

Structural refinement in Object-Z / CSP

John Derrick¹ and Graeme Smith²

¹ Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.

² Software Verification Research Centre, University of Queensland 4072, Australia

Abstract. State-based refinement relations have been developed for use on the Object-Z components in an integrated Object-Z / CSP specification. However this refinement methodology does not allow the structure of a specification to be changed in a refinement, whereas a full methodology would allow concurrency to be introduced during the development life-cycle. In this paper we tackle these concerns and discuss refinements of specifications written using Object-Z and CSP where we change the structure of the specification when performing the refinement. In particular, we develop a set of structural simulation rules which allow a single Object-Z component to be refined to a number of communicating or interleaved classes. We prove soundness of these rules and illustrate them with a small example.

1 Introduction

There has been an increasing amount of work recently on combining state based languages such as Z[11] and Object-Z[8] with process algebras such as CSP[4] and CCS[6]. The motivation for the work is that these combinations of languages provide a suitable medium for the description of complex systems which involve aspects of both concurrency and non-trivial data structures. Indeed, there are many application areas for such an approach including the obvious area of distributed systems specification.

The combination we are interested in here is the use of Object-Z together with CSP. This combination of languages has been investigated by a number of researchers including Smith [7], Fischer [2] and Mahony and Dong [5]. In this paper we work with the integration described by Smith [7] and Smith and Derrick [10, 9], although the concerns we address are relevant to all combinations of these two languages.

In the integration discussed here Object-Z is used to describe the components of a system, and these are then combined using CSP operators which describe how the components synchronise and communicate. For example, an elevator in a building might be described in this approach as $(\| \|_{n:Name} User_n) \| LiftSys$ where $User_n$ and $LiftSys$ are given as Object-Z classes describing a user and the lift respectively, and the CSP operators $\|$ and $\| \|$ describe the interaction between these components. The combined notation benefits from reusing existing notations, being relatively simple and having a well-defined meaning given by a semantics based upon the failures-divergences semantics of CSP.

Of course as well as specifying systems we need a method of developing them, and there has been considerable work on *refinement* for both state-based languages and process algebras. This work has been applied to the combination of Object-Z and CSP by Smith and Derrick [10, 9] who develop state-based refinement relations for use on the Object-Z components within an integrated specification. Because Object-Z classes have been given a CSP semantics in the combined language, the refinements are compositional so that when a single Object-Z component is refined then so is the overall Object-Z / CSP specification. For example, if we refine the single *LiftSys* component to *LiftSys2* then application of the theory tells us that $(\| \|_{n:Name} User_n) \| LiftSys2$ will be a refinement of $(\| \|_{n:Name} User_n) \| LiftSys$.

However, the rules presented in [10, 9] do not allow the *structure* of the specification to be changed in a refinement. That is, only single Object-Z classes can be refined individually and therefore the structure of the specification, and in particular the use of the CSP operators, has to be fixed at the initial level of abstraction. This is clearly undesirable and in this paper we provide a means to refine the very structure of a specification written in the integrated notation.

In particular, we develop refinement rules that allow us to refine a single Object-Z class into a number of communicating or interleaved classes. For example, we show how to refine the class *LiftSys* into a parallel composition of a class *Lift*, representing the lift, with a class *Con*, representing the lift controller. This approach therefore allows concurrency to be introduced during refinement as and when appropriate rather than having to fix the eventual structure of the implementation early in the development life-cycle. Similar concerns have been addressed by Fischer and Wehrheim [3], however there the emphasis was on using model checking in order to demonstrate refinement between specifications with different structures. Our work compliments this work nicely.

This paper is structured as follows. In Sections 2 and 3 we discuss the integration and refinement of specifications written using Object-Z and CSP. Then in Section 4 we look at refinements between specifications where we change their structure, and we develop rules for introducing the CSP parallel operator $\|$. These rules are illustrated using the elevator case study. In Section 5 we derive similar rules for the CSP interleaving operator $\| \|$. We conclude in Section 6.

2 Combining Object-Z and CSP

In this section we discuss the specification of systems described using a combination of Object-Z and CSP. In general the specification of a system described using these languages comprises three phases. The first phase involves specifying the components using Object-Z, the second involves modifying the class interfaces (e.g. using inheritance) so that they will communicate and synchronise as desired, and the third phase constructs the final system by combining the components using CSP operators.

Since all interaction of system components is specified via the CSP operators a restricted subset of Object-Z is used. In particular, there is no need for object

instantiation, polymorphism, object containment nor any need for the parallel or enrichment schema operators. Similarly not all CSP operators are required. For example, neither piping nor the sequential composition operator are needed. These restrictions help simplify the language and its semantic basis considerably.

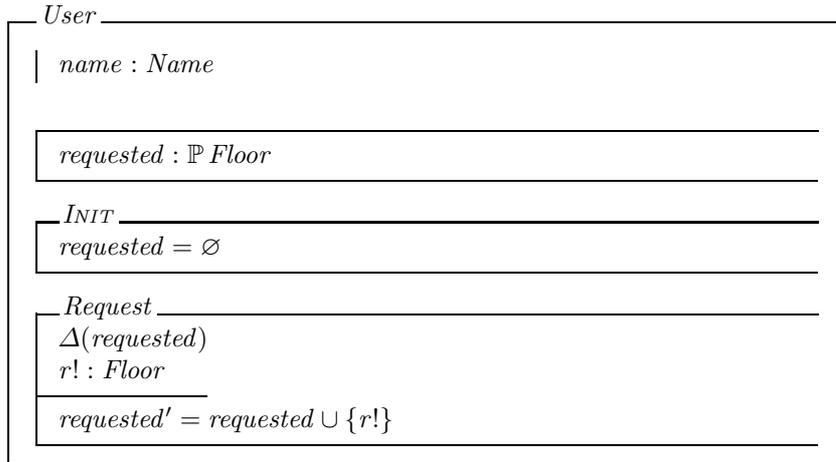
2.1 Example - A lift system

As a simple example we consider the classic lift case study [3]. The components of our system will be the users and the elevator system, and both are specified by Object-Z classes. To do so let *Name* denote the names of all possible users of the system, and suppose the available floors in a building are described as follows.

$$\left| \begin{array}{l} \hline \textit{minFloor}, \textit{maxFloor} : \mathbb{N} \\ \hline \textit{minFloor} < \textit{maxFloor} \end{array} \right.$$

