Formal Methods in System Design, 18, 249–284, 2001

Specification, Refinement and Verification of Concurrent Systems—An Integration of Object-Z and CSP

GRAEME SMITH smith@it.uq.edu.au
Software Verification Research Centre, University of Queensland 4072, Australia

JOHN DERRICK j.derrick@ukc.ac.uk
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK

Received March 30, 1998; Accepted March 27, 2000

Abstract. This paper presents a method of formally specifying, refining and verifying concurrent systems which uses the object-oriented state-based specification language Object-Z together with the process algebra CSP. Object-Z provides a convenient way of modelling complex data structures needed to define the component processes of such systems, and CSP enables the concise specification of process interactions. The basis of the integration is a semantics of Object-Z classes identical to that of CSP processes. This allows classes specified in Object-Z to be used directly within the CSP part of the specification.

In addition to specification, we also discuss refinement and verification in this model. The common semantic basis enables a unified method of refinement to be used, based upon CSP refinement. To enable state-based techniques to be used for the Object-Z components of a specification we develop state-based refinement relations which are sound and complete with respect to CSP refinement. In addition, a verification method for static and dynamic properties is presented. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes by using the laws of the CSP operators together with the logic for Object-Z.

Keywords: Object-Z, CSP, specification, refinement, verification, concurrency

1. Introduction

A primary purpose of formal specification is to provide concise and easily comprehensible descriptions of software systems. For particularly large or complex systems, this goal may be more readily achieved by using more than one specification language. While most specification languages can be used to specify entire systems, few, if any, are particularly suited to modelling all aspects of such systems.

A good example of where such a combination of languages is particularly useful is the specification of concurrent or distributed systems. Such systems comprise a number of distinct component *processes* operating concurrently and synchronising on certain events. Process algebras such as CCS [29] and CSP [21, 30] are suitable vehicles for modelling the interactions between processes or their temporal ordering. State-based languages such as Z [41] or VDM [23], however, offer better facilities for the specification of the complex data structures which may be needed to describe the processes themselves. Indeed,

modern distributed systems architectures (such as the Open Distributed Processing reference model [22]) partition a specification into a number of related *viewpoints*, and recognise that different languages are likely to be used in the different viewpoint specifications of a large distributed system.

This realisation has lead to the development of new specification languages which combine features of one or more existing languages [3, 16] and, more recently, approaches for formally integrating existing languages [4, 9, 12, 17, 20, 27, 45].

For example, the RAISE specification language (RSL) [16] designed for use with complex industrial-scale systems involving concurrency includes features drawn from both VDM and CSP. The use of state-based languages together with process algebras has also been proposed to support the use of viewpoints in ODP where Z and LOTOS [3][1] are candidates for use in the information and computational viewpoints respectively.

An advantage of the use of separate viewpoints in ODP over that of RAISE is that, rather than defining a new language, existing languages can be used without altering their syntax or semantics. This makes the approach more accessible to users who are already familiar with the existing languages and also enables the use of tools and methods of verification and refinement developed for these languages. A disadvantage of the ODP approach, however, is that it produces, rather than a single specification, a number of related specifications which must be checked for consistency [2].

In order to produce a single specification while using a combination of existing languages, constructs defined in one of the languages need to be applicable in the other enabling the various parts of the specification to be linked. This is possible if such constructs are given a semantics identical to that of an identifiable construct in the other.

One problem with adopting such an approach, however, is the lack of a construct in most state-based specification languages identifiable with processes in a process algebra. This is not true of object-oriented state-based specification languages such as Object-Z [14, 39], an object-oriented extension of Z. Central to object orientation is the view of a system as a collection of distinct, interacting objects whose state and operations are encapsulated in *classes*. Hence, there is a strong relationship between classes in object-oriented systems and processes in concurrent systems: interactions between instances of each define the system behaviour. This relationship has been recognised by many researchers in both the theory and practice of object orientation [15, 37, 44, 49].

This paper presents a semantic integration of Object-Z and CSP based on the relationship between classes and processes, and thereby enables a method of formally specifying concurrent systems using Object-Z together with CSP to be described. Object-Z provides a convenient method of modelling the complex data structures needed to define component processes, and CSP enables a concise specification of process interaction. The semantic integration enables classes specified in Object-Z to be used directly within the CSP part of the specification.

The approach we describe consists of three phases. The first phase involves specifying the component processes as Object-Z classes. To maintain a separation of concerns and allow maximum flexibility in describing the component processes, each class is described independently of the others and of the environment in which they are to be placed. The components thus specified will generally not be in a form that allows them to be composed

using CSP operators. The second phase involves modifying the class interfaces (by using Object-Z inheritance and CSP renaming) so that they will synchronise and communicate as desired. Finally, the specification of the whole system is given using CSP operators to describe how the components interact.

In addition to *specification*, a notation needs to be able to support incremental *development* of specifications through a well-defined method of refinement. It is also desirable to be able to verify both static and dynamic, i.e. behavioural, properties of these specifications. The work described here presents a method of refining specifications written in the integrated Object-Z/CSP notation, and a method for verifying such properties of those specifications.

Having a common semantic basis for the two languages enables a unified method of refinement to be developed for the integrated notation: because we give Object-Z classes a CSP semantics, we can use CSP refinement as the refinement relation for their combination. However, as a means for verifying a refinement it is more convenient to be able to use a state-based refinement relation for the Object-Z components, rather than having to calculate their semantics. In order to do so, we adapt the work of Josephs [24], who has developed refinement relations for state-based systems which are sound and complete with respect to CSP refinement.

In order to be able to verify static and dynamic properties, we present a method of verification for the integrated notation. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes by using the laws of the CSP operators presented in [21] together with the logic for Object-Z described in [35]. CSP and Object-Z properties are related via auxiliary variables introduced into the Object-Z classes using inheritance.

Although the emphasis in this paper is on the application of the approach to concurrent and distributed systems, its applicability extends to other domains. For example, it has been applied to the specification of interactive systems [26]. Also, both Z and CSP have been advocated for specifying different aspects of software architectures [34] and combinations of Z and Object-Z with timed CSP have been suggested for the specification of embedded systems [20, 28, 42].

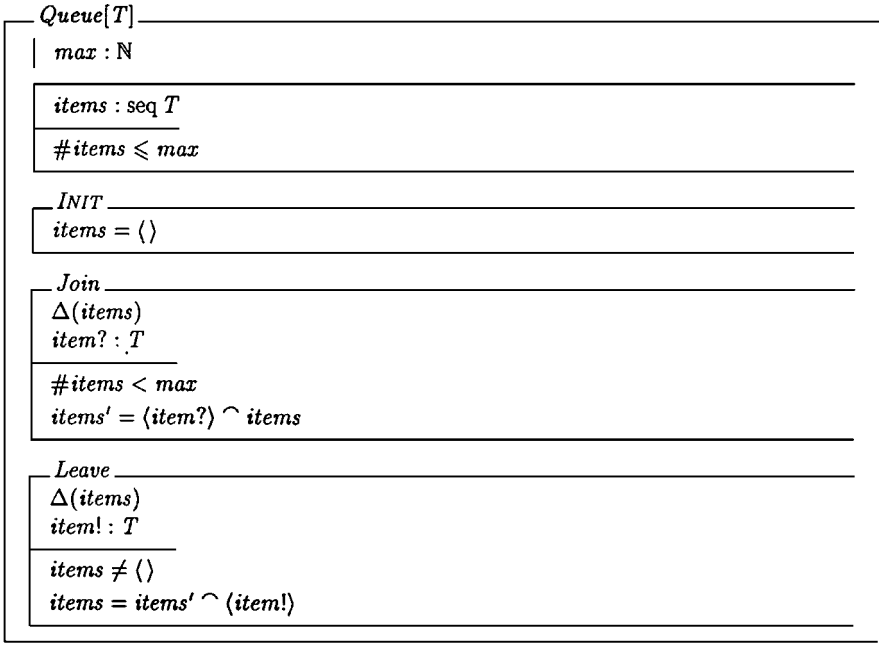
The paper is structured as follows. Sections 2 and 3 introduce Object-Z and CSP respectively by providing simple examples of their use and an overview of their semantics. Section 4 presents the semantic integration of the languages. Classes in Object-Z are given a failures-divergences [6] semantics identical to that of CSP processes. In Section 5 the approach to specification using the integrated notations is illustrated through a simple case study of a cinema booking system. Section 6 then discusses refinement in the integrated notation, and defines the state-based refinement relations that we will use for the Object-Z components of a specification. Section 7 explains how properties of specifications can be verified, and we conclude in Section 8.

2. Object-Z

Object-Z [14, 39] is an extension of Z designed to support an object-oriented specification style. It includes a special class construct to encapsulate a state schema with all the operation schemas which may affect its variables. A class may be used to define one or more

components of a system or to specify the interactions between components by referring to instances, i.e. *objects*, of their classes. In the approach adopted in this paper, the interactions between components will be specified using CSP. We restrict our attention, therefore, to classes which do not refer to objects of other classes.

A class in Object-Z is represented syntactically by a named box possibly with generic parameters. In this box there may be local type and constant definitions, at most one state schema and associated initial state schema, and zero or more operation schemas. As an example, consider the following specification of a bounded queue. This specification is generic in that the type T, of the items in the queue, is not specified.

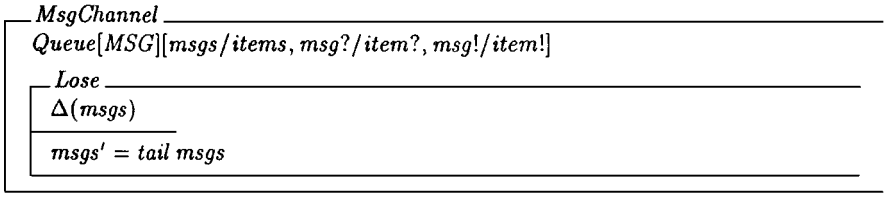


The class has a single constant *max* denoting the maximum length of the queue and a single state variable *items* denoting the items in the queue. Constants are associated with a fixed value which cannot be changed by any operations of the class. However, the value of constants may differ for different objects of the class. Initially the queue is empty and the operations *Join* and *Leave* enable items to join and leave the queue, respectively, on a first-in/first-out basis. Each operation schema has a Δ-list of state variables which it may change, a declaration part consisting of input (denoted by names ending in ?) and output (denoted by names ending in !) parameters and a predicate part relating the pre- and post-values (denoted by names ending in $'$) of the state variables.

A class may also *inherit* the definitions of one or more other classes. Inheritance is a powerful mechanism for incremental specification allowing peripheral concerns to be postponed while specifying the intrinsic behaviour of a class of objects. An inherited class's local types and constants are implicitly available in the inheriting class. Its schemas are

also either implicitly available or are implicitly conjoined with common-named schemas declared in the inheriting class. As an example, consider the following definition of a lossy message channel.

Let *MSG* denote the set of all possible messages.



The class *MsgChannel* inherits *Queue* with its generic type T instantiated with *MSG* and with the state variable *items* renamed to *msgs* and the operation parameters *item*? and *item*! renamed to *msg*? and *msg*! respectively. In general, constants, state variables, operation schemas and operation parameters may be renamed. Objects of *MsgChannel* behave identically to those of *Queue* up to renaming, except that they have an additional operation *Lose* corresponding to the loss of a message.

2.1. Semantics of Object-Z classes

A class in Object-Z can be modelled as a set of values each corresponding to a potential object of the class at some stage of its evolution. Such a semantics is presented in [37] where, following the work of [13], the value chosen to represent an object is the sequence of states the object has passed through together with the corresponding sequence of operations the object has undergone. This value is referred to as the *history* of the object.

To define the structure and properties of the histories of a class, we first need to define what is meant by its states and the operations it can undergo. The states of a class assign values to the state variables and any constants the state schema may refer to. This includes both constants defined in the class and those defined globally.

Given the set of all possible identifiers, i.e. strings of characters denoting names, *Ident* and the set of all possible values *Value*, the states of a class can be represented by a set[2]

$$S \subseteq (Ident \pfun Value)$$

such that the following property holds.

$$st_1 \in S \land st_2 \in S \Rightarrow \mathrm{dom}\ st_1 = \mathrm{dom}\ st_2 \qquad (S1)$$

That is, the states of a class refer to a common set of variables.

The operations a class can undergo are instances of the class's operation schemas. They can be represented by the name of an operation schema together with an assignment of values to its parameters.

The operations of a class can be represented by a set

$$O \subseteq Ident \times (Ident \pfun Value)$$

such that the following property holds.

$$(n, p_1) \in O \land (n, p_2) \in O \Rightarrow \text{dom } p_1 = \text{dom } p_2 \qquad (O1)$$

That is, an operation name is always associated with the same set of parameters.

