial and final states, then terminates. If the initial state does not satisfy the precondition, the process diverges. For $O : \mathit{Opr}$,

$$\begin{array}{l} \mathscr{F}_{TIS}\,[\![\mathit{op}]\!](O) == \\ \quad \{\mathscr{B}_{TIS}\,;\ \mathit{Act} \mid \theta\mathit{Act} \in O.X \wedge \end{array}$$

$$\begin{aligned}
&\quad O.\Gamma_i \lhd \iota \in \mathrm{pre}\, O \land s = s \upharpoonright_{\sigma;v} O.\Gamma_f \land \mathrm{end}\,\aleph \leqslant d \land \mathrm{end}\, s \leqslant d \\
&\quad \bullet \theta\mathscr{B}_{TIS}\} \cup \\
&\{\mathscr{B}_{TIS}\,;\ Act \mid \theta Act \in O.X\ \land \\
&\quad O.\Gamma_i \lhd \iota \in \mathrm{pre}\, O \land s = s \upharpoonright_{\sigma;v} O.\Gamma_f \land \mathrm{end}\,\aleph > d \land \mathrm{end}\, s = d\ \land \\
&\quad f = \mathit{final}(s) \land \checkmark \notin \sigma(\!|\aleph \upharpoonright_\tau [\!-d \ldots \infty -\!]\,|\!) \bullet \theta\mathscr{B}_{TIS}\} \cup \\
&\{\mathscr{B}_{TIS}\,;\ s_u : \mathit{Trace};\ t : \mathit{Time};\ Act \mid \theta Act \in O.X\ \land \\
&\quad \Gamma_i \lhd \iota \in \mathrm{pre}\, O \land f = \mathit{final}(s_u) \land s_u = s_u \upharpoonright_{\sigma;v} O.\Gamma_f\ \land \\
&\quad \mathrm{end}\, s_u = d \land t \geqslant d\ \land \\
&\quad s = s_u \frown \langle \langle\!| \tau \mathrel{\widehat=} t, \sigma \mathrel{\widehat=} \checkmark |\!\rangle \rangle \land \checkmark \notin \sigma(\!|\aleph \upharpoonright_\tau [\!-d \ldots t -\!]\,|\!) \\
&\quad \bullet \theta\mathscr{B}_{TIS}\} \cup \\
&\{\mathscr{B}_{TIS} \mid \Gamma_i \lhd \iota \notin \mathrm{pre}\, O \bullet \theta\mathscr{B}_{TIS}\}
\end{aligned}$$

Theorem 4.1 The Z operation $O_1 : Opr$ is refined by $O_2 : Opr$ if and only if $\mathscr{F}_{TIS}\,[\![op]\!](O_1)$ is refined by $\mathscr{F}_{TIS}\,[\![op]\!](O_2)$.

Proof. Assume that $O_1 \sqsubseteq O_2$ and let $b : \mathscr{B}_{TIS}$ be such that $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_2)$ and let $i == O_2.\Gamma_i \lhd b.\iota$. If $i \notin \mathrm{pre}\, O_2$, then also $i \notin \mathrm{pre}\, O_1$ and hence $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_1)$. If $i \in \mathrm{pre}\, O_2 \setminus \mathrm{pre}\, O_1$, then $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_1)$ trivially.

Now suppose that $i \in \mathrm{pre}\, O_1$. If $b.s$ is non-terminating and there exists $\chi : O_2.X$ such that $\mathrm{end}\, b.\aleph \leqslant \chi.d$, then since $\chi.d$ is also an execution time allowed by O_1, $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_1)$.

Finally, suppose there exists $\chi : O_2.X$ such that $\mathrm{end}\, b.\aleph \geqslant \chi.d$, $\chi.i = O_2.\Gamma_i \lhd \iota$, and $\mathit{final}\ b.s = \chi.f$. If $b.s$ is non-terminating and $\checkmark$ is not refused in $b.\aleph$ from time $\chi.d$, then $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_1)$. If $b.s$ is terminating and $\checkmark$ is not refused in $b.\aleph$ from time $\chi.d$ before it occurs, then again $b \in \mathscr{F}_{TIS}\,[\![op]\!](O_1)$.

The reverse implication is similar. □

State guarding causes a process to deadlock if its initial state fails the guard. For $G : Grd$, $P : \mathscr{M}_{TIS}$,

$$\begin{aligned}
&\mathscr{F}_{TIS}\,[\![(\_ \bullet \_)]\!](G, P) = \\
&\quad \{\mathscr{B}_{TIS} \mid G.\Gamma \lhd \iota \in G.\phi\theta\mathscr{B}_{TIS} \in P \bullet \theta\mathscr{B}_{TIS}\} \cup \\
&\quad \{\mathscr{B}_{TIS} \mid G.\Gamma \lhd \iota \notin G.\phi \land s = \langle\ \rangle \bullet \theta\mathscr{B}_{TIS}\}
\end{aligned}$$

The details of the semantics of other ground terms can be found in [MaD99].