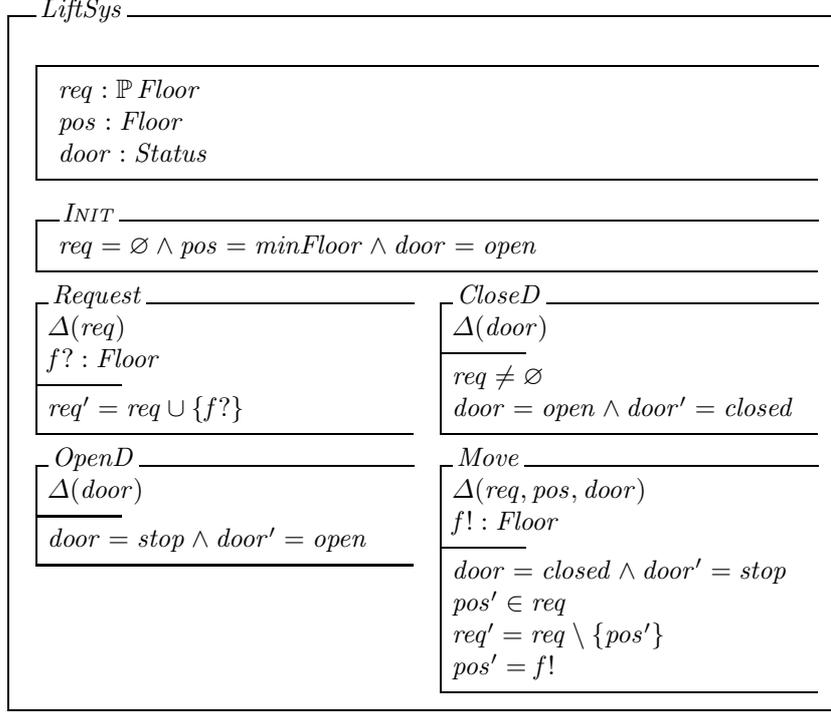
$$\textit{Floor} ::= \textit{minFloor}.. \textit{maxFloor}$$

A single user is capable of one operation: to request a lift to a floor. The state variable *requested* is the set of all floors requested by the user.



Our initial specification also contains the Object-Z class *LiftSys* which describes an abstract view of the elevator system. The class consists of four operations: *Request* models requests for the lift being made by customers, *CloseD* and *OpenD* model the closing and opening of the lift doors respectively, and *Move* describes the movement of the lift inside the shaft. Which request is serviced next is non-deterministic – any valid request is potentially chosen.

$$\textit{Status} ::= \textit{open} \mid \textit{closed} \mid \textit{stop}$$



To specify the complete system we combine the components together in a way which captures their interaction. If we define $User_n$ to be the $User$ class with $name$ instantiated to n [7], then this interaction is given by

$$Building = (\|_{n:Name} User_n) \| LiftSys$$

which describes a single lift with which a number of users can independently interact.

2.2 The semantic model

Combined Object-Z and CSP specifications are given a well-defined meaning by giving the Object-Z classes a failures-divergences semantics identical to that of a CSP process. In a failures-divergences semantics a process is modelled by a triple (A, F, D) where A is its alphabet, F its failures and D 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 .

To give Object-Z classes a failures-divergences semantics the failures of a class are derived from its *histories*, that is from the Object-Z semantic model of a class [9].

In doing so Object-Z operations are mapped to CSP events using the following function which turns an operation op with assignment of values to its

parameters p to the appropriate event:

$$event((op, p)) = op.\beta(p)$$

The meta-function β replaces each parameter name in p by its basename, i.e., it removes the $?$ or $!$. Thus the event corresponding to an operation (op, p) is a communication event with the operation name op as the channel and an assignment of values to the basenames of the operation's parameters as the value passed on that channel. For example, the event corresponding to a user requesting a floor f is $Request.\{(r, f)\}$.

Since Object-Z does not allow hiding of operations (hiding is only possible at the CSP level), divergence is not possible within a component. Therefore a class is represented by its failures together with empty divergences.

As well as giving a well-defined meaning to a combined Object-Z / CSP specification, the semantics also allows a coherent theory of refinement to be developed. We discuss this in the next section.

3 Refinement in Object-Z and CSP

With our integrated semantics, refinement is based on CSP failures-divergences. Thus a specification C is a refinement of a specification A if

$$failures\ C \subseteq failures\ A\ \text{and}\ divergences\ C \subseteq divergences\ A$$

and if we are considering a single Object-Z component we need consider only the failures since its divergences will be empty as noted above.

However, calculating the failures of a system is not practical for anything other than small specifications. To make the verification of refinements tractable we can adapt state-based verification techniques for use in our combined notation, and in particular adapt the idea of upward and downward simulations used in Z [12]. This allows refinements to be verified at the specification level, rather than working explicitly in terms of failures, traces and refusals at the semantic level.

The use of simulations between Object-Z components in the integrated notation is described by Smith and Derrick in [10, 9]. In a simulation, a retrieve relation Abs links the abstract state ($AState$) and the concrete state ($CState$), and, for example, the definition of a downward simulation is as follows.

Definition 1 *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.

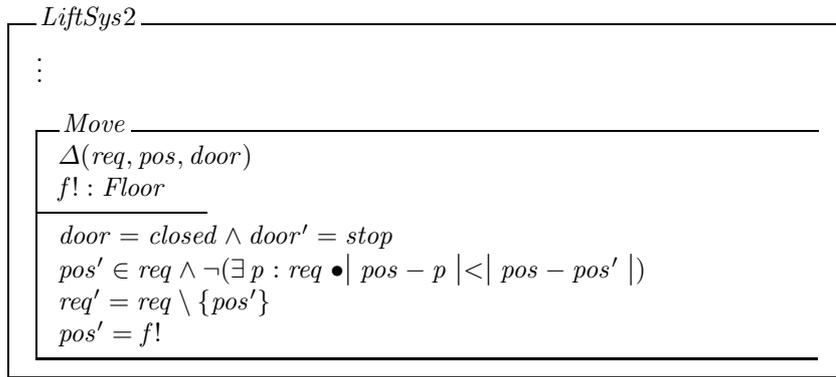
DS.1 $\forall AState; CState \bullet Abs \implies (pre\ AOp \iff pre\ COp)$

DS.2 $\forall AState; CState; CState' \bullet Abs \wedge COp \implies (\exists AState' \bullet Abs' \wedge AOp)$

DS.3 $\forall CInit \bullet \exists AInit \bullet Abs$

Not all refinements change the state space, those that do not are called *operation* refinements as opposed to *data* refinements and these can be verified with a retrieve relation which is the identity (thus simplifying the refinement rules).