A history is a non-empty sequence of states together with a sequence of operations. Either both sequences are infinite[3] or the state sequence is one longer than the operation sequence. The histories of a class with states S and operations O can be represented by a set[4]

$$H \subseteq S^\omega \times O^\omega$$

such that the following properties hold.

$$(s, o) \in H \Rightarrow s \neq \langle\,\rangle \qquad (H1)$$

$$(s, o) \in H \land s \notin S^* \Rightarrow o \notin O^* \qquad (H2)$$

$$(s, o) \in H \land s \in S^* \Rightarrow \#s = \#o + 1 \qquad (H3)$$

$$(s_1 \frown s_2, o_1 \frown o_2) \in H \land \#s_1 = \#o_1 + 1 \Rightarrow (s_1, o_1) \in H \qquad (H4)$$

The first three properties capture the requirements on an individual history detailed above. The final property is a condition on the set of histories representing a class. This set must be *prefix-closed*. This is necessary since the first state in the sequence of states satisfies the class's initial state, and each pair of consecutive states is a possible state transition of the operation whose position in the operation sequence is the same as that of the first state of the pair in the state sequence. Therefore, any prefix of an object's history is the history of that object at some earlier stage of its evolution and hence represents a possible history of the object's class.

3. CSP

CSP has been designed specifically to specify concurrent systems. It models a system as a collection of processes which run concurrently, communicate over unbuffered channels and synchronise on particular events. Processes are specified by guarded, and usually recursive, equations. For example, a simple one-place buffer can be specified as follows.

$$\alpha(BUFFER1) = \{in.n \mid n \in \mathbb{N}\} \cup \{out.n \mid n \in \mathbb{N}\}$$
$$BUFFER1 = in?x \rightarrow out!x \rightarrow BUFFER1$$

The first line of the specification defines the *alphabet* of the process $BUFFER1$. The alphabet is the set of all events that the process can possibly engage in. Each event, in

this case, is of the form $c.v$ where c is the name of a channel and v is a value communicated along that channel. The second line of the specification defines the temporal order in which the process undergoes those events. The notation $in?x$ corresponds to an event $in.x$ where x is input on channel in. Similarly, $out!x$ corresponds to an event $out.x$ where x is output on channel out.

To compose processes, CSP has a number of operators. One of the most important of these is the concurrency operator[5] $\|$. When two processes are combined using this operator, they run concurrently and synchronise on events with the same name or, in the case of communications events, the same channel name and value. All other events are interleaved in the resulting process. For example, a two-place buffer can be specified in terms of two one-place buffers which are concurrently composed such that the $out.x$ event of the first synchronises and communicates with the $in.x$ of the second. To do this we use the substitution notation where $P[\![a/b]\!]$ means that the event or channel b in process P is replaced by a.

The two-place buffer is then specified as follows.

$$BUFFER2 = (BUFFER1[\![transfer/out]\!] \parallel BUFFER1[\![transfer/in]\!]) \setminus \{transfer.n \mid n \in \mathbb{N}\}$$

The events $transfer.n$, where $n \in \mathbb{N}$, are *hidden* in the resulting process so that these events are not available to the environment for further synchronisation.

In addition to the concurrency operator $\|$, CSP also provides an interleaving operator $|||$. When two processes are combined using this operator all events are interleaved in the resulting process (i.e. there is no synchronisation).

Although CSP processes are specified without reference to an explicit state, to simplify specifications, they often have parameters which simulate the state information. For example, consider the following equations which specify a queue of arbitrary length.

$$\begin{aligned}
&\alpha(Queue) = \{in.n \mid n \in \mathbb{N}\} \cup \{out.n \mid n \in \mathbb{N}\} \\
&Queue = Queue_{<>} \\
&Queue_{<>} = in.x \rightarrow Queue_{<x>} \\
&Queue_{s\hat{}<x>} = in.y \rightarrow Queue_{<y>\hat{}s\hat{}<x>} \\
&\qquad\qquad \Box \\
&\qquad\qquad out.x \rightarrow Queue_s
\end{aligned}$$

The subscript parameters represent the sequence of items in the queue. Parameters can also appear in brackets following the process name.

3.1. Semantics of CSP processes

The standard semantics of CSP is the failures-divergences semantics developed in [5, 6]. A process is modelled by the triple (A, F, D) where A is its alphabet, F is its *failures* and D is its *divergences*. The failures of a process are pairs (t, X) where t is a finite

sequence of events that the process may undergo and X is a set of events the process may refuse to perform after undergoing t. That is, if the process after undergoing t is in an environment which only allows it to undergo events in X, it may deadlock. The divergences of a process are the sequences of events after which the process may undergo an infinite sequence of internal events, i.e. livelock. Divergences also result from unguarded recursion.

Failures and divergences are defined in terms of the events in the alphabet of the class. The failures of a process with alphabet A are a set

$$F \subseteq A^* \times \mathbb{P}A$$

such that the following properties hold.

$$(\langle\,\rangle, \varnothing) \in F \tag{$F1$}$$
$$(t_1 \frown t_2, \varnothing) \in F \Rightarrow (t_1, \varnothing) \in F \tag{$F2$}$$
$$(t, X) \in F \wedge Y \subseteq X \Rightarrow (t, Y) \in F \tag{$F3$}$$
$$(t, X) \in F \wedge (\forall e \in Y \bullet (t \frown \langle e \rangle, \varnothing) \notin F) \Rightarrow (t, X \cup Y) \in F \tag{$F4$}$$

Notice that we have dropped the restriction in [5] that the set of refused events is finite as is also done in [6][6] and [24].

Properties $F1$ and $F2$ capture the requirement that the sequences of events a process can undergo form a non-empty, prefix-closed set. Property $F3$ states that if a process can refuse all events in a set X then it can refuse all events in any subset of X. Property $F4$ states that a process can refuse any event which cannot occur as the next event.

The divergences of a process with alphabet A and failures F are a set

$$D \subseteq A^*$$

such that the following properties hold.

$$D \subseteq \operatorname{dom} F \tag{$D1$}$$
$$t_1 \in D \wedge t_2 \in A^* \Rightarrow t_1 \frown t_2 \in D \tag{$D2$}$$
$$t \in D \wedge X \subseteq A \Rightarrow (t, X) \in F \tag{$D3$}$$

The first property simply states that a divergence is a possible sequence of events of the process. Properties $D2$ and $D3$ capture the idea that it is impossible to determine anything about a divergent process in a finite time. Therefore, the possibility that it might undergo further events cannot be ruled out. In other words, a divergent process behaves *chaotically*.

4. Modelling classes as processes

As discussed in the introduction, there is a strong relationship between object-oriented and concurrent systems. More precisely, the notion of class corresponds closely to that of

process. In this section, we use this correspondence as the basis for a semantic integration of Object-Z and CSP: Object-Z classes are given a failures-divergences semantics identical to that of CSP processes. This allows classes defined in the Object-Z part of the specification to be used directly in the CSP part.

4.1. Operations and events

In order to relate classes and processes, we need a relationship between operations and events. Since both represent observable, atomic actions,[7] we adopt the approach of simply identifying them. This needs to be done in such a way that appropriate input and output parameters of synchronising operations can be identified. We therefore define a meta-function β which returns the basename of a parameter name, i.e. $\beta(x?) = \beta(x!) = x$, and allow it to be applied to the assignment of values to an operation's parameters as follows.

$$\beta(\{(x_1, v_1), \ldots, (x_n, v_n)\}) = \{(\beta(x_1), v_1), \ldots, (\beta(x_n), v_n)\}$$
$$\text{where } \{x_1, \ldots, x_n\} \subseteq \mathit{Ident} \text{ and } \{v_1, \ldots, v_n\} \subseteq \mathit{Value}$$

The function relating operations and events is then defined as follows.

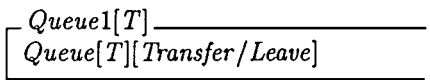
$$\mathit{event}((n, p)) = n.\beta(p) \quad \text{where } n \in \mathit{Ident} \text{ and } p \in (\mathit{Ident} \twoheadrightarrow \mathit{Value})$$

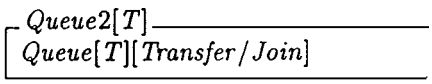
The event corresponding to an operation (n, p) is a communication event with the operation name n as the channel and the mapping from the basenames of its parameters to their values $\beta(p)$ as the value 'passed' on that channel. For example, the event corresponding to joining a value x on to a *Queue* object is $\mathit{Join}.\{(\mathit{item}, x)\}$. In this event *Join* is a CSP channel, *item* is a member of *Ident* and x is a metavariable denoting a value of type T.

This allows operations of different classes to interact in the following three ways.

- An output parameter $x!$ can be equated with an input parameter $x?$ in a synchronising operation.

  This type of interaction is the most common and models message passing communication between processes. For example, to join two queues so that the values output by one are input by the other, we concurrently compose the following classes which inherit the class *Queue* of Section 2.





  The operations *Leave* of *Queue*1 and *Join* of *Queue*2 are renamed to *Transfer* to allow them to synchronise as required. Communication is achieved by the identification of the output *item*! of *Queue*2 with the input *item*? of *Queue*2.
- An input parameter $x?$ can be equated with an input parameter $x?$ in a synchronising operation.

This type of interaction models sharing of an input value. For example, two message channels which concurrently accept broadcast messages can be specified by composing the following classes.

*MsgChannel*1

$$MsgChannel[Leave1/Leave, Lose1/Lose]$$

*MsgChannel*2

$$MsgChannel[Leave2/Leave, Lose2/Lose]$$

In this case, the *Leave* and *Lose* operations of each class are renamed to prevent them from synchronising. The *Join* operations synchronise and the sharing of inputs is achieved by identifying their *item*? inputs.

- An output parameter $x!$ can be equated with an output parameter $x!$ in a synchronising operation.

  This type of interaction models cooperation of two processes to produce an output. It is used when we wish to abstract away from the actual cooperation mechanism which in general would require additional message passing. For example, two exchanges in a mobile phone network may cooperate to output a frequency which neither are currently using for calls. This can be specified as follows.

  Let *Freq* be the set of all frequencies.

*Exchange*1

$used\_freqs : \mathbb{P}\, Freq$

...

Available

$freq! : Freq \setminus used\_freqs$

*Exchange*2

$used\_freqs : \mathbb{P}\, Freq$

...

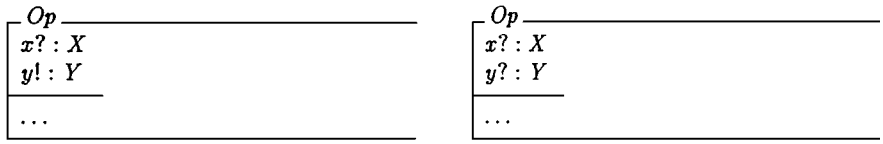
Available

$freq! : Freq \setminus used\_freqs$

The exchanges synchronise on the operation *Available* and the necessary output is achieved by identifying their *freq*! outputs.

Note that when the values of the outputs of one or both of the operations are specified nondeterministically (as in the above example), it cannot be guaranteed that they will have the same output values and that the operations will synchronise. The choice of output values is internal to the class instance and not constrained by the environment. Hence, the specifier should be aware of the possibility of deadlock.

In general, our approach is to allow operations with both input and output parameters. Any two operations with the same name and parameters with identical basenames will be modelled by identical events when their parameters have the same values and hence will be able to synchronise. For example, the following operations could synchronise.



Particular attention must be given, however, to operations which have an input and output parameter with the same basename. When these parameters have the same value, they will be identified in the the set $\beta(p)$ where p is the assignment of values to the operation's parameters, and hence the operation can synchronise with an operation with just one parameter with this basename. When the parameters have different values, any synchronising operation would necessarily have both a corresponding input and output parameter. In this case, however, the inputs of each operation could be equated either to each other or to the outputs of the other operation. Since such specifications could easily lead to misunderstandings, input and output parameters with common basenames should be avoided in operations as a matter of style.

Our use of communications events differs from the conventional usage suggested in [21]. A channel in our approach is a means of bidirectional transfer of multiple messages between processes.[8] Conventionally, however, channels are used for unidirectional transfer of a single message. If desired, conventional usage of channels can be achieved by restricting the individual Object-Z operations to have only inputs or only outputs. In this case, each input, or output, of the operation can be regarded as the field of an input, or output, message to be passed on the channel identified by the operation name.