4.4.2. The Semantics of Passive Classes

The semantics of a passive class is modelled as a binding from names to infinite timed-state processes, referred to as *process bindings*. Where only ground terms appear in defining equations, this binding could be determined directly from the ground term semantics. However, if name references are to appear, it is necessary to adopt a fixed point approach to determining the semantics.

The first step in this process is to interpret a TCOZ expression relative to a given binding of its signature. That is, expressions are modelled as mappings from process bindings to processes. Suppose that $tze : TZE$ is a TCOZ expression and that $\beta : NAME \twoheadrightarrow \mathscr{M}_{TIS}$ is a process binding such that $\mathrm{dom}\, tze \subseteq \mathrm{sig}\, \beta$. The semantics of tze relative to β, written $\mathscr{F}_{TIS}\,[\![tze]\!]_\beta$, is defined by recursion on the structure of tze:

$$\begin{aligned}
&\mathscr{F}_{TIS}\,[\![ref(N)]\!]_\beta = \beta(N) \\
&\mathscr{F}_{TIS}\,[\![\mathrm{S{\small TOP}}]\!]_\beta = \mathscr{F}_{TIS}\,[\![\mathrm{S{\small TOP}}]\!] \\
&\mathscr{F}_{TIS}\,[\![\mathrm{W{\small AIT}}\, te]\!]_\beta = \mathscr{F}_{TIS}\,[\![\mathrm{W{\small AIT}}\,\_]\!](\mathscr{F}_Z\,[\![te]\!]) \\
&\mathscr{F}_{TIS}\,[\![op(OS)]\!]_\beta = \mathscr{F}_{TIS}\,[\![op]\!](mk\_Opr(\mathscr{F}_Z\,[\![OS]\!])) \\
&\mathscr{F}_{TIS}\,[\![GS \bullet P]\!]_\beta = \mathscr{F}_{TIS}\,[\![\_ \bullet \_]\!](mk\_Grd(\mathscr{F}_Z\,[\![GS]\!]), \mathscr{F}_{TIS}\,[\![P]\!]_\beta)
\end{aligned}$$

... others omitted

$$\mathscr{F}_{TIS}\,[\![\mu N \bullet P]\!]_\beta = \mathit{fix}(\lambda Q : \mathscr{M}_{TIS} \bullet \mathscr{F}_{TIS}\,[\![P]\!]_{\beta \oplus \{N \mapsto Q\}})$$

The semantics of a collection of process definitions is then the fixed point of the corresponding mapping from process bindings to process bindings:

$$\mathscr{F}_{TIS}\ [\![\mathscr{C}]\!] = \mathit{fix}(\mathscr{D}_{TISB}, \lambda\,\beta : \mathrm{sig}\,\mathscr{C} \rightarrow \mathscr{M}_{TIS} \bullet \lambda N : \mathrm{sig}\,\mathscr{C} \bullet \mathscr{F}_{TIS}\ [\![\mathscr{C}(N)]\!]_{\beta})$$

4.4.3. The Semantics of Active Classes

The state initialisation operator converts a stateful process into a pure process by hiding the initial state and the state update events. The stateful process is also restricted to those behaviours with initial states that satisfy the initialisation predicate:

For I : INIT and $P : \mathscr{M}_{TIS}$,

$$\begin{aligned}
&\mathscr{F}_{TIS}\ [\![\mathit{init}]\!](I, P) == \\
&\quad \{\mathscr{B}_{TIF}\,;\ (\mathscr{B}_{TIS})_P \mid \\
&\qquad \theta(\mathscr{B}_{TIS})_P \in P \wedge I.\Gamma \lhd \iota_P \in I.\phi \wedge s = s_P \restriction_{\sigma} \mathbb{E} \wedge \aleph = \aleph_P \\
&\qquad \bullet\ \theta\mathscr{B}_{TIF}\}
\end{aligned}$$

The semantics of an active class is the main process evaluated in the binding defined by the class definitions, with the local state initialised by the class initialisation:

$$\mathscr{F}_{TIF}\ [\![C]\!] = \mathscr{F}_{TIS}\ [\![mk\_Init(\mathscr{F}_{z}\,[\![C.init]\!])]\!](\mathscr{F}_{TIS}\ [\![C.main]\!]_{\mathscr{F}_{TIS}\ [\![C.\mathscr{C}]\!]})$$

5. Conclusion

In TCOZ, the key semantic links between Object-Z and TCSP are the identifications between operation schemas with terminating CSP processes and active objects with non-terminating CSP processes. This leads to the flexible use of CSP process expressions deep in Object-Z class constructs.

This paper presents detailed discussions of the semantic links between Object-Z and TCSP. The paper also summarises the essential process semantics aspects of TCOZ which appeared in an early version [MaD99]. Other aspects of the TCOZ semantics, such as the object-reference semantics and data refinement, are not discussed and are the subject of further work.