The simulation rules allow a single Object-Z class to be refined by another. For example, we might refine the *LiftSys* component to *LiftSys2*. This new lift system is identical to the initial specification except that our *Move* operation is more deterministic and chooses the nearest requested floor instead of an arbitrary one.



This refinement can be verified in the standard way using a downward simulation, and since simulations are together sound and complete with respect to CSP failures-divergences refinement, $(\parallel_{n:Name} User_n) \parallel LiftSys2$ is a refinement of $(\parallel_{n:Name} User_n) \parallel LiftSys$.

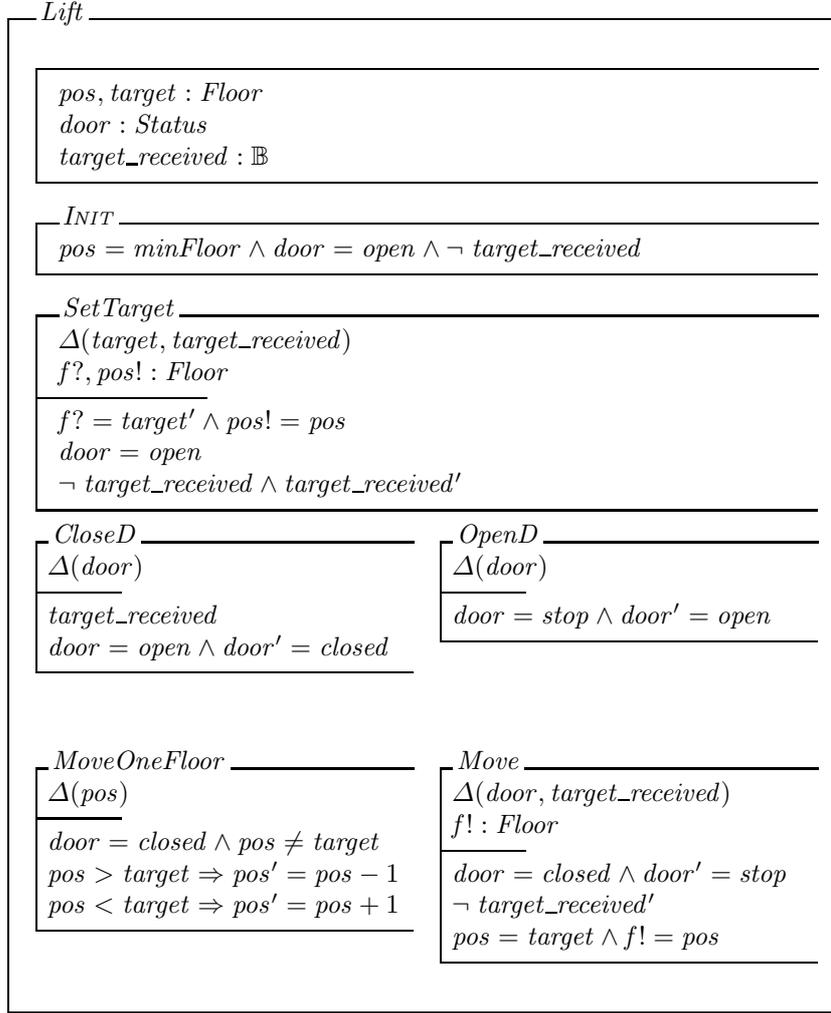
However, this refinement is between single Object-Z classes and simulations do not allow us to change the overall structure of the specification. To understand the problem consider the following example.

3.1 Example - changing the structure of the lift specification

In an implementation we wish to refine the lift system into two separate components (a lift and a controller) which reflect more accurately the underlying physical configuration. The *Lift* class will control the position and movement of the lift, whilst the controller *Con* will marshal the requests and determine the next floor that the lift should service.

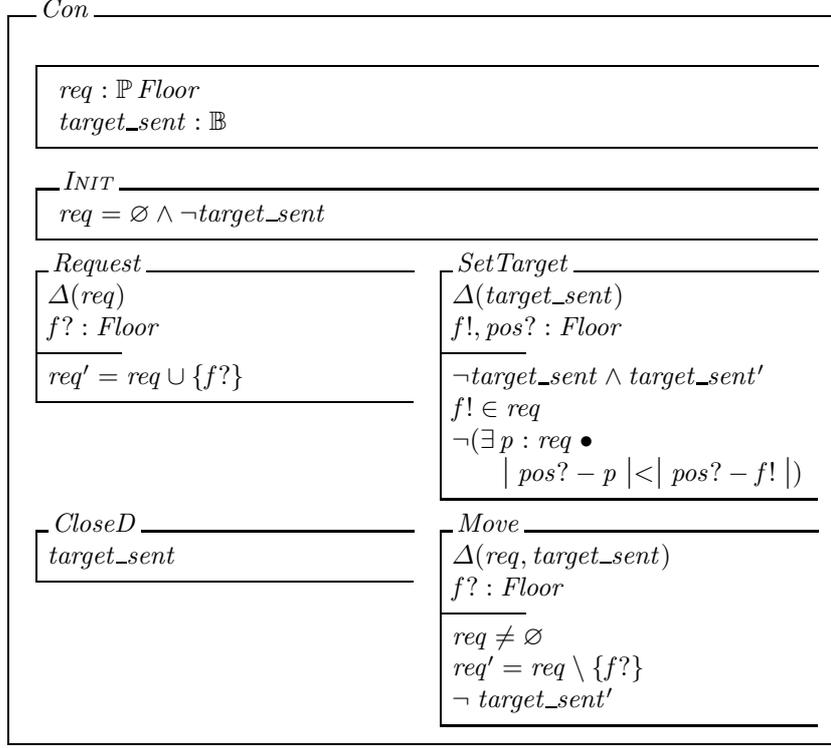
The *Lift* class consists of a position (*pos*), a *door* and a current *target*, and the class *Con* keeps track of the current requests (*req*). The two classes communicate in order to determine the current position and the new target floor.

