Structural refinement in Object-Z / CSP

John Derrick[1] and Graeme Smith[2]

[1] Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK.
[2] Software Verification Research Centre, University of Queensland 4072, Australia

Abstract. In this paper we discuss refinements of specifications written using Object-Z and CSP where we change the structure of the specification when performing the refinement.

Keywords: Object-Z, CSP, refinement.

1 Introduction

There has been an increasing amount of work recently on combining state based languages such as Z[?] and Object-Z[?] with process algebras such as CSP[?] and CCS[?]. 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 Fischer [?], Smith [?] and Mahony and Dong [?]. In this paper we work with the integration described by Smith [?] and Smith and Derrick [?,?], 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 [?,?] 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 *LiftSys*2 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 [**?**,**?**] 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 will show how to refine the single *LiftSys* into $(Lift || Con)$ which consists of a *Lift* operated by a controller *Con*. This approach will therefore allow 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 [**?**], 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 operation refinements between specifications where we change their structure, and we develop rules for introducing the CSP parallel operators $||$ and $|||$. These rules are illustrated using the elevator case study. In Section 5 we generalise this approach to look at data refinement where we change the data representation of the classes, and a vending machine provides a suitable example to illustrate its use. 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. Furthermore, a well-definedness condition is placed on the CSP hiding operator to stop the introduction of unbounded nondeterminism (see [**?**] for a discussion of this aspect).

2.1 Example - A lift system

As a simple example we consider the classic lift case study [**?**]. 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.

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

$$Floor == minFloor..maxFloor$$

A single user is capable of one operation: to request a lift to a different floor, where their current floor is denoted by *position*.

$$\begin{array}{|l}
\textit{User} \\ \hline
\quad \begin{array}{|l} name : Name \end{array} \\
\quad \begin{array}{|l} \hline position : Floor \\ \hline \end{array} \\
\quad \begin{array}{|l} \textit{INIT} \\ \hline position = minFloor \\ \hline \end{array} \\
\quad \begin{array}{|l} \textit{Request} \\ \hline r! : Floor \\ \hline r! \neq position \\ \hline \