While identifying operations and events seems an appropriate choice, it should be noted that there are in fact other options. For example, Benjamin [1] suggests identifying operation parameters and events in order to integrate Z and CSP. This leads to a conventional use of channels in communications events, i.e. channels are unidirectional and used to transfer single messages, however it can also lead to more complex specifications. Ensuring the correct synchronisation when an operation has several parameters can be difficult when each parameter is treated as a separate event. Combinations of Z and Object-Z with timed CSP have identified operation invocations and terminations with events [20, 42] and operations with processes [27].

4.2. Classes and processes

The essence of the approach presented in this paper is that each Object-Z class can be referred to as a process in the CSP part of the specification. The process representing a class must describe the behaviour of all possible objects of that class. However, it is often convenient to refer to the behaviour of particular objects corresponding to particular values of the class's constants. Since constants and state variables are not distinguished in the history semantics of Object-Z, we do not wish to distinguish them here. That is, we do not want to add to the existing semantics of Object-Z. We allow, therefore, the value of state variables as well as constants in a class's initial state to be referred to when the class is used as a process.

Corresponding to a class C is a set of parameterised processes C_i. The parameter i is an assignment of values to a subset of the state of C satisfying a possible initial state of C. That is, $i \in \{j \mid \exists(s,\ o) \in H \bullet j \subseteq s(1)\}$.[9] The subset i is chosen to reflect the assignment of initial values corresponding to the particular objects required by the specification. For example, $MsgChannel_{\{(\max,10)\}}$ refers to a message channel of length 10.

Classes with generic parameters are referenced with the actual parameters in brackets following the class name. For example, $Queue_{\varnothing}(\mathbb{N})$ refers to a queue of natural numbers. The $\varnothing$ subscript in this case denotes the fact that there is no restriction on the initial state of the class. That is, the process represents the behaviours of all possible objects of the class. For notational convenience, we introduce the convention that $C = C_{\varnothing}$ write simply $Queue(\mathbb{N})$.

Given a class C with states S, operations O and histories H, the alphabet of process C_i comprises the events corresponding to the operations in O.

$$alphabet(C_i) = \{event(op) \mid op \in O\}$$

To define the failures of a class we use the following function which maps a sequence of operations to a sequence of events.

$$events(\langle\,\rangle) = \langle\,\rangle$$
$$events(\langle op\rangle \frown o) = \langle event(op)\rangle \frown events(o)$$

We also introduce a meta-function ι which returns the assignment of values to the inputs of an operation's parameters.

$$\iota(\{(x_1!,\ v_1), \ldots,\ (x_n!, v_n), (y_1?, u_1), \ldots, (y_m?, u_m)\}) = \{(y_1?, u_1), \ldots, (y_m?, u_m)\}$$

The failures of C_i are derived from the histories in H as follows: $(t,\ X)$ is a failure of C_i if

- there exists a finite history of C whose initial state is satisfied by i,
- the sequence of operations of the history corresponds to the sequence of events in t, and
- for each event in X, either
  - there does not exist a history which extends the original history by an operation corresponding to the event, or
  - there exists a history which extends the original history by an operation corresponding to a second event which has the same operation name and assignment of values to input parameters and is not in X.

The final condition on the set X models the fact that the outputs of an operation cannot be constrained by the environment: a class instance may refuse all but one of the possible assignment of values to the output parameters corresponding to a particular operation and assignment of values to its input parameters. This enables the choice of values for output parameters to be resolved during refinement.

$$
\begin{aligned}
\mathit{failures}(C_i) = \{(t, X) \mid \exists\, (s, o) \in H \bullet\ & \\
& s \in S^* \wedge \\
& i \subseteq \mathrm{s}(1) \wedge \\
& t = \mathit{events}(\mathrm{o}) \wedge \\
& (\forall\, \mathrm{e} \in X \bullet \\
& \quad (\nexists\, st \in S, op \in O \bullet \\
& \qquad e = \mathit{event}(op) \wedge (s \frown \langle\, st\rangle, o \frown \langle op\rangle) \in H) \\
& \quad \vee \\
& \quad (\exists\, (n, p) \in O, (n, q) \in O \bullet \\
& \qquad \iota(p) = \iota(q) \wedge \\
& \qquad e = \mathit{event}((n, p)) \wedge \\
& \qquad (\exists\, st \in S \bullet (s \frown \langle st\rangle, o \frown \langle (n, q)\rangle)) \in H) \\
& \qquad \mathit{event}(n, q) \notin X)\}
\end{aligned}
$$

It is necessary to show that the set of failures of C_i satisfy the properties $F1$ to $F4$ of Section 3.

The properties $F1$ and $F2$ follow from the fact that the set of histories is prefix-closed (property $H4$ of Section 2).

Proof of $F1$**:** Since C_i is only defined for i where $\exists (s, o) \in H \bullet i \subseteq s(1)$ and $\#\langle s(1)\rangle = \#\langle\,\rangle + 1$, $(\langle s(1)\rangle, \langle\,\rangle) \in H$ by $H4$. Since $\langle s(1)\rangle \in S^*$ and $\mathit{events}(\langle\,\rangle) = \langle\,\rangle$ and $\forall e \in \varnothing \bullet P$ for any predicate P, $(\langle\,\rangle, \varnothing) \in \mathit{failures}(C_i)$. □

Proof of $F2$**:** If $(t_1 \frown t_2, \varnothing) \in \mathit{failures}(C_i)$ then $\exists (s, o) \in H \bullet i \subseteq s(1) \wedge t_1 \frown t_2 = \mathit{events}(o)$. If $s = s_1 \frown s_2$ and $o = o_1 \frown o_2$ such that $\#o_1 = \#t_1$ and $\#s_1 = \#o_1 + 1$ then $(s_1, o_1) \in H$ by $H4$. Since $s_1(1) = s(1)$, $i \subseteq s_1(1)$ and since $s_1 \in S^*$ and $\mathit{events}(o_1) = t_1$ and $\forall e \in \varnothing \bullet P$ for any predicate P, $(t_1, \varnothing) \in \mathit{failures}(C_i)$. □

The properties $F3$ and $F4$ follow directly from the definition of the function *failures*.

Proof of $F3$**:** Since $(\forall e \in X \bullet P) \Rightarrow (\forall e \in Y \bullet P)$ for any predicate P when $Y \subseteq X$, if $(t, X) \in \mathit{failures}(C_i)$ and $Y \subseteq X$ then $(t, Y) \in \mathit{failures}(C_i)$. □

Proof of $F4$**:** If $\forall e \in Y \bullet (t \frown \langle e\rangle, \varnothing) \notin \mathit{failures}(C_i)$ then, since $\forall e \in \varnothing \bullet P$ for any predicate P, $\forall e \in Y \bullet \nexists (s, o) \in H \bullet i \subseteq s(1) \wedge s \in S^* \wedge t \frown \langle e\rangle = \mathit{events}(o)$. Therefore, given $(s, o) \in H$ such that $i \subseteq s(1)$ and $s \in S^*$ and $t = \mathit{events}(o)$, $\forall e \in Y \bullet \nexists st \in S, op \in O \bullet e = \mathit{event}(op) \wedge (s \frown \langle st\rangle, o \frown \langle op\rangle) \in H$. Hence, if $(t, X) \in \mathit{failures}(C_i)$ then $(t, X \cup Y) \in \mathit{failures}(C_i)$. □

Since Object-Z does not allow hiding of operations, divergence is not possible. The set of divergences of C_i are hence defined as follows.

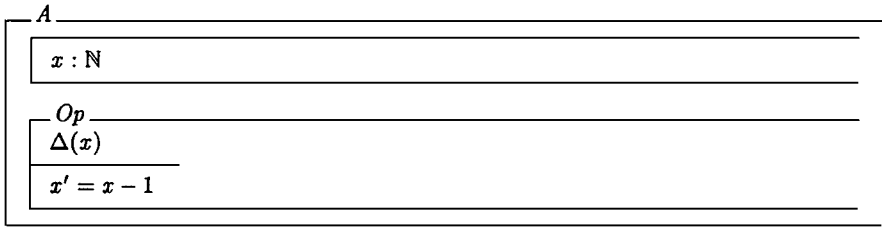
$$\mathit{divergences}(C_i) = \varnothing$$

This definition trivially satisfies the properties $D1$ to $D3$ of Section 3.

Alternatively, divergences could be used to indicate chaotic behaviour which occurs when an operation is applied outside its precondition as in Z. However, our approach is consistent with Object-Z where operations are "blocked", i.e. cannot occur, outside their preconditions (see [37]).

4.3. Unbounded nondeterminism

The failures-divergences semantics of CSP does not support the specification of *unbounded nondeterminism* (i.e. where a process can choose from an infinite set of options). For example, it does not allow the specification of a process which nondeterministically selects any natural number n and then performs a particular event n times. More precisely, because it only uses finite traces to model a process, it cannot distinguish between a process which can undergo any finite sequence of an event a and a process which can also undergo an infinite sequence of a's. This leads to problems when using this semantics for Object-Z classes where unbounded nondeterminism arises naturally. For example, consider the following Object-Z class.



An object of this class can perform the operation Op v times, where v is the actual value of x. Therefore, the corresponding process can perform any finite sequence of $Op.\varnothing$ events but cannot perform an infinite sequence of $Op.\varnothing$ events. This fact is not captured in the semantics which assumes that the infinite sequences of events can be extrapolated from the finite sequences. Hence, in this case, the semantics assumes that the process can in fact undergo an infinite sequence of $Op.\varnothing$ events. Hence, if $Op.\varnothing$ is subsequently hidden, the process will diverge. Such a semantics is counter-intuitive and either the use of hiding in the CSP part of a specification needs to be restricted or an alternative semantic model needs to be adopted.

The use of hiding can be restricted by placing a well-definedness condition on the hiding operator as is done in [24]. That is, given a process P with failures F, $P \backslash C$ is well-defined only if

$$\forall s \in \mathrm{dom}\ F \bullet \neg\ (\forall n \in \mathbb{N} \bullet \exists t \in C^* \bullet \#t > n \wedge s \frown t \in \mathrm{dom}\ F)$$

This prevents unbounded sequences of events being hidden. Although this seems to reduce the expressibility of the notation, it should be noted that unbounded sequences of events are not likely to occur in real systems. Usually there is a finite bound on the number of consecutive occurrences of any particular event and this bound can be specified fairly

abstractly by a constant whose type is, for example, the set of all natural numbers. This approach was taken to model the bounded queue in Section 2: the constant *max* modelled the bound on the number of consecutive *Join* or *Leave* events without placing an exact value on this bound.

However, although this restriction prevents unbounded sequences of events being hidden it does not tackle all the counter-intuitive behavioural problems of unbounded nondeterminism. For example, if we define[10]

$$P = a?n : \mathbb{N} \rightarrow Q(n)$$
$$Q(n) = \text{if } n > 0 \text{ then } b \rightarrow Q(n-1) \text{ else } \mathit{stop}$$

then with the above semantic model $P\backslash b\backslash a$ is *stop* and $P\backslash a\backslash b$ diverges. Although the well-definedness condition on the hiding operator prevents the formation of $P\backslash a\backslash b$ (and therefore solves the problem with divergence) the solution is perhaps less than satisfactory since $P\backslash b\backslash a$ is still different from $P\backslash a\backslash b$.

An alternative approach is via the subject of data independent (c.f. the definition and discussion of data independence in Section 15 of [30]). If the program P is data independent in a type T, and the set C can be infinite only by virtue of being itself data independent in T (when T is infinite, for example by being the set of events associated with a channel of type T), then $P\backslash C$ is well behaved even when T and C are infinite.

Another possibility would be to extend the failures-divergences semantics with a component corresponding to the infinite traces of a process as is done in [32]. This approach is adopted for combining CSP and action systems in [8]. In addition to failures and divergences, the semantics of a process includes a component I corresponding to the infinite traces of a class. I is defined in terms of the alphabet of events A as follows.

$I \subseteq A^{\omega}\backslash A^{*}$ such that the following properties hold.[11]

$$t^{\frown}u \in I \Rightarrow (t, \varnothing) \in F \tag{I1}$$
$$t \in D \Rightarrow t^{\frown}u \in I \tag{I2}$$
$$(t_1, \varnothing) \in F \Rightarrow (\exists T \bullet (\forall t_2 \in T \bullet (t_1{}^{\frown}t_2, \{a \mid t_2{}^{\frown}\langle a\rangle \notin T\}) \in F) \wedge \{t_1{}^{\frown}u \mid u \in \bar{T}\} \subseteq I) \tag{I3}$$

where T is a non-empty, prefix-closed set of finite traces and $\bar{T} = \{u \in A^{\omega}\backslash A^{*} \mid \forall t < u \bullet t \in T\}$.