The *Lift* class is as follows.



The controller accepts requests and determines which floor the lift should go to next. Now, instead of being completely non-deterministic, the floor closest to the current position is chosen. To achieve this, the two classes communicate when performing the *SetTarget* operation. A similar communication takes place in the *Move* operation to determine which floor has been reached. We have also changed the granularity of *Move* from *LiftSys* by using *MoveOneFloor* to move the lift one floor at a time. Since Object-Z operations are blocked outside their preconditions, *MoveOneFloor* can only happen a finite number of times in a row.

Note also that neither class contains the complete temporal ordering of operations. This will be determined by the final synchronisation between the two classes.



It is then possible to show (e.g. by a calculation of failures-divergences) that *LiftSys* is refined by $(Lift \parallel Con) \setminus \{SetTarget, MoveOneFloor\}$ ¹. However, we cannot use simulations to verify the refinement. Furthermore, if we compare the *LiftSys* component with those given by *Lift* and *Con* we cannot even claim that individually the latter classes refine *LiftSys*. In particular,

- the classes are not *conformal*, i.e. neither *Lift* nor *Con* contain all the operations in *LiftSys*, yet they also contain additional operations such as *SetTarget*;
- the new operations have additional inputs and outputs, and
- the behaviour of the operations is different, e.g. the preconditions have been changed.

Yet clearly *LiftSys* is refined by $(Lift \parallel Con) \setminus \{SetTarget, MoveOneFloor\}$, and what we seek to do is to derive state-based techniques that allow us to verify refinements like these without having to expand the synchronisation between the two classes and then calculate their failures. The next section discusses how we can do this.

¹ Throughout this paper we use the shorthand $C \setminus \{Op\}$ to denote the hiding of all events corresponding to the operation *Op*. In general, the names of these events will consist of, as well as the name of the operation, a value corresponding to the mapping of the operation's parameters to their values.

4 Structural Refinement

In this section, we present simulation rules that allow us to prove refinements between Object-Z classes and CSP expressions involving more than one Object-Z class. Specifically, we look at rules for introducing the CSP parallel operator \parallel .

In Smith and Derrick [10, 9], simulation rules which correspond to failures-divergences refinement were presented for refining an Object-Z class to another Object-Z class. To build on this work, we show, in this section, how to construct an Object-Z class which is semantically equivalent to a CSP expression involving parallel composition and hiding. This constructed class, and hence the equivalent CSP expression, can be shown to be a refinement of another class using the existing simulation rules.

From the relationship between the schemas of the constructed class and those of the component classes of the CSP expression, we can also re-express the existing simulation rules in terms of schemas of the component classes.

Figure 1 illustrates the process. We wish to refine a class A into $(D \parallel B) \setminus \{x_1, \dots, x_n\}$. To do so we show that $D \parallel B$ is failures-divergences equivalent to the Object-Z class E and that $E \setminus \{x_1, \dots, x_n\}$ is failures-divergences equivalent to the class C , and then, from the existing simulation rules, we derive alternative simulation rules to show that C is a downward simulation of A . The usefulness of the approach is that these alternative rules are expressed in terms of the original classes B and D , thus these rules allow us to verify the refinement without constructing the semantically equivalent class C .

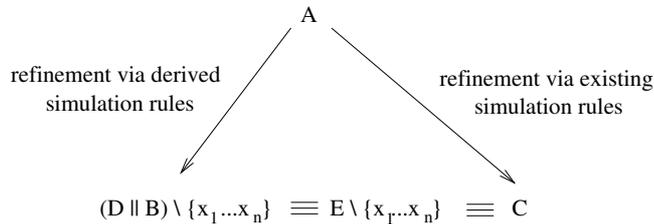


Figure 1. Approaches to refinement

In Section 4.1 we show how to construct the semantically equivalent class and in Section 4.2 we derive the simulation rules in terms of the component classes of the CSP expression. In Section 4.4 we prove the refinement of the Lift example from Section 3. The ideas are extended to the CSP interleaving operator $\parallel\parallel$ in Section 5.

4.1 Constructing an equivalent class

Our approach works for refinements of a class A into specifications of the form $(D \parallel B) \setminus \{x_1, \dots, x_n\}$ where classes D and B and events x_1, \dots, x_n are restricted as follows.

1. The variables declared in the state schema of class D are distinct from those declared in the state schema of class B .
2. Any operations common to D and B (i.e. they have the same operation name) have parameters with identical basenames (i.e., apart from the '?'s and '!s).
3. Each hidden event x_i , $i \in 1 \dots n$, must, in one of the classes D or B , occur a finite number of times immediately before a visible event y corresponding to one particular operation and not at any other time.

In such cases, the finite sequence of hidden events followed by the event y represents an operation refinement of an event y of the abstract specification.

4. When an operation name is shared by D and B , an input in one of the operations with the same basename as an output in the other cannot be constrained more than the output. That is, given that Op in D has input $x?$ and predicate p and Op in B has output $x!$ and predicate q , the following must hold.

$$\exists BState, BState' \bullet q \Rightarrow \exists DState; DState' \bullet p[x!/x?]$$

where $DState$ and $BState$ are the state schemas of classes D and B respectively, and $p[x!/x?]$ is the predicate p with all free occurrences of $x?$ renamed to $x!$.

These restrictions are in fact entirely natural consequences of the events x_1, \dots, x_n acting as a communication medium between the two classes.

Restriction 1 allows us to derive simulation rules expressed as rules on the two separate classes. Restriction 2 says that operations common to D and B will communicate on common channels, and restriction 3 stops divergence due to infinite sequences of hidden events.

Restriction 4 allows us to construct an equivalent class by combining same-named operations with the Object-Z associative parallel composition operator $\parallel_!$ [8]. This operator conjoins its argument operations and renames any inputs in one operation for which there exists a common-named output in the other operation to an output. The common-named parameters are hence identified in the conjoined operation and exist as an output.

To see why the restriction is needed, consider the following same-named operations from classes D and B .

$$\begin{array}{|l} \hline Op \\ \hline x? : \mathbb{N} \\ \hline x? \leq 5 \\ \hline \end{array} \quad \begin{array}{|l} \hline Op \\ \hline x! : \mathbb{N} \\ \hline x! \leq 10 \\ \hline \end{array}$$

When combined, the operations communicate via their parameters. The predicate of the operation from D , that with the input, places a stronger condition on the communicated value than the predicate of the operation from B (thus restriction 4 is not satisfied). The result is that the combined operation can occur with the communicated value less than or equal to 5.