The process model used by TCOZ consists of sets of tuples consisting of: an initial state; a (possibly infinite) trace (a sequence of time-stamped events, including update-events), and a (possibly infinite) refusal (a record of what and when events are refused by the process). This represents a conservative extension to the basic infinite timed-failures model of Mislove et al. [MRS95] which allows us to retain the same basic model of recursion. An infinite trace model has been used so as to ensure proper account is taken of the potentially unbounded non-determinism allowed by Z schemas.

TCOZ differs from other approaches to blending Object-Z with a process algebra in that it does not identify operations with events. Instead a fine-grained collection of (abstract) 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 (terminating) CSP process. It therefore makes sense also to identify their syntactic classes. TCOZ allows operations to be defined either using the schema calculus or the process calculus and it allows operation schemas to appear as processes in CSP expressions.

Finally we note that Theorem 4.1 means that the TCOZ semantics of operations ensures TCSP process refinement agrees with Z operation refinement. This means that the operation refinement and structured specification techniques of standard Z are more applicable to TCOZ than to either Object-Z or any blended notation of which we are aware. A subject of further work is the investigation of data refinement in the TCOZ setting. Since the TCOZ model hides all update events, we expect it to be straightforward to apply data-refinement techniques in the TCOZ setting. Another interesting further research work perhaps would be to develop proof rules for reasoning about TCOZ based on the Object-Z's logic [Smi95a].

Acknowledgements

We would like to thank Hugh Anderson for his helpful comments on this work. This work is supported in part by the DSTO/CSIRO Fellowship programme from Australia and the research grant (Integrated Formal Methods, RP3991615) from the National University of Singapore.

References

[AGT99] Araki, K., Galloway, A. and Taguchi, K. editors: *IFM'99: Integrated Formal Methods*, York, UK. Springer-Verlag, June 1999.

[Dav91] Davies, J.: *Specification and Proof in Real-Time Systems*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1991.

[DuR00] Duke, R. and Rose, G.: *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.

[Fis00] Fischer, C.: *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, University of Oldenburg, Gemany, January 2000.

[FiW99] Fischer, C. and Wehrheim, H.: Model-checking CSP-OZ specifications with FDR. In Araki et al. [AGT99].

[Gal96] Galloway, A. J.: *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems*. PhD thesis, University of Teesside, School of Computing and Mathematics, August 1996.

[GrR95] Griffiths, A. and Rose, G.: A semantic foundation for object identity in formal specification. *Object-Oriented Systems*, 2:195–215, 1995.

[GaS97] Galloway, A. J. and Stoddart, W. J.: An operational semantics for ZCCS. In Hinchey and Liu [HiL97], pages 272–282.

[He89] He, J.: Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[HiL97] Hinchey, M. and Liu, S. editors: *In IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hoa99] Hoare, C. A. R.: Formal methods observations. Conference overview. In *World Congress on Formal Methods, FM'99*, http://www.cert.fr/fm99/conferen.htm, 1999.

[MaD98] Mahony, B. P. and Dong, J. S.: Blending Object-Z and Timed CSP: an introduction to TCOZ. In K. Futatsugi, R. Kemmerer, and K. Torii, editors, *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Kyoto, Japan, April 1998. IEEE Press.

[MaD99] Mahony, B. P. and Dong, J. S.: Overview of the semantics of TCOZ. In Araki et al. [AGT99], pages 66–85.

[Mor94] Morgan, C. C.: *Programming from Specifications*, second edition. Prentice-Hall, 1994.

[MRS95] Mislove, M., Roscoe, A. and Schneider, S.: Fixed points without completeness. *Theoretical Computer Science*, 138:273–314, 1995.

[Ros97] Roscoe, A. W.: *Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1997.

[ReR88] Reed, G. and Roscoe, A.: A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[ScD95] Schneider, S. and Davies, J.: A brief history of Timed CSP. *Theoretical Computer Science*, 138, 1995.

[SmD01] Smith, G. and Derrick, J.: Specification, refinement and verification of current systems: an integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[Smi95a] Smith, G.: Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

[Smi95b] Smith, G.: A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[Smi97] Smith, G.: A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of FME'97: Industrial Benefit of Formal Methods*, Graz, Austria, September 1997. Springer-Verlag.

[Smi00] Smith, G.: *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.

[Spi88] Spivey, J. M.: *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, UK, 1988.

[Suh99] Suhl, C.: RT-Z: an integration of Z and timed CSP. In Araki et al. [AGT99].

[TaA97] Taguchi, K. and Araki, K.: The state-based CCS semantics for concurrent Z specification. In Hinchey and Liu [HiL97], pages 283–292.

[Val95] Valentine, S. H.: Equal rights for schemas in Z. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, number 967 in Lecture Notes in Computer Science, pages 203–223, Limerick, Ireland, September 1995. Springer-Verlag.

Received September 2000
Accepted in revised form June 2001