Axioms I1 and I2 are straightforward extensions of axioms F2 and D2 respectively. The purpose of axiom I3 is to ensure there are enough infinite traces analogously to the way that axiom F4 ensures there are enough failures. It states that there exists at least one deterministic refinement of a process after it has undergone a trace t (the finite traces of this refinement are given by the set T), and that any infinite trace that this refinement can undergo is in I. The explanation and derivation for this axiom are quite subtle and the interested reader is referred to [32] for details.

The set of infinite traces corresponding to an Object-Z class C with operations O can be derived from the histories of C as follows.

$$\mathcal{I}(C) = \{t \mid \exists (s, o) \in \mathcal{H}(C) \bullet o \in O^{\omega} \backslash O^{*} \wedge t = events(o)\}$$

The proofs of the axioms I1 to I3 for this definition are given in [18]. This paper also includes a case study illustrating the differences between using the standard failures-divergences semantics and that with infinite traces for combining Object-Z and CSP. It shows that the infinite trace semantics is preferable from a theoretical point of view: it can handle all forms of unbounded nondeterminism, the rules for refinement in the presence of hidden events have fewer side conditions and are hence more applicable, and it is even possible to handle fairness constraints under certain restrictions.

An alternative model of infinite traces is discussed in Chapter 10 of [30], and this could also serve as a semantic model for a notation including unbounded nondeterminism.

However, there is still a strong case for using the standard CSP semantics. This semantics is more widespread in the CSP community and is the basis of tools like the CSP model checker FDR [25] and an encoding of CSP in Isabelle/HOL [43]. Furthermore, although the refinement rule in the presence of hidden events is more complicated, it can be proven to be complete whereas that for the infinite trace semantics cannot [7]. For the purposes of this paper, we adopt the standard CSP semantics and therefore restrict the use of hiding in the CSP part of the specification as previously discussed.

5. Specifying concurrent systems

In this section, we describe the approach to specifying concurrent systems using the integrated notations. The approach comprises three phases.

- The first phase involves specifying the component processes using Object-Z. Since all interaction of system components is specified in the CSP part of the specification, a restricted subset of Object-Z is used which does not include instantiation of objects of a class. It also, therefore, does not include polymorphism, class union or object containment which are only used in the context of object instantiation, and the parallel $\|$ and enrichment $\bullet$ operators which were introduced in Object-Z to model object interaction (see [14, 39] for details). These restrictions greatly simplify reasoning about the Object-Z part of the specification.

  To maintain a separation of concerns and allow maximum flexibility in describing the component processes, each is described independently of the others and of the environment in which they are to be placed. This also allows classes to be more easily shared between specifications.
- The components specified in the first phase will generally not be in a form that allows them to be composed using CSP operators. The second phase involves modifying the class interfaces so that they will synchronise and communicate as desired. This may be achieved using Object-Z inheritance to rename operations and operation

parameters and to add state variables and operation parameters where required. For example, 'dummy' operation parameters may be introduced to allow operations with different parameters to synchronise. Such 'dummy' parameters should not be restricted in the operation's predicate. CSP renaming may also be used on appropriate processes in this phase.

- The final phase involves the specification of the system using CSP operators. Only a subset of the operators defined in [21] is, in fact, required. For example, since we are not specifying processes using the notations for input and output channels, the piping operator $\gg$ intended for use with these notations is not required. Also, since Object-Z classes have no notion of termination, the sequential composition operator; and its associated notations (see [21] for details) are not required.

  As noted above, a well-definedness condition is placed on the hiding operator restricting its use.

To illustrate the approach we present a case study of a cinema booking system. This case study is based on the specification of the Apollo box office in [47] but extended to support multiple customers.

5.1. Specifying the components of a system

The Marlowe box office allows customers to book tickets in advance by telephone. When a customer calls, if there is an available ticket then one is allocated and put to one side for the caller. When the customer arrives, they are presented with this ticket.

The components of the booking system are the customers and the Marlowe box office. In our approach, these will be specified by Object-Z classes. Consider the specification of a customer of the booking system. Let *Name* denote the set of all customer names and *Ticket* the set of all tickets.

Customer

$my\_name : Name$

Book

$name! : Name$

$name! = my\_name$

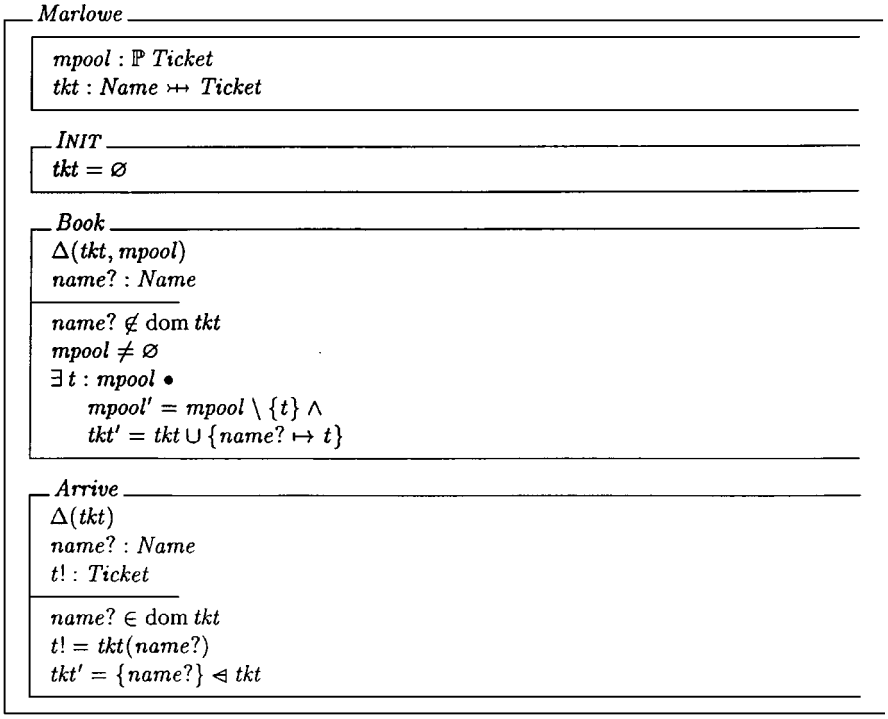
Arrive

$name! : Name$
$t? : Ticket$

$name! = my\_name$

This class has a single constant *my_name* denoting the name of the customer and two operations: *Book* and *Arrive*. The operations *Book* and *Arrive* correspond to the customer booking a ticket and arriving to collect a ticket respectively.

The other component of the system is the Marlowe box office which is also specified as an Object-Z class.



This class has a state schema with two state variables: *mpool*, denoting the pool of tickets, and *tkt*, a partial injective function from *Name* to *Ticket* recording which tickets have been allocated to which customers. Initially, no tickets have been allocated.

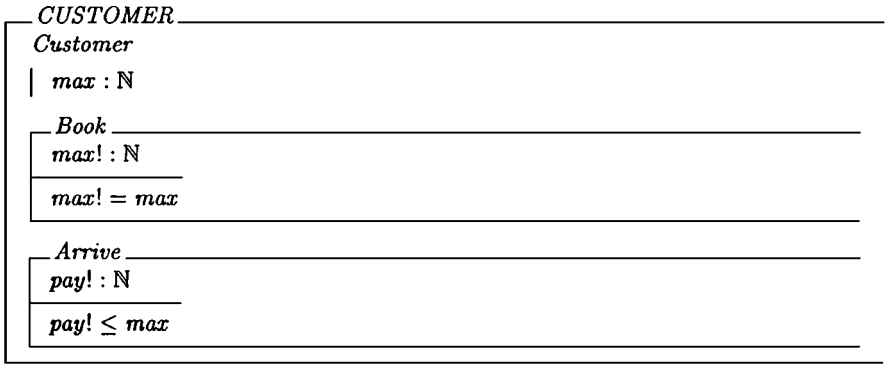
The operation *Book* is feasible whenever there are still tickets available ($mpool \neq \varnothing$) and allocates a ticket to a customer who has not already made a booking ($name? \notin \text{dom } tkt$). The operation *Arrive* issues the ticket but does not change the pool of tickets ($mpool = mpool'$ is a consequence of *mpool* not appearing in the Δ-list of the operation *Arrive*).

5.2. Specifying the component interfaces

Although we do not need to modify the component interfaces in our running example, we will illustrate how it can be achieved if needed by use of Object-Z inheritance and CSP renaming.

Suppose, for example, we wished to use the *Customer* component in another system where the customer specified a maximum price (*max*) they were willing to pay when purchasing a ticket. Furthermore, when collecting the ticket, the customer pays for it at the box office. We

reuse the *Customer* class to build a new class *CUSTOMER* where we add this information to the operations *Book* and *Arrive* as follows.



We have inherited the *Customer* class and captured the new requirements by including additional declarations and predicates in the two operations. If we wished to rename operations in this interface, for example renaming *Arrive* to *Collect*, we can do so in the CSP part of the specification by applying a substitution to the appropriate components. For example, we could define

$$NCUSTOMER = CUSTOMER[\![Collect/Arrive]\!]$$

to achieve the renaming we required.

5.3. Specifying the system

To specify the booking system we use CSP operators to capture the interaction between the customers and box office. To do so we define a process $Customer_n$ corresponding to the customer with name n as follows.

$$Customer_n = Customer_{\{(my\_name,n)\}}$$

We also write *Marlowe* for the process $Marlowe_{\varnothing}$ corresponding to the class *Marlowe* without any restriction on the initial state. The complete booking system specification is then given by the composition of the processes $Customer_n$ and *Marlowe* as follows:

$$BookingSystem = (|||_{n:Name}\ Customer_n) \parallel Marlowe$$

That is, the booking system consists of the box office running concurrently with a collection of customers—one for each name in *Name*.

Since this part of the specification is a CSP specification, we can state properties we wish to prove about it in the same way as they are stated in CSP (see [21]). That is, in the form P **sat** S where P is a process and S is a predicate in terms of *tr*, the traces, and *ref*, the refusal sets, of the failures of process P. For example, the property that the number of bookings

made is greater than or equal to the number of tickets allocated to arriving customers can be stated as follows.[12]

$$BookingSystem \textbf{ sat } \#tr \downarrow Book \geq \#tr \downarrow Arrive$$

An approach to proving such properties in terms of the component Object-Z classes is presented in Section 7.

6. Refining concurrent systems

This section presents a method of refinement for systems specified using the integrated Object-Z/CSP notation. The use of CSP semantics for Object-Z classes enables us to use CSP refinement as the refinement relation for the integrated notation. To verify such a refinement there are two different approaches that can be employed:

- The first is based on the approach used in CSP. The refinement is verified directly by calculating and comparing the failures of the specifications or, in the case where the specifications have identical structure, the failures of the components of the specifications.
- The second involves using state-based methods to verify the refinement of the component Object-Z classes of a specification. This is achieved by adapting the work of Josephs [24], which provides refinement relations for state-based systems that are sound and complete with respect to CSP refinement.

In this section we illustrate both approaches by refining the cinema booking system of Section 5.

6.1. Failures approach

Refinement in CSP is defined in terms of failures and divergences [6]. A process Q is a refinement of a process P if

$$failures\ Q \subseteq failures\ P \text{ and } divergences\ Q \subseteq divergences\ P$$

Since divergences are empty for all Object-Z classes, we only need to show the subset relation between the failures sets, that is

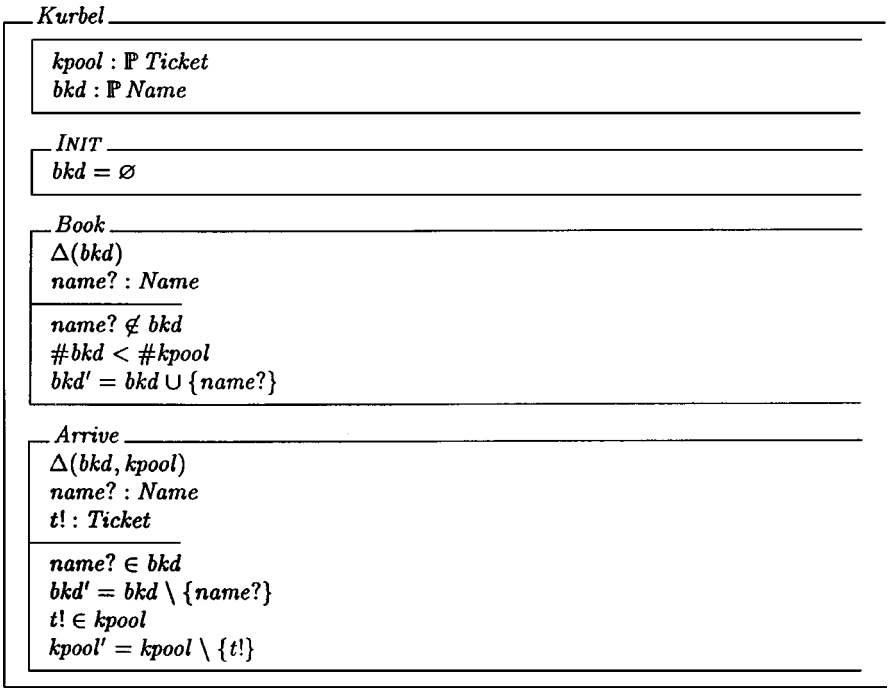
$$failures\ Q \subseteq failures\ P.$$

We write $P \sqsubseteq Q$ to denote the latter. As an example, consider an alternative booking system to the *BookingSystem* specification given in Section 5.