Now consider refining the operation in B to the following.

Op
$x! : \mathbb{N}$
$5 < x! \leq 10$

This is possible since refinement allows conditions on outputs to be strengthened [9]. However, now the combined operation can never occur since there is no value of the communicated variable which satisfies both the operation in D and the operation in B . Hence, despite the individual classes D and B being refined, the resulting equivalent class is not refined (since we have effectively increased the refusals for any trace after which Op could have been performed). Restriction 4 prevents this situation from occurring.

We will now show how to construct an equivalent class C for the CSP expression $(D \parallel B) \setminus \{x_1, \dots, x_n\}$ by considering first parallel composition and then hiding.

Parallel composition Consider classes D and B below where $i, j, k \in \mathbb{N}$ and $j \leq i$.

D <hr/> $DState$ $DINIT$ Op_1 \vdots Op_i	B <hr/> $BState$ $BINIT$ Op_{i-j+1} \vdots Op_k
--	--

When $j \neq 0$, the classes share the operation names $Op_{i-j+1} \dots Op_i$.

The parallel composition of classes D and B , $D \parallel B$, is semantically equivalent to the following class.

E
$DState \wedge BState$
$DINIT \wedge BINIT$
Op_1
\vdots
Op_k

where for each $n : 1 \dots i - j$, Op_n is defined as in D , and for each $n : i + 1 \dots k$, Op_n is defined as in B , and for each $n : i - j + 1 \dots i$, Op_n is the associative parallel composition of the definition in D with the definition in B , i.e. $D.Op_n \parallel B.Op_n$.

Therefore, for each $n : i - j + 1 \dots i$, due to the $DState$ and $BState$ declaring distinct variables (restriction 1), Op_n in E transforms those variables from $DState$ according to the operation Op_n of D and those variables in $BState$ according to the operation Op_n of B . Furthermore, Op_n in E has parameters with

identical basenames to those in Op_n of D and B . Therefore, the alphabet of E is the union of the alphabets of D and B .

To see why the constructed class E is equivalent to $D \parallel B$, consider deriving the failures of E by the approach outlined in [9]. The failures of E are all traces s and refusal sets $X \cup Y$ where

- s is a trace comprising events corresponding to operations $Op_1 \dots Op_k$,
- X and Y are sets of events corresponding to operations in D and B respectively,
- after s , since $DState$ is only changed by events corresponding to operations of D (due to $DState$ and $BState$ declaring distinct variables), X includes only those events that can be refused by D after undergoing trace s restricted to the alphabet of D ,
- similarly, Y includes only those events that can be refused by B after undergoing trace s restricted to the alphabet of B .

Hence,

$$\begin{aligned} failures(E) = \{s, X \cup Y \mid & s \in alphabet(D) \cup alphabet(B) \\ & \wedge (s \triangleright alphabet(D), X) \in failures(D) \\ & \wedge (s \triangleright alphabet(B), Y) \in failures(B)\} \end{aligned}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $(D \parallel B)$ as given by Hoare [4].

Hiding Consider the class E below where Op_2 occurs $m : \mathbb{N}$ times before each occurrence of Op_3 and not at any other time.

E
$EState$
$EINIT$
Op_1
Op_2
Op_3

The CSP expression which hides Op_2 in E , i.e., $E \setminus \{Op_2\}$, is semantically equivalent to the following class². ‘ Op_3 ’ denotes the name of the operation Op_3 in E , as opposed to its definition denoted simply by Op_3 . The effect of ‘ Op_3 ’ is to perform $Op_2 \setminus (x_1, \dots, x_n)$ m times in sequence followed by the original operation Op_3 . (x_1, \dots, x_n are the parameters of Op_2 .)

C
$EState$
$EINIT$
Op_1
‘ Op_3 ’ $\hat{=}$ ($\circledast i : \mathbb{N} \mid i < m \bullet Op_2 \setminus (x_1, \dots, x_n)$) $\circledast Op_3$

² Note that the syntax for hiding operation parameters in Object-Z, $Op \setminus (x)$, is different to that of hiding events in CSP, $P \setminus \{x\}$.

Consider deriving the failures of the constructed class C by the approach outlined in [9]. The failures of C are all traces t and refusal sets X where there exists a failure (s, Y) of E such that s restricted to the events of C is t , and Y includes, as well as the events in X , all events corresponding to Op_2 .

Hence,

$$failures(C) = \{s \triangleright alphabet(C), X \mid (s, X \cup \{Op_2\}) \in failures(E)\}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $E \setminus \{Op_2\}$ as given by Hoare [4]. The definition can be extended for hiding of multiple operations in the obvious way. In fact, what we are using here is a restricted form of weak refinement [1] where we compare in the refinement an abstract operation with a concrete one preceded by a finite number of internal operations.

4.2 Deriving the simulation rules

Given a refinement $(D \parallel B) \setminus \{x_1, \dots, x_n\}$ of A , we could verify the refinement by constructing an equivalent class as outlined in Section 4.1 and using the simulation rules of Smith and Derrick [9]. However, it is preferable not to have to construct an equivalent class but to instead have rules which refer directly to the schemas of the component classes D and B . We now show how we can derive these rules.

Parallel composition We begin by considering the case where we have parallel composition only (and no hiding). For operation names occurring in only one component class, the operation given this name in the constructed class is identical to that in the component class in which it occurs. Hence, the simulation rules are unchanged.

For shared operations, however, the operation in the constructed class is the associative parallel composition of the operations in the component classes. In this case to verify the refinement we can use the downward simulation rules **DS.1** and **DS.2** which, for the communicating operations, require that:

$$\begin{aligned} \mathbf{DS.1}' & \forall AState; DState; BState \bullet \text{pre } AOp \iff \text{pre } (DOp \parallel BOp) \\ \mathbf{DS.2}' & \forall AState; DState; BState; DState'; BState' \bullet \\ & (DOp \parallel BOp) \implies (\exists AState' \bullet AOp) \end{aligned}$$

where $DState$ and $BState$, and DOp and BOp are the two component states and operations respectively.