Like the Marlowe box office, the Kurbel box office allows customers to book tickets in advance by telephone. However, the procedure is different from that used at the Marlowe. When a customer calls, if there is an available ticket then the customer's name is simply recorded. When a customer whose name has been recorded arrives at the box office, a ticket is allocated.

The contrast between the Marlowe and the Kurbel box offices is the point of allocation of tickets (at booking time *vs* at collection time). However, at this level of abstraction the customer cannot tell that the Kurbel is behaving differently to the Marlowe. We will prove this property by showing that the Kurbel booking system is a CSP refinement of the Marlowe booking system.

The components of the Kurbel booking system are the customers and the Kurbel box office. The specification of a customer is identical to that given in the Marlowe booking system. The Kurbel box office is represented by the following Object-Z class.



The state variable *kpool* denotes the pool of tickets and *bkd* denotes the set of names of customers who have booked a ticket. Initially, *bkd* is empty. The operation *Book* records a booking provided that there are currently less bookings than tickets and, hence, still tickets available. The operation *Arrive* allocates a ticket to a customer who has a booking.

The complete system again consists of the box office running concurrently with a collection of customers.

$$BookingSystem_K = (|||_{n:Name}\ Customer_n) \parallel Kurbel$$

To show that $BookingSystem_K$ is a refinement of *BookingSystem*, we will compare their failures. Since the structure of the booking system specifications are identical and the components $Customer_n$ are identical, we need only show that $failures(Kurbel) \subseteq failures(Marlowe)$.

Consider first the class *Kurbel*. The failures of *Kurbel* can be given in terms of the failures of the processes $Kurbel_{\{(kpool,p)\}}$ for each possible set of tickets p.

$$failures(Kurbel) = \bigcup_{p \in \mathbb{P}\ Ticket} failures\big(Kurbel_{\{(kpool,p)\}}\big)$$

The traces of $Kurbel_{\{(kpool,p)\}}$ comprise the empty trace and any trace formed by extending a trace of $Kurbel_{\{(kpool,p)\}}$ by

- a *Book* event whenever the customer doing the booking has arrived and collected any tickets he or she has previously booked and whenever there are tickets left, and
- an *Arrive* event whenever
  - the ticket being collected was initially in *kpool*,
  - the ticket being collected has not been previously collected by any customer and
  - the customer arriving has booked once more than he or she has arrived to collect a ticket.

$$\begin{aligned}
traces(Kurbel) = {} & \{\langle\,\rangle\} \\
& \cup \\
& \{s \frown \langle\, Book.\{(name, n)\}\,\rangle \mid s \in traces(Kurbel) \wedge n \in Name \wedge \\
& \quad \#(s \upharpoonright \{Book.\{(name, n)\}\}) = \#(s \upharpoonright \{Arrive.\{(name, n), (t, x)\} \mid x \in Ticket\}) \wedge \\
& \quad \#(s \upharpoonright \{Book.\{(name, m)\} \mid m \in Name\}) < \#p\} \\
& \cup \\
& \{s \frown \langle\, Arrive.\{(name, n), (t, x)\}\,\rangle \mid s \in traces(Kurbel) \wedge n \in Name \wedge \\
& \quad x \in p \wedge \#(s \upharpoonright \{Arrive.\{(name, m), (t, x)\} \mid m \in Name\}) = 0 \wedge \\
& \quad \#(s \upharpoonright \{Book.\{(name, n)\}\}) = \#\,(s \upharpoonright \{Arrive.\{(name, n), (t, y)\} \\
& \qquad\qquad\qquad\qquad \mid y \in Ticket\}) + 1\ \}
\end{aligned}$$

$Kurbel_{\{(kpool,\,p)\}}$ can refuse a *Book* event whenever the customer making the booking has booked more times than he or she has arrived, or there are no tickets remaining in *kpool*. It can refuse an *Arrive* event whenever the customer arriving has already arrived as many times as he or she has booked, the ticket of the *Arrive* event has already been allocated to a customer or the ticket of the *Arrive* event was not in *kpool* initially. It can also refuse all but one of the possible assignment of values to the output parameter $t!$ for the *Arrive* event reflecting the constraint that the outputs of *Arrive* cannot be constrained by the environment.

Hence, the failures of $Kurbel_{\{(kpool,\,p)\}}$ are

$$failures\big(Kurbel_{\{(kpool,\,p)\}}\big) = \{(tr, X) \mid tr \in traces\big(Kurbel_{\{(kpool,\,p)\}}\big) \wedge X \subseteq S\}$$

where

$$\begin{aligned}
S = {} & \{Book.\{(name, n)\} \mid \mathrm{n} \in Name \wedge \\
& \quad (\#(tr \upharpoonright \{Book.\{(name, n)\}\} \geq \#(tr \upharpoonright \{Arrive.\{(name, n), (t, y) \mid y \in Ticket\}) \\
& \qquad \vee \#(tr\ \{Arrive.\{(name, l), (t, y) \mid l \in Name \wedge y \in Ticket\} = \#\,p)\} \\
& \cup \\
& \{Arrive.\{(name, m), (t, x)\} \mid x \in Ticket \wedge m \in Name \wedge
\end{aligned}$$

$$\begin{aligned}(\#(tr \upharpoonright \{Book.\{(name, m)\}\}) = \#(tr\ \{Arrive.\{(name, m), (t, x)\}\})\\ \vee \#(tr \upharpoonright \{Arrive.\{(name, l), (t, x)\} \mid l \in Name\}) \neq 0\\ \vee x \notin p\\ \vee (\exists y \in Ticket \bullet Arrive.\{(name, m), (t, y)\} \notin S)\}\end{aligned}$$

The failures of *Marlowe* can similarly be given in terms of the failures of the processes $Marlowe_{\{(mpool, p)\}}$ for each possible set of tickets p.

$$failures(Marlowe) = \bigcup_{p \in \mathbb{P}\ Ticket} failures\big(Marlowe_{\{(mpool, p)\}}\big)$$

It easy to see that the traces of $Marlowe_{\{(mpool, p)\}}$ are identical to those of $Kurbel_{\{(kpool, p)\}}$. In addition, $Marlowe_{\{(mpool, p)\}}$ can refuse exactly the same events as $Kurbel_{\{(kpool, p)\}}$ can after every trace. Hence, $failures(Kurbel_{\{(mpool,k)\}}) = failures(Marlowe_{\{(mpool,k)\}})$ for all $k \in \mathbb{P}\ Ticket$ and, therefore, $failures(Kurbel) \subseteq failures(Marlowe)$ as desired. This proves that *Kurbel* is a refinement of *Marlowe*, and in fact we can also see that *Marlowe* is a refinement of *Kurbel*.

6.2. State-based approach

Calculating and comparing the failures of classes as illustrated above is feasible, but can be complex for non-trivial specifications. The purpose of this section is to show how we can use state-based refinement techniques for the Object-Z component of a specification. This will enable refinements to be verified at the specification level, rather than working explicitly in terms of failures, traces and refusals at the semantic level.

Work on state-based refinement for concurrent systems goes back to He [19] and Josephs [24], who have developed refinement relations for state-based transition systems which are complete and sound with respect to CSP refinement. Woodcock and Morgan [48] have produced similar results in the context of action systems and weakest precondition formulae. In this section we adapt the work of Josephs to the Object-Z setting. This work is directly applicable to this context because it uses processes which do not diverge and places the same restrictions on hiding that we have adopted. We produce two refinement relations, called upward and downward simulation, which together are sound and complete with respect to CSP refinement. Using these rules we can refine the Object-Z components of an integrated Object-Z/CSP specification such that the entire specification is also refined, i.e. in a compositional manner.

Josephs considers a state-based system P to be defined by a tuple $(A, S, \rightarrow, R)$ where A is its alphabet, S its states, $\rightarrow$ its transition relation and R its initial states ($R \subseteq S$, $R \neq \varnothing$). As usual we will denote a transition under event e from state σ_1 to σ_2 by $\sigma_1 \xrightarrow{e} \sigma_2$. In addition, the set of next possible events that a system P can undergo when in state σ is denoted $next_P(\sigma)$, i.e.

$$next_P(\sigma) = \{e \in A \mid \exists \sigma' \in S \bullet \sigma \xrightarrow{e} \sigma'\}$$

Refinement in state-based systems is based on the concept of simulations. For example, simulation forms the basis of the refinement rules in Z as they are usually presented [47].

Josephs uses two versions called downward and upward simulation (sometimes called forward and backward simulations respectively) defined as follows.

Definition 1 (Downward simulation). P_2 is a downward simulation of P_1 if there is a relation $D \subseteq S_1 \times S_2$ such that

1. $\forall \sigma_1 \in S_1, \sigma_2 \in S_2 \bullet \sigma_1 D \sigma_2 \Rightarrow next_{P_1}(\sigma_1) = next_{P_2}(\sigma_2)$
2. $\forall \sigma_1 \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_1 D \sigma_2 \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2' \Rightarrow (\exists \sigma_1' \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1' D \sigma_2')$
3. $\forall \sigma_2 \in R_2 \bullet \exists \sigma_1 \in R_1 \bullet \sigma_1 D \sigma_2$

Definition 2 (Upward simulation). P_2 is an upward simulation of P_1 if there is a relation $U \subseteq S_1 \times S_2$ such that

1. $\forall \sigma_2 \in S_2 \bullet \exists \sigma_1 \in S_1 \bullet \sigma_1 U \sigma_2 \wedge next_{P_1}(\sigma_1) \subseteq next_{P_2}(\sigma_2)$
2. $\forall \sigma_1' \in S_1, \sigma_2, \sigma_2' \in S_2, e \in A \bullet \sigma_1' U \sigma_2' \wedge \sigma_2 \xrightarrow{e}_2 \sigma_2' \Rightarrow (\exists \sigma_1 \in S_1 \bullet \sigma_1 \xrightarrow{e}_1 \sigma_1' \wedge \sigma_1 U \sigma_2)$
3. $\forall \sigma_1 \in S_1, \sigma_2 \in R_2 \bullet \sigma_1 U \sigma_2 \Rightarrow \sigma_1 \in R_1$.

Josephs then proves that these two relations are sound and complete with respect to CSP refinement.

To use these results, we first adapt the definitions to the Object-Z setting. The translation is straightforward, and the relations D and U between the state spaces are re-cast as retrieve relations (denoted *Abs*) between the abstract state (*Astate*) and the concrete state (*Cstate*).

To translate the rules involving $next_P(\sigma)$ we adopt the precondition operator *pre* from Z to return the precondition of Object-Z operations. The event corresponding to an Object-Z operation *Op* is in $next_P(\sigma)$ iff pre *Op* is true in the state representing σ. This is because the interpretation of operations in Object-Z differs from that in Z in that an operation cannot occur when its precondition is not enabled.[13] We can now give the definition of downward and upward simulation in Object-Z.

Definition 3 (Downward simulation). An Object-Z class C is a downward simulation of the class A if there is a retrieve relation *Abs* such that every abstract operation *AOp* is recast into a concrete operation *COp* and the following hold.

DS.1 $\forall$ *Astate*; *Cstate* $\bullet$ *Abs* $\Rightarrow$ (*pre AOp* $\Leftrightarrow$ *pre COp*)
DS.2 $\forall$ *Astate*; *Cstate*; *Cstate*$'$ $\bullet$ *Abs* $\wedge$ *COp* $\Rightarrow$ ($\exists$ *Astate*$'$ $\bullet$ *Abs*$'$ $\wedge$ *AOp*)
DS.3 $\forall$ *Cinit* $\bullet$ $\exists$ *Ainit* $\bullet$ *Abs*

Definition 4 (Upward simulation). An Object-Z class C is an upward simulation of the class A if there is a retrieve relation *Abs* such that every abstract operation *AOp* is recast into a concrete operation *COp* and the following hold.