These rules still involve an operation, $DOp \parallel BOp$, to be constructed from the two classes. However, we can by-pass the necessity to construct this operation as follows. Consider the following operations DOp and BOp (p and q are predicates).

$\frac{DOP}{\begin{array}{l} \Delta(x) \\ x, x' : X \\ z? : Z \end{array}} \quad \frac{BOP}{\begin{array}{l} \Delta(y) \\ y, y' : Y \\ z! : Z \end{array}}$	
p	q

where x and y are the state variables of the two component classes and are distinct (by restriction 1).

The associative parallel composition of these operations is

$\frac{DOP \parallel BOP}{\begin{array}{l} \Delta(x, y) \\ x, x' : X \\ y, y' : Y \\ z! : Z \end{array}}$
$p[z!/z?] \wedge q$

where $p[z!/z?]$ is the predicate p with all free occurrences of $z?$ renamed to $z!$. Hence we can simplify the precondition calculation as follows:

$$\begin{aligned} \text{pre}(DOP \parallel BOP) &\equiv \exists x' : X, y' : Y, z! : Z \bullet p[z!/z?] \wedge q \\ &\equiv \exists x' : X, y' : Y, z! : Z, w! : Z \bullet p[w!/z?] \wedge q \\ &\quad \text{[by restriction 4]} \\ &\equiv (\exists x' : X, w! : Z \bullet p[w!/z?]) \wedge (\exists y' : Y, z! : Z \bullet q) \\ &\quad \text{[since } p[w!/z?] \text{ and } q \text{ refer to distinct variables]} \\ &\equiv \text{pre } DOP[w!/z?] \wedge \text{pre } BOP \end{aligned}$$

In addition, we have

$$DOP \parallel BOP \equiv DOP[z!/z?] \wedge BOP$$

Extrapolating to the general case, we have the following.

$$\begin{aligned} \text{pre}(DOP \parallel BOP) &\equiv \text{pre } DOP[w_1!/z_1?, \dots, w_n!/z_n?] \\ &\quad \wedge \\ &\quad \text{pre } BOP[w_{n+1}!/z_{n+1}?, \dots, w_{n+m}!/z_{n+m}?] \\ DOP \parallel BOP &\equiv DOP[z_1!/z_1?, \dots, z_n!/z_n?] \\ &\quad \wedge \\ &\quad BOP[z_{n+1}!/z_{n+1}?, \dots, z_{n+m}!/z_{n+m}?] \end{aligned}$$

Hence, the simulation rules can be re-expressed as follows.

Definition 2 *Parallel downward simulation*

A CSP expression $D \parallel B$ is an operation downward simulation of the Object-Z class A if D and B satisfy restrictions 1, 2 and 4 (above) and the following hold.

$$\begin{aligned}
\mathbf{PS.1} \quad & \forall AState; DState; BState \bullet \\
& \text{pre } AOp \iff \text{pre } DOp[w_1!/z_1?, \dots, w_n!/z_n?] \\
& \quad \wedge \\
& \quad \text{pre } BOp[w_{n+1}!/z_{n+1}?, \dots, w_{n+m}!/z_{n+m}?] \\
\mathbf{PS.2} \quad & \forall AState; DState; BState; DState'; BState' \bullet \\
& DOp[z_1!/z_1?, \dots, z_n!/z_n?] \\
& \quad \wedge \\
& BOp[z_{n+1}!/z_{n+1}?, \dots, z_{n+m}!/z_{n+m}?] \\
& \implies (\exists AState' \bullet AOp) \\
\mathbf{PS.3} \quad & \forall DInit \wedge BInit \bullet (\exists AInit \bullet \text{true})
\end{aligned}$$

Hiding When we have hiding (as well as parallel composition), for those particular operations which can be preceded by a finite sequence of hidden operations (as required by restriction 3), we replace the operation in the simulation rules with a sequential composition comprising the sequence of hidden operations and the operation. For example, if Op_3 can be preceded by $m : \mathbb{N}$ occurrences of a hidden operation Op_2 (as in the example of Section 4.1), we replace Op_3 in the simulation rules by $(\mathcal{G}i : \mathbb{N} \mid i < m \bullet Op_2 \setminus (x_1, \dots, x_n)) \mathcal{G} Op_3$.

To illustrate this, we consider two cases. The first when one of the hidden events x_i occurs in just one of the classes D and B and has no parameters, and the second when it occurs in both classes and has parameters for communication. Our lift example illustrates both: *MoveOneFloor* only occurs in *Lift* and has no parameters whereas *SetTarget* occurs in *Lift* and *Con* and the parameters $f?$ and $pos!$, and $f!$ and $pos?$ respectively, are used to communicate values.

Case 1. Suppose, without loss of generality, an event x occurs in just one class D in $(D \parallel B) \setminus \{x\}$ and can occur m times in a row where $m \in S \subseteq \mathbb{N}$. Let us denote this operation by Dx . Given Dx has no parameters, we have to show a simulation between AOp and $Int_D \mathcal{G} (DOp \parallel! BOp)$, where Int_D is $\prod m : S \bullet (\mathcal{G}i : \mathbb{N} \mid i < m \bullet Dx)$.

Since the state spaces of D and B are disjoint this can be re-written as $(Int_D \mathcal{G} DOp) \parallel! BOp$, and thus the simulation rules for the operations require that:

$$\begin{aligned}
\mathbf{PS.1} \quad & \forall AState; DState; BState \bullet \\
& \text{pre } AOp \iff \text{pre}(Int_D \mathcal{G} DOp[w_1!/z_1?, \dots, w_n!/z_n?]) \\
& \quad \wedge \\
& \quad \text{pre } BOp[w_{n+1}!/z_{n+1}?, \dots, w_{n+m}!/z_{n+m}?] \\
\mathbf{PS.2} \quad & \forall AState; DState; BState; DState'; BState' \bullet \\
& (Int_D \mathcal{G} DOp[z_1!/z_1?, \dots, z_n!/z_n?]) \\
& \quad \wedge \\
& BOp[z_{n+1}!/z_{n+1}?, \dots, z_{n+m}!/z_{n+m}?] \\
& \implies (\exists AState' \bullet AOp)
\end{aligned}$$

Case 2. Suppose an event x occurs in both classes (e.g. in order to perform a communication as in *SetTarget*) in $(D \parallel B) \setminus \{x\}$ and that it communicates over

channel y . Let Dx denote the operation x in class D etc. Then we have to show a simulation between AOp and $((Dx \parallel Bx) \setminus \{y!\}) \circledast (DOp \parallel BOp)$.

Since the state spaces of D and B are disjoint and assuming $y!$ does not occur as a parameter in DOp and BOp , this can be re-written as $((Dx \circledast DOp) \parallel (Bx \circledast BOp)) \setminus \{y!\}$. Since hiding distributes through pre, the simulation requirements become:

$$\begin{aligned}
\mathbf{PS.1} \quad & \forall AState; DState; BState \bullet \\
& \text{pre } AOp \iff \\
& \quad \exists y! \bullet \text{pre}(Dx \circledast DOp[w_1!/z_1?, \dots, w_n!/z_n?]) \\
& \quad \wedge \\
& \quad \text{pre}(Bx \circledast BOp[w_{n+1}!/z_{n+1}?, \dots, w_{n+m}!/z_{n+m}?]) \\
\mathbf{PS.2} \quad & \forall AState; DState; BState; DState'; BState' \bullet \\
& \exists y! \bullet ((Dx \circledast DOp[z_1!/z_1?, \dots, z_n!/z_n?]) \\
& \quad \wedge \\
& \quad (Bx \circledast BOp[z_{n+1}!/z_{n+1}?, \dots, z_{n+m}!/z_{n+m}?]) \\
& \quad \implies (\exists AState' \bullet AOp)
\end{aligned}$$

These rules can be generalised to multiple events and parameters, as well as to the case when the hidden event x occurs more than once before its corresponding visible event.

4.3 Data refinement

In this section we generalise the rules to cover data refinement. That is, we consider the case when the state space of A is changed when refining this class to $(D \parallel B) \setminus \{x_1, \dots, x_n\}$, and as noted in Section 3, a retrieve relation Abs is used in these circumstances to verify the simulation rules. Our task here then is to determine how this impacts on the structural refinement rules given in Definition 2.

In taking a change of data into account we first note that the construction of the single class C which is semantically identical to $(D \parallel B) \setminus \{x_1, \dots, x_n\}$ is unchanged. Hence the impact of data refinement only occurs in using the simulation rules when verifying the refinement from C to the original component class A . The rules for parallel composition are easily expressed as follows.

$$\begin{aligned}
\mathbf{PS.1} \quad & \forall AState; DState; BState \bullet \\
& Abs \implies (\text{pre } AOp \iff \text{pre } DOp[w_1!/z_1?, \dots, w_n!/z_n?]) \\
& \quad \wedge \\
& \quad \text{pre } BOp[w_{n+1}!/z_{n+1}?, \dots, w_{n+m}!/z_{n+m}?]) \\
\mathbf{PS.2} \quad & \forall AState; DState; BState; DState'; BState' \bullet \\
& Abs \wedge DOp[z_1!/z_1?, \dots, z_n!/z_n?] \\
& \quad \wedge \\
& \quad BOp[z_{n+1}!/z_{n+1}?, \dots, z_{n+m}!/z_{n+m}?] \\
& \quad \implies (\exists AState' \bullet AOp \wedge Abs') \\
\mathbf{PS.3} \quad & \forall DInit \wedge BInit \bullet (\exists AInit \bullet Abs)
\end{aligned}$$

4.4 Example

Using these rules we can now show that *LiftSys* is refined by $(Lift \parallel Con) \setminus \{SetTarget, MoveOneFloor\}$. To do so we must check that the decomposition satisfies the restrictions 1-4 outlined at the start of Section 4.1, and then verify conditions **PS.1-3**.

Restrictions Restriction 1 clearly holds. For restriction 2 we have to compare parameters in *SetTarget*, *CloseD* and *Move* between the two classes *Lift* and *Con*. In each case the basenames in the pairs of operations are the same (e.g. *f* in *Move*).

The hidden operations are *SetTarget* and *MoveOneFloor*. A Boolean variable in each class has been inserted to ensure that *SetTarget* happens once, but only once, before each (visible) *CloseD* operation. Restriction 3 therefore holds for *SetTarget*.

MoveOneFloor is slightly more complicated, because in effect the refinement has decomposed the *Move* in *LiftSys* into a finite number of *MoveOneFloor* operations followed by a *Move* in the classes *Lift* and *Con*. However, restriction 3 is still satisfied because *MoveOneFloor* can only happen a finite number of times (until the lift reaches its *target*), after which a *Move* must happen.

We also have to check the restrictions on the predicates given by 4. Thus, for example, for the *Move* operation we have to check:

$$\begin{aligned} & \exists pos, target, door, pos', target', door' \bullet \\ & \quad door = closed \wedge door' = stop \\ & \quad pos = target \wedge f! = pos \\ \Rightarrow & \\ & \exists req, close, req', close' \bullet req \neq \emptyset \wedge req' = req \setminus \{f!\} \end{aligned}$$

which, since no constraints are being placed on the input, is trivially satisfied.

Simulation rule PS.1 For each operation in *LiftSys* we have to show that either it is a standard refinement if it occurs in just one class, or show that **PS.1** holds if it occurs in both classes.

For the former, *Request* and *OpenD* are identical in the refinement, and both operations appear in just one class.

For operations *CloseD* and *Move* we are going to have to demonstrate that **PS.1** holds, remembering that we have to take into account the hidden operations *SetTarget* and *MoveOneFloor* in doing so.