US.1 $\forall$ *Cstate* $\bullet$ $\exists$ *Astate* $\bullet$ *Abs* $\wedge$ *pre AOp* $\Rightarrow$ *pre COp*
US.2 $\forall$ *Astate*$'$; *Cstate*; *Cstate*$'$ $\bullet$ *COp* $\wedge$ *Abs*$'$ $\Rightarrow$ ($\exists$ *Astate* $\bullet$ *Abs* $\wedge$ *AOp*)
US.3 $\forall$ *Astate*; *Cinit* $\bullet$ *Abs* $\Rightarrow$ *Ainit*

Using these rules we can show that the Kurbel class is an upward simulation, and hence a refinement, of the Marlowe class without having to calculate the failures. To do so we first

record the relationship between the two classes as a retrieve relation given by

Ret

$$Kurbel.\textsc{State}$$
$$Marlowe.\textsc{State}$$

$$bkd = \text{dom } tkt$$
$$kpool = mpool \cup \text{ran } tkt$$
$$mpool \cap \text{ran } tkt = \varnothing$$

Kurbel.*STATE* denotes the state schema in the class *Kurbel*, etc.

Firstly, to prove the initialisation correct (US.3) we must prove the following:

$$\forall\ Marlowe.State; Kurbel.\textsc{Init} \bullet Ret \Rightarrow Marlowe.\textsc{Init}$$

To do so we must show the following holds (which it clearly does).

$$\forall\ mpool : \mathbb{P}\ Ticket; tkt : Name \rightarrowtail Ticket; kpool : \mathbb{P}\ Ticket; bkd : \mathbb{P}\ Name \mid bkd = \varnothing \bullet$$
$$bkd = \text{dom } tkt \wedge kpool = mpool \cup \text{ran } tkt \wedge mpool \cap \text{ran } tkt = \varnothing \Rightarrow tkt = \varnothing$$

Next, we must show that US.1 holds for the operations *Book* and *Arrive*. For the *Book* operation, this requires us to show that

$$\forall\ Kurbel.\textsc{State} \bullet \exists\ Marlowe.\textsc{State} \bullet Ret \wedge pre\ Marlowe.Book \Rightarrow \text{pre } Kurbel.Book$$

This amounts to showing that

$$\forall\ kpool : \mathbb{P}\ Ticket; bkd : \mathbb{P}\ Name \bullet \exists\ mpool : \mathbb{P}\ Ticket; tkt : Name \rightarrowtail Ticket \bullet$$
$$(bkd = \text{dom } tkt \wedge kpool = mpool \cup \text{ran } tkt \wedge mpool \cap \text{ran } tkt = \varnothing) \wedge$$
$$(name? \notin \text{dom } tkt \wedge mpool \neq \varnothing) \Rightarrow$$
$$(name? \notin bkd \wedge \#\, bkd < \#\, kpool).$$

Given the declarations and the constraints in *Ret*, we proceed as follows.

$$name? \notin \text{dom } tkt \wedge mpool \neq \varnothing$$
$$\Rightarrow name? \notin \text{dom } tkt \wedge \#\, mpool > 0$$
$$\Rightarrow name? \notin \text{dom } tkt \wedge \#\, \text{ran } tkt < \#(mpool \cup \text{ran } tkt)$$
$$\Rightarrow name? \notin \text{dom } tkt \wedge \#\, \text{dom } tkt < \#(mpool \cup \text{ran } tkt) \qquad [\text{since } \#\, \text{dom } tkt = \#\, \text{ran } tkt]$$
$$\Rightarrow name? \notin bkd \wedge \#\, bkd < \#\, kpool \qquad [\text{By } Ret]$$

A similar proof can be given for the operation *Arrive*.

Finally, we must show that US.2 holds for the operations *Book* and *Arrive*. For the *Arrive* operation, this requires us to show that

$$\forall\ Marlowe.\textsc{State}', Kurbel.\textsc{State}, Kurbel.\textsc{State}' \bullet$$
$$Kurbel.Arrive \wedge Ret' \Rightarrow (\exists\ Marlowe.\textsc{State} \bullet Ret \wedge Marlowe.Arrive).$$

That is, given the declarations we need to show that

$$
\begin{array}{l}
(name? \in bkd \wedge bkd' = bkd \setminus \{name?\} \wedge t! \in kpool \wedge kpool' = kpool \setminus \{t!\} \wedge \\
bkd' = \mathrm{dom}\ tkt' \wedge kpool' = mpool' \cup \mathrm{ran}\ tkt' \wedge \varnothing = mpool' \cap \mathrm{ran}\ tkt') \Rightarrow \\
\quad (\exists\, mpool : \mathbb{P}\ Ticket;\ tkt : Name \rightarrowtail\!\!\!\!\!\!\rightarrow Ticket \bullet \\
\qquad bkd = \mathrm{dom}\ tkt \wedge kpool = mpool \cup \mathrm{ran}\ tkt \wedge mpool \cap \mathrm{ran}\ tkt = \varnothing \wedge \\
\qquad name? \in \mathrm{dom}\ tkt \wedge mpool = mpool' \wedge tkt' = \{name?\} \lhd tkt \wedge \\
\qquad\quad t! = tkt(name?)).
\end{array}
$$

This can be seen to be true if we take $mpool = mpool'$ and $tkt = tkt' \cup \{name? \mapsto t!\}$. We only need to prove the first three conjuncts of the consequent, the rest follow trivially from our choice of *mpool*, etc. For example, with these choices we can then make the following deductions.

$$
\begin{aligned}
\mathrm{dom}\ tkt &= \mathrm{dom}(tkt' \cup \{name? \mapsto t!\}) = \mathrm{dom}\ tkt' \cup \{name?\} \\
&= bkd' \cup \{name?\} = (bkd \setminus \{name?\}) \cup \{name?\} \\
&= bkd
\end{aligned}
$$

$$mpool \cup \mathrm{ran}\ tkt = mpool' \cup \mathrm{ran}\ tkt' \cup \{t!\} = kpool' \cup \{t!\} = kpool$$

Finally, to show that $mpool \cap \mathrm{ran}\ tkt = \varnothing$ we note that (since $\mathrm{ran}\ tkt = \mathrm{ran}\ tkt' \cup \{t!\}$)

$$mpool \cap \mathrm{ran}\ tkt = (mpool \cap \mathrm{ran}\ tkt') \cup mpool \cap \{t!\} = \varnothing \cup (mpool \cap \{t!\})$$

Now from $t! \in kpool \wedge t! \notin kpool'$ we deduce that $t! \notin mpool' = mpool$. Therefore $mpool \cap \mathrm{ran}\ tkt = \varnothing$.

A similar proof can be given for the operation *Book*.

This concludes the proof that *Kurbel* is an upward simulation of *Marlowe*, and therefore a CSP refinement. As with the failures approach, from this we can conclude that $BookingSystem_K$ is indeed a refinement of *BookingSystem*.

6.3. Weak simulations

The upward and downward simulation techniques discussed in the previous subsection can be extended to classes which contain operations which are subsequently *hidden* (by the use of the CSP hiding operator), and are thus not visible to the environment. Such operations are called *local* operations. The extensions of the simulation rules enable one to prove refinements of the form $A \sqsubseteq C \backslash X$ where X is the set of events of the concrete class C being hidden. Simulation rules for this situation are called *weak simulations* [10, 11]. [18] discusses the use of these rules for the combined Object-Z/CSP notation with respect to both the finite and infinite trace model. Here we present the results for the standard semantics (i.e. the finite trace model).

In a weak simulation, the abstract state is not changed by a local operation in the concrete class. For C to be a weak downwards simulation of a class A we require that there is a retrieve relation *Abs* such that every abstract operation *AOp* is recast into a concrete operation *COp* and the following hold (where L is the set of local operations in C).

WDS.1 $\forall$ *Astate*; *Cstate* $\bullet$ (*Abs* $\wedge$ $\forall l : L \bullet \neg$pre l) $\Rightarrow$ (pre *AOp* $\Leftrightarrow$ pre *COp*)
WDS.2 $\forall$ *Astate*; *Cstate*; *Cstate*$'$ $\bullet$ *Abs* $\wedge$ *COp* $\Rightarrow$ ($\exists$ *Astate*$'$ $\bullet$ *Abs*$'$ $\wedge$ *AOp*)
WDS.3 $\forall$ *Cinit* $\bullet$ $\exists$ *Ainit* $\bullet$ *Abs*

Furthermore there exists a termination schema *term* $\widehat{=}$ [*Cstate*; $t : \mathbb{N} \mid P$] such that *term* is defined for all states of C and for all local operations $l \in L$ the following hold.

WDS.4 l does not change the abstract state: $l \wedge Abs \Rightarrow (\exists\, Astate' \bullet \Xi\, Astate \wedge Abs')$
WDS.5 l does not diverge: $l \wedge term \Rightarrow t' < t \wedge term'$
WDS.6 There exists a constant (b) that bounds the increase of t for all concrete operations: $COp \wedge \Delta\, term \Rightarrow t' < t + b$.
WDS.7 In addition, the initial value of t must be bound, i.e., $Cinit \wedge term \Rightarrow t < b$

The definition of weak upwards simulation follows similar lines.

As an example, consider the following specification of the Schonell box office.

Schonell

$spool : \mathbb{P}\, Ticket$
$bd, ar : \text{seq}\, Name$

INIT

$bd = ar = \langle\rangle$

Book

$\Delta(bd)$
$name? : Name$

$name? \notin \text{ran}\, bd$
$\#bd < \#spool$
$bd' = \langle name? \rangle \frown bd$

Arrive

$\Delta(bd, ar, spool)$
$name? : Name$
$t! : Ticket$

$name? \in \text{ran}\, ar$
$t! \in spool$
$spool' = spool \setminus \{t!\}$
$bd' = bd \upharpoonright (Name \setminus \{name?\})$
$ar' = ar \upharpoonright (Name \setminus \{name?\})$

transfer

$\Delta(ar)$

$\#bd \neq \#ar$
$ar' = ar \frown \langle head\, bd \rangle$

The Schonell box office implements the same booking strategy as the Kurbel box office, but is described in terms of two sequences of names, one for the booking process and one for the ticket collection. A local operation *transfers* names into the sequence used for collection. As it is internal it is hidden from the environment, and thus we wish to prove that $Kurbel \sqsubseteq Schonell \backslash \{transfer\}$. To do so we describe the retrieve relation as

Ret

$Kurbel.\textsc{State}$
$Schonell.\textsc{State}$

$spool = kpool$
$bkd = \operatorname{ran} bd$

and verify conditions **WDS.1–7** above. For example, since $\neg \text{pre } transfer = (\#bd = \#ar)$ to show **WDS.1** we need to prove that

$$Ret \wedge \#bd = \#ar \Rightarrow (\text{pre } AOp \Leftrightarrow \text{pre } COp)$$

for operations *Book* and *Arrive*, which is easily done. To prove the last three conditions, we take as termination schema the following

term

$Schonell.\textsc{State}$
$t : \mathbb{N}$

$t = \#bd - \#ar$

from which the conditions easily follow.

7. Verifying concurrent systems

This section presents a method of verification for the integrated notation. The method allows us to verify properties of the CSP system specification in terms of its component Object-Z classes. It comprises three phases.

- The first phase involves reasoning about the CSP part of the specification. System properties are stated and transformed to properties of the component Object-Z classes using the notation and laws for CSP operators of [21].
- The properties of the Object-Z classes derived in the first phase will often include terms not readily reasoned about in Object-Z. The second phase involves extending the Object-Z classes with auxiliary variables to model these terms. This is achieved using Object-Z inheritance which allows the addition of variables and predicates to the state schema, initial state schema and operations of a class. Reasoning can then be carried out using the logic for Object-Z presented in [35].
- The final phase involves showing that the classes extended with the auxiliary variables are refined by the original Object-Z classes and hence the original classes also satisfy the desired properties.

To illustrate the approach, we will verify the property of *BookingSystem* stated at the end of Section 5.

7.1. Reasoning about the CSP processes

Properties about CSP processes can be stated in term of their failures. Given a process P with failures F, the property $\forall(tr, ref) \in F \bullet S(tr, ref)$ can be expressed using the notation of [21] as P **sat** $S(tr, ref)$. For example, the following property of the process *BookingSystem* states that the number of bookings made is greater than or equal to the number of tickets allocated to arriving customers.

$$BookingSystem \textbf{ sat } \#tr \downarrow Book \geq \#tr \downarrow Arrive$$

To prove such a property in CSP, we would use the laws for the various CSP operators given in [21]. Therefore, we re-express the property in terms of CSP operators by replacing *BookingSystem* with its definition in terms of component processes.

$$(|||_{n:Name}\, Customer_n) \parallel Marlowe \textbf{ sat } \#tr \downarrow Book \geq \#tr \downarrow Arrive$$

In this form, we can apply the following law for the parallel composition operator.[14]

If P **sat** $S(tr)$
and Q **sat** $T(tr)$
then $(P \parallel Q)$ **sat** $(S(tr \restriction \alpha P) \wedge T(tr \restriction \alpha Q))$.

Let $S(tr \restriction \alpha(|||_{n:Name}\, Customer_n)) = true$ and, since the alphabet of *Marlowe* is identical to that of *BookingSystem*, let $T(tr \restriction \alpha Marlowe) = \#tr \downarrow Book \geq \#tr \downarrow Arrive$. Using the law for the parallel composition operator, the above property is true whenever the following is true.

$$Marlowe \textbf{ sat } \#tr \downarrow Book \geq \#tr \downarrow Arrive$$

This property is now in terms of a process corresponding to an Object-Z class and we can no longer use the laws for CSP operators. To complete the proof, we require a method for showing the above property is true for the Object-Z class *Marlowe*.

7.2. Reasoning about the Object-Z classes

Building on the work in [46], a logic for reasoning about Object-Z classes is presented in [35]. Properties of classes are expressed as sequents of the form

$$A :: d \mid \Psi \vdash \Phi$$

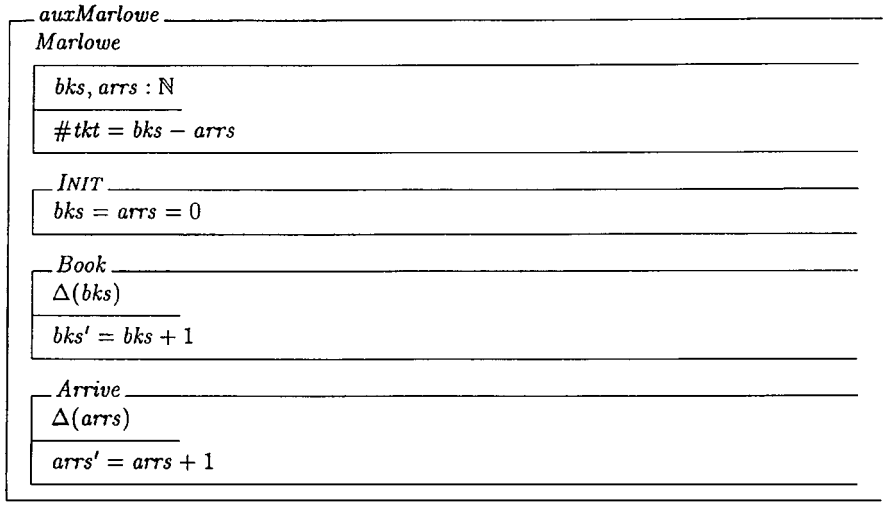
where A is a class name, d is a list of declarations and Ψ and Φ are lists of predicates. The sequent is valid, i.e. the stated property is true, whenever given the declarations d

and predicates Ψ at least one of the predicates in Φ is true in class A. For example, the following is a valid sequent (*INIT* denotes the declarations and predicates of the *INIT* schema of *Marlowe*).

$$Marlowe :: INIT \vdash tkt = \varnothing$$

The predicates in Ψ and Φ are only defined in terms of variables and constants which are accessible in the class or declared in d. Hence, it is not possible to state properties about sequences of events such as those we would like to prove about the CSP process corresponding to a class. Therefore, we need to introduce auxiliary variables to capture such properties. For example, an auxiliary variable $bks : \mathbb{N}$ could be added to the class *Marlowe* to model the CSP term $\#tr \downarrow Book$. Initially *bks* would be zero, it would be incremented each time *Book* occurs and remain unchanged each time *Arrive* occurs. Similarly, an auxiliary variable $arrs : \mathbb{N}$ could be added to model the CSP term $\#tr \downarrow Arrive$.

The addition of such variables to a class is possible using Object-Z inheritance. For example, consider the following class *auxMarlowe* which inherits *Marlowe*.



The class *auxMarlowe* includes all the definitions of class *Marlowe* and extends them as follows. The state schema has the additional state variables *bks* and *arrs* and the additional predicate $\#tkt = bks - arrs$. This predicate isn't strictly necessary but aids the proof of the refinement relation between *Marlowe* and *auxMarlowe* as shown in Section 7.3. The initial state schema includes the additional constraint that *bks* and *arrs* are equal to zero and the operations *Book* and *Arrive* increment the variables *bks* and *arrs* respectively.

To prove the property that the number of bookings is greater than or equal to the number of tickets allocated to arriving customers for the class *auxMarlowe*, i.e. *auxMarlowe* **sat** $\#tr \downarrow$

$Book \geq \#tr \downarrow Arrive$, we need to show that the following sequents are valid.

$$\begin{array}{ll} auxMarlowe :: & INIT \vdash bks = 0 \land arrs = 0 \\ auxMarlowe :: & Book \vdash bks' = bks + 1 \land arrs' = arrs \\ auxMarlowe :: & Arrive \vdash bks' = bks \land arrs' = arrs + 1 \\ auxMarlowe :: & \vdash bks \geq arrs \end{array}$$

The first three sequents ensure that *bks* and *arrs* model the number of occurrences of the operations *Book* and *Arrive* respectively. They can easily be proved using the logic for Object-Z (see [36] for examples of proofs in the logic). The final sequent states the desired property. It can be proved by structural induction, i.e. by proving the following sequents.

$$\begin{array}{ll} auxMarlowe :: & INIT \vdash bks \geq arrs \\ auxMarlowe :: & Book \vdash bks \geq arrs \Rightarrow bks' \geq arrs' \\ auxMarlowe :: & Arrive \vdash bks \geq arrs \Rightarrow bks' \geq arrs' \end{array}$$

These sequents can also be easily proved using the logic for Object-Z.

The above can be generalised as follows. A property P of a process corresponding to a class C in terms of the number of occurrences of particular events $Op_1, \ldots, Op_n$,

$$C \textbf{ sat } P(\#tr \downarrow Op_1, \ldots, \#tr \downarrow Op_n)$$

is true when the following sequents are valid. (The set of operations of the class are $Op_1, \ldots, Op_m$ where $m \geq n$.)

$$\begin{array}{ll} C :: & INIT \vdash a_1 = 0 \land \cdots \land a_n = 0 \\ C :: & Op_1 \vdash a_1' = a_1 + 1 \land a_2' = a_2 \land \cdots \land a_n' = a_n \\ \vdots & \\ C :: & Op_n \vdash a_1' = a_1 \land \cdots a_{n-1}' = a_{n-1} \land a_n' = a_n + 1 \\ C :: & Op_{n+1} \vdash a_1' = a_1 \land \cdots \land a_n' = a_n \\ \vdots & \\ C :: & Op_m \vdash a_1' = a_1 \land \cdots \land a_n' = a_n \\ C :: & \vdash P(a_1, \ldots, a_n) \end{array}$$

Similarly, we can develop rules for proving other types of properties. For example, a CSP predicate in terms of $Op \in ref$ can be replaced by a predicate in terms of $\neg pre\ Op$. For example, we might wish to prove that if there are uncollected tickets, then the box office must be able to issue a ticket to some customer. This property could be specified as

$$\begin{array}{l} BookingSystem \textbf{ sat } (\#tr \downarrow Book > \#tr \downarrow Arrive) \\ \quad \Rightarrow (\exists n, x \bullet Arrive.\{(name,\ n),\ (t,\ x)\} \notin ref) \end{array}$$

which can then be proved by showing that the following Object-Z predicate is true:

$$auxMarlowe :: \vdash bks > arrs \Rightarrow \text{pre } Arrive$$

The use of such rules need to be proved sound. This can be done with respect to the failures semantics of classes presented in Section 3.

7.3. Proving the refinement relations

To show that the property proved for *auxMarlowe* also holds for *Marlowe*, we need to prove the refinement relation $auxMarlowe \sqsubseteq Marlowe$. This can be done using the notion of downward simulation defined in Section 6. To do so we first note that the retrieve relation between *auxMarlowe* and *Marlowe* is simply the identity (which we denote *Id*). Therefore to prove the refinement we have to show that

DS.1 $\forall\, auxMarlowe.\mathrm{STATE};\ Marlowe.\mathrm{STATE} \bullet (\mathrm{pre}\ auxMarlowe.Book$
$\Leftrightarrow \mathrm{pre}\ Marlowe.Book)$
DS.2 $\forall\, auxMarlowe.\mathrm{STATE};\ Marlowe.\mathrm{STATE};\ Marlowe.\mathrm{STATE}' \bullet Marlowe.Book$
$\Rightarrow (\exists\, auxMarlowe.\mathrm{STATE}' \bullet auxMarlowe.Book)$
DS.3 $\forall\, Marlowe.INIT \bullet \exists\, auxMarlowe.\mathrm{INIT} \bullet Id$

together with similar conditions for the operation *Arrive*. Because we have simply added new state variables under the refinement, these conditions are easily discharged.

DS.1: This amounts to showing that

$$
\begin{array}{l}
(name? \notin \mathrm{dom}\ tkt \wedge mpool \neq \varnothing \wedge \#\, tkt = bks - arrs \wedge \\
(\exists\, tkt' : Name \rightarrowtail\!\!\!\!\!\!\!\!\;\; \mapsto Ticket;\ mpool' : \mathbb{P}\ Ticket;\ bks', arrs' : \mathbb{N} \bullet \\
\quad \exists\, t : mpool \bullet tkt' = tkt \cup \{name? \mapsto t\} \wedge mpool' = mpool \setminus \{t\}) \wedge \\
\quad \#\, tkt' = bks' - arrs' \wedge bks' = bks + 1 \wedge arrs' = arrs) \\
\Leftrightarrow \\
(name? \notin \mathrm{dom}\ tkt \wedge mpool \neq \varnothing \wedge \\
(\exists\, tkt' : Name \rightarrowtail\!\!\!\!\!\!\!\!\;\; \mapsto Ticket;\ mpool' : \mathbb{P}\ Ticket \bullet \\
\quad \exists\, t : mpool \bullet tkt' = tkt \cup \{name? \mapsto t\} \wedge mpool' = mpool \setminus \{t\}))
\end{array}
$$

which is easily shown to be true (for example, $\#tkt' = \#tkt + 1 = bks - arrs + 1 = bks' - arrs = bks' - arrs'$).

DS.2: This amounts to showing the following, which again can easily shown to be true.

$$
\begin{array}{l}
(name? \notin \mathrm{dom}\ tkt \wedge mpool \neq \varnothing \wedge (\exists\, t : mpool \bullet tkt' = tkt \cup \{name? \mapsto t\} \\
\quad \wedge\ mpool' = mpool \setminus \{t\})) \\
\Longrightarrow \\
(\exists\, bks', arrs' : \mathbb{N} \bullet \\
\quad name? \notin \mathrm{dom}\ tkt \wedge mpool \neq \varnothing \wedge (\exists\, t : Ticket \bullet tkt' = tkt \cup \{name? \mapsto t\} \\
\quad \wedge\ mpool' = mpool \setminus \{t\}) \wedge \\
\quad \#\, tkt = bks - arrs \wedge \#\, tkt' = bks' - arrs' \wedge bks' = bks + 1 \wedge arrs' = arrs)
\end{array}
$$

DS.3: To prove this, it is sufficient to show the following, which is easily done.

$$\forall\, tkt : Name \rightarrowtail Ticket \mid tkt = \varnothing \bullet \exists\, bks,\ arrs : \mathbb{N} \mid \#tkt = bks - arrs \wedge bks = arrs = 0$$

The conditions for *Arrive* can be proved in a similar fashion. Hence, *auxMarlowe* $\sqsubseteq$ *Marlowe*. Since we have shown that *auxMarlowe* **sat** $\#tr \downarrow Book \geq \#tr \downarrow Arrive$ we can deduce that *Marlowe* **sat** $\#tr \downarrow Book \geq \#tr \downarrow Arrive$, and hence conclude the proof that the booking system satisfies the desired property. Furthermore, since *Marlowe* $\sqsubseteq$ *Kurbel* and *Marlowe* $\sqsubseteq$ *Schonell* $\setminus$ {*transfer*}, we can also conclude that both the Kurbel and Schonell booking systems satisfy the property.

8. Conclusion

This paper has presented an approach to specifying concurrent systems using a combination of Object-Z and CSP. A common semantic basis allows classes specified in the Object-Z part of the specification to be used directly as processes in the CSP part. The explicit modelling of state in Object-Z facilitates the specification of data structures needed to describe the concurrent components of a system. Furthermore, inheritance allows issues concerning a component's interface with the system into which it is to be placed to be separated from the specification of its intrinsic behaviour. This, together with the explicit mechanisms for modelling concurrency and communication in CSP, leads to system specifications which are more concise and, we believe, more easily comprehensible than those specified using just one of the languages.