Consider the *Move* operation. The *MoveOneFloor* hidden event can occur a finite number of times before it, thus according to the above we need to consider the effect of $\bigsqcup m : (0 \dots maxFloor - minFloor) \bullet (\textcircled{\$}i : \mathbb{N} \mid i < m \bullet MoveOneFloor) \textcircled{\$} Move$. Although this sounds complicated, it is not difficult in practice. Simply looking at the behaviour of *MoveOneFloor* and *Move* together shows us that we have to verify (with *Abs* equating the variables *req*, *pos* and *door* of *LiftSys* with those of *Lift* and *Con*):

$$\begin{aligned}
& Abs \wedge \text{pre } LiftSys.Move \Leftrightarrow \\
& \text{pre } Lift.(\sqcap m : (0 \dots maxFloor - minFloor) \bullet \\
& \quad (\exists i : \mathbb{N} \mid i < m \bullet MoveOneFloor) \wp Move) \\
& \wedge \\
& \text{pre } Con.Move
\end{aligned}$$

and this boils down to the trivial

$$door = closed \wedge req \neq \emptyset \Leftrightarrow (door = closed) \wedge (req \neq \emptyset)$$

Simulation rule PS.2 In a similar fashion we must show **PS.2** holds for both *CloseD* and *Move*. For example, for *CloseD* we need to show that

$$\begin{aligned}
& Con.(SetTarget \wp CloseD[pos!/pos?]) \wedge Lift.(SetTarget \wp CloseD[f!/f?]) \\
& \Rightarrow (\exists LiftSysState \bullet LiftSys.CloseD)
\end{aligned}$$

This amounts to showing that

$$\begin{aligned}
& f! \in req \wedge \neg(\exists p : req \bullet |pos? - p| < |pos? - f!|) \\
& door = open \wedge door' = closed \wedge f! = target' \wedge pos! = pos \\
\Rightarrow & \exists LiftSysState \bullet req \neq \emptyset \wedge door = open \wedge door' = closed
\end{aligned}$$

which again is clearly true.

Simulation rule PS.3 This amounts to showing that together the initialisations of *Lift* and *Con* imply the initialisation in *LiftSys*.

Therefore *LiftSys* is refined by $(Lift \parallel Con) \setminus \{SetTarget, MoveOneFloor\}$, and since the hidden events do not occur in $User_n$, $(\parallel_{n:Name} User_n) \parallel LiftSys$ is refined by $(\parallel_{n:Name} User_n) \parallel (Lift \parallel Con) \setminus \{SetTarget, MoveOneFloor\}$.

5 Rules for introducing interleaving

We can also derive rules which allow us to refine a class A into $D \parallel B$, by a similar derivation to the above. Given a CSP expression involving interleaving of the form $D \parallel B$, an equivalent class can be constructed following the approach for parallel composition except that the Object-Z choice operator [8], denoted \sqcap , is used in place of associative parallel composition to combine common-named operations. (Reasoning in terms of failures similar to that for parallel composition can be used to show why this is the case.) Because there is no communication between the components D and B there is no need to impose restriction 4 that was needed for the parallel composition operator.

The choice operator disjoins its arguments adding first to each a predicate stating that variables in the Δ -list of the other operation which are not also in their Δ -list remain unchanged. It also has a requirement that the combined operations have the same parameters.

Given the operations, DOp and BOp of Section 4.2, therefore, the operation in the equivalent constructed class is

$$\frac{\begin{array}{l} DOp \parallel BOp \\ \Delta(x, y) \\ x, x' : X \\ y, y' : Y \\ z?, z! : Z \end{array}}{\begin{array}{l} (p \wedge y' = y) \\ \vee \\ (q \wedge x' = x) \end{array}}$$

Hence, we can simplify preconditions as follows.

$$\begin{aligned} \text{pre}(DOp \parallel BOp) &\equiv \exists x' : X; y' : Y \bullet (p \wedge y' = y) \vee (q \wedge x' = x) \\ &\equiv (\exists x' : X; y' : Y \bullet p \wedge y' = y) \\ &\quad \vee \\ &\quad (\exists x' : X; y' : Y \bullet q \wedge x' = x) \\ &\equiv (\exists x' : X \bullet p) \vee (\exists y' : Y \bullet q) \\ &\quad [\text{since } y' \text{ is not free in } p \text{ and } x' \text{ is not free in } q] \\ &\equiv \text{pre } DOp \vee \text{pre } BOp \end{aligned}$$

In addition, we have

$$\begin{aligned} DOp \parallel BOp &\equiv (DOp \wedge [y' = y]) \vee (BOp \wedge [x' = x]) \\ &\equiv DOp \vee BOp \end{aligned}$$

since y is not in the Δ -list of DOp , and hence we implicitly have $y' = y$ in DOp in the constructed class, and x is not in the Δ -list of BOp .

Hence, the simulation rules can be re-expressed as follows.

Definition 3 *Interleaving downward simulation*

A CSP expression $D \parallel B$ is an operation downward simulation of the Object-Z class A if D and B satisfy restrictions 1-2 and the following hold.

- IS.1** $\forall AState; DState; BState \bullet \text{pre } AOp \iff \text{pre } DOp \vee \text{pre } BOp$
- IS.2** $\forall AState; DState; BState; DState'; BState' \bullet$
 $DOp \vee BOp \implies (\exists AState' \bullet AOp)$
- IS.3** $\forall DInit \wedge BInit \bullet (\exists AInit \bullet \text{true})$

Given restriction 3, these rules can be modified for hiding in exactly the same way as the parallel composition rules. They also extend to data refinement in the obvious way.

6 Conclusions

The purpose of this paper has been to present some refinement rules for developments where we change the structure of a combined Object-Z / CSP specification.

The simulation rules we have developed are sound, since we have verified them with respect to failures-divergences refinement, but they are clearly not yet complete. That is, there will be further refinements of structure that cannot be verified by application of Definitions 2 and 3. One further avenue for this research is to extend these rules to a set of complete structural refinement rules.

One aspect that could be considered are the restrictions placed on the decomposition, and in particular, one could envisage relaxing the third restriction concerning when hidden events can occur. It is likely that a full generalisation would use the ideas of weak refinement [1] here to produce quite arbitrary combined classes.

This work could also be extended to other combinations of Object-Z and CSP, such as those described in [2, 5].

References

1. J. Derrick, E. Boiten, H. Bowman, and M. Steen. Specifying and Refining Internal Operations in Z. *Formal Aspects of Computing*, 10:125–159, December 1998.
2. C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.
3. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *1st International Conference on Integrated Formal Methods*, pages 315–334. Springer-Verlag, 1999.
4. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
5. B.P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.
6. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
7. 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, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
8. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
9. G. Smith. Recursive schema definitions in Object-Z. In A. Galloway J. Bowen, S. Dunne and S. King, editors, *International Conference of B and Z Users (ZB 2000)*, volume 1878 of *Lecture Notes in Computer Science*, pages 42–58. Springer-Verlag, 2000.
10. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M.G. Hinchey and Shaoying Lui, editors, *First International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302. IEEE Computer Society Press, 1997.
11. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.
12. J.C.P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.