We also presented methods for refining and verifying specifications written using the integrated notation. Because we have not modified either of the languages used, we have been able to use existing methods in our approach to refinement and verification in the combined notation. For example, by giving Object-Z classes a CSP semantics, we can use CSP refinement as the refinement relation for the integrated notation. A refinement can be verified by either calculating the failures semantics directly, or by applying standard state-based refinement relations to the Object-Z components.

To verify behavioural properties of the CSP system specification we use the Object-Z logic to prove subsidiary properties of the Object-Z component classes, these properties are then combined by application of CSP laws to deduce the desired behavioural properties of the overall system.

Acknowledgments

Thanks to Howard Bowman, Felix Cornelius, Clemens Fischer, Maritta Heisel and Matthias Weber for interesting discussions about this work and comments on earlier versions [38, 40] of the paper. The first named author was supported by a research fellowship granted by the Alexander von Humboldt-Stiftung, Germany whilst conducting this research. The anonymous referees also made many valuable observations which improved the paper.

Notes

1. LOTOS includes a process algebra part bases on CCS.
2. $S * T$ denotes the set of finite, partial functions from S to T.
3. Infinite histories enable liveness properties of classes to be modelled. Such properties have been ignored in the description of Object-Z in this paper.
4. S^{ω} and S^{*} denote the set of sequences and set of finite sequences, respectively, of elements from the set S.
5. The definition of CSP in [30] uses a different concurreny operator to that used here which is defined in [21]. Concurreny operators defined in [30] use *interface sets* to describe the extent to which two processes must synchronise. It is possible to re-write definitions in one form into the other, and as Roscoe states: the choice of one version or the other is largely a matter of taste [30]. Because our processes are very simple we use the concurreny operator from [21].
6. The additional property stating that a set is refusable if all its finite subsets are refusable in [6] was shown to be unnecessary in [31].
7. Both operations and events represent instantaneous observations of actions which may themselves take time to occur.
8. Our notion of channel is in fact closer to that of a LOTOS gate[3].
9. An Object-Z class with unsatisfiable initial constraints is not given a semantics in this approach. Such degenerate classes are, however, unimplementable and of no practical interest to the specifier.
10. This example is due to an anonymous referee.
11. We adopt the form of axiom 13 from [33] which is equivalent to that in [32] as argued in the appendix of that paper.
12. $s \downarrow c$ denotes the sequence of values v of events of the form $c.v$ in s, e.g. $\langle c.1, a.4, c.3, d.1\rangle \downarrow c = \langle 1, 3\rangle$.
13. In Z when operations occur outside their preconditions, the post-state is undefined.
14. As mentioned in [21], this law is valid provided S and T do not mention refusal sets.

References

1. M. Benjamin, "A message passing system: An example of combining CSP and Z," in J.E. Nicholls (Ed.), *Z User Workshop, Workshops in Computing*, Springer-Verlag, Oxford, 1989, pp. 221–228.
2. E.A. Boiten, H. Bowman, J. Derrick, and M.W.A. Steen, "Viewpoint consistency in Z and LOTOS: A case study," in J. Fitzgerald, C.B. Jones, and P. Lucas (Eds.), *Formal Methods Europe (FME '97)*, Graz, Austria, September 1997, Lecture Notes in Computer Science, Vol. 1313, Springer-Verlag, pp. 644–664.
3. T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems*, Vol. 14, No. 1, pp. 25–29, 1998.
4. C. Bolton, J. Davies, and J.C.P. Woodcock, "On the refinement and simulation of data types and processes," in K. Araki, A. Galloway, and K. Taguchi (Eds.), *International Conference on Integrated Formal Methods 1999 (IFM '99)*, Springer, July 1999, pp. 273–292.
5. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe, "A theory of communicating sequential processes," *Journal of the ACM*, Vol. 31, No. 3, pp. 560–599, 1984.
6. S.D. Brookes and A.W. Roscoe, "An improved failures model for communicating processes," in *Pittsburgh Symposium on Concurrency*, Lecture Notes in Computer Science, Vol. 197, Springer-Verlag, 1985, pp. 281–305.
7. M.J. Butler, "A CSP Approach to Action Systems," Ph.D. Thesis, Oxford University, 1992.
8. M.J. Butler and C.C. Morgan, "Action systems, unbounded nondeterminism, and infinite traces," *Formal Aspects of Computing*, Vol. 7, No. 1, pp. 37–53, 1995.
9. J. Derrick and E.A. Boiten, "Separating component and context specification using promotion," in K. Araki, A. Galloway, and K. Taguchi (Eds.), *International Conference on Integrated Formal Methods 1999 (IFM'99)*, Springer, July 1999, pp. 293–312.
10. J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen, "Weak refinement in Z," in J.P. Bowen, M.G. Hinchey, and D. Till (Eds.), *ZUM'97: The Z formal specification notation*, Lecture Notes in Computer Science, Vol. 1212, Springer-Verlag, Reading, April 1997, pp. 369–388.

11. J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen, "Specifying and refining internal operations in Z," *Formal Aspects of Computing*, Vol. 10, pp. 125–159, 1998.
12. J. Derrick, E.A. Boiten, H. Bowman, and M.W.A. Steen, "Supporting ODP—Translating LOTOS to Z," in *First IFIP International Workshop on Formal Methods for Open Object-based Distributed Systems*, Chapman & Hall, 1996.
13. D. Duke and R. Duke, "Towards a semantics for Object-Z," in D. Bjorner, C.A.R. Hoare, and H. Langmaack (Eds.), *VDM'90: VDM and Z!* Lecture Notes in Computer Science, Vol. 428, Springer-Verlag, 1990, pp. 242–262.
14. R. Duke, G. Rose, and G. Smith, "Object-Z: A specification language advocated for the description of standards," *Computer Standards and Interfaces*, Vol. 17, pp. 511–533, 1995.
15. H. Ehrich, J. Goguen, and A. Sernadas, "A categorical theory of objects as observed processes," in J.W. Bakker, W.P. de Roever, and G. Rozenberg (Eds.), *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, Vol. 489, Springer-Verlag, 1991, pp. 203–228.
16. M. Nielsen et al., "The RAISE language, methods and tools," *Formal Aspects of Computing*, Vol. 1, pp. 85–114, 1989.
17. C. Fischer, "CSP-OZ—A combination of CSP and Object-Z," in H. Bowman and J. Derrick (Eds.), *Second IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, Chapman & Hall, July 1997, pp. 423–438.
18. C. Fischer and G. Smith, "Combining CSP and Object-Z: Finite or infinite trace semantics," in T. Higashino and A. Togashi (Eds.), *FORTE/PSTV'97*, Osaka, Japan, November 1997. Chapman & Hall, pp. 503–518.
19. J. He, "Process refinement," in J. McDermid (Ed.), *The Theory and Practice of Refinement*, Butterworths, 1989.
20. M. Heisel and C. Sühl, "Formal specification of safety-critical software with Z and real-time CSP," in E. Schoitsch (Ed.), *Proceedings 15th International Conference on Computer Safety, Reliability and Security*, Springer, 1996, pp. 31–45.
21. C.A.R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, 1985.
22. ITU Recommendation X.901–904, *Open Distributed Processing—Reference Model—Parts 1–4*, July 1995.
23. C.B. Jones, *Systematic Software Development using VDM*, International Series in Computer Science, Prentice-Hall, 1986.
24. M.B. Josephs, "A state-based approach to communicating processes," *Distributed Computing*, Vol. 3, pp. 9–18, 1988.
25. Formal Systems (Europe) Ltd., *Failures-Divergences Refinement: FDR 2*, Oct. 1997, FDR2 User Manual.
26. I. MacColl, "Specifying interactive systems in Object-Z and CSP," in K. Araki, A. Galloway, and K. Taguchi (Eds.), *International Conference on Integrated Formal Methods (IFM'99)*, Springer-Verlag, 1999, pp. 335–352.
27. B.P. Mahony and J.S. Dong, "Blending Object-Z and timed CSP: An introduction to TCOZ," in K. Futatsugi, R. Kemmerer, and K. Torii (Eds.), *20th International Conference on Software Engineering (ICSE'98)*, IEEE Press, 1998.
28. B.P. Mahony and J.S. Dong, "Sensors and actuators in TCOZ," in J.M.Wing, J.C.P. Woodcock, and J. Davies (Eds.), *World Congress on Formal Methods (FM'99)*, Springer-Verlag, 1999, pp. 1166–1185.
29. R. Milner, *Communication and Concurrency*, International Series in Computer Science, Prentice-Hall, 1989.
30. A.W. Roscoe, *The Theory and Practice of Concurrency*, International Series in Computer Science, Prentice-Hall, 1998.
31. A.W. Roscoe, "An alternative order for the failures model," *Journal of Logic and Computation*, Vol. 3, No. 2, 1993.
32. A.W. Roscoe, "Unbounded nondeterminism in CSP," *Journal of Logic and Computation*, Vol. 3, No. 2, 1993.
33. A.W. Roscoe and G. Barrett, "Unbounded nondeterminism in CSP," in *Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science, Vol. 442, Springer-Verlag, 1989, pp. 160–193.
34. M. Shaw and D. Garlan, "Formulations and formalisms in software architecture," in J. van Leeuwen (Ed.), *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Vol. 1000, Springer-Verlag, 1996, pp. 307–323.

35. G. Smith, "Extending W for Object-Z," in J. Bowen and M. Hinchey (Eds.), *9th International Conference of Z Users*, Lecture Notes in Computer Science, Vol. 967, Springer-Verlag, 1995, pp. 276–295.
36. G. Smith, "Formal verification of Object-Z specifications," Technical Report 95-55, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1995.
37. G. Smith, "A fully abstract semantics of classes for Object-Z," *Formal Aspects of Computing*, Vol. 7, No. 3, pp. 289–313, 1995.
38. G. Smith, "A semantic integration of Object-Z and CSP for the specification of concurrent systems," in J. Fitzgerald, C.B. Jones, and P. Lucas (Eds.), *Formal Methods Europe (FME '97)*, Graz, Austria, Sept. 1997, Lecture Notes in Computer Science, Vol. 1313, Springer-Verlag, pp. 62–81.
39. G. Smith, *The Object-Z Specification Language*, Kluwer Academic Publishers, 2000.
40. G. Smith and J. Derrick, "Refinement and verification of concurrent systems specified in Object-Z and CSP," in M. Hinchey and Shaoying Liu (Eds.), *First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, Hiroshima, Japan, Nov. 1997, IEEE Computer Society, pp. 293–302.
41. J.M. Spivey, *The Z Notation: A Reference Manual*, 2nd Ed., International Series in Computer Science, Prentice-Hall, 1992.
42. C. Sühl, "RT-Z: An integration of Z and timed CSP," in K. Araki, A. Galloway, and K. Taguchi (Eds.), *International Conference on Integrated Formal Methods (IFM'99)*, Springer-Verlag, 1999, pp. 29–48.
43. H. Tej and B. Wolff, "A corrected failure-divergence-model for CSP in Isabelle/HOL," in J. Fitzgerald, C.B. Jones, and P. Lucas (Eds.), *Formal Methods Europe (FME '97)*, Lecture Notes in Computer Science, Vol. 1313, Springer-Verlag, 1997, pp. 318–337.
44. F.W. Vaandrager, "Process algebra semantics for POOL," Technical Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1991.
45. M. Weber, "Combining statecharts and Z for the design of safety-critical systems," in M.-C. Gaudel and J.C.P. Woodcock (Eds.), *FME '96—Industrial Benefits and Advances in Formal Methods*, Lecture Notes in Computer Science, Vol. 1051, Springer-Verlag, 1996, pp. 307–326.
46. J.C.P. Woodcock and S.M. Brien, "W: A logic for Z," in J.E. Nicholls (Ed.), *Z User Workshop*, Workshops in Computing, Springer-Verlag, 1992, pp. 77–98.
47. J.C.P. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*, International Series in Computer Science, Prentice-Hall, 1996.
48. J.C.P. Woodcock and C.C. Morgan, "Refinement of state-based concurrent systems," in D. Bjorner, C.A.R. Hoare, and H. Langmaack (Eds.), *VDM'90: VDM and Z!* Lecture Notes in Computer Science, Vol. 428, Springer-Verlag, 1990.
49. A. Yonezawa and M. Tokoro (Eds.), *Object-Oriented Concurrent Programming*, MIT Press, 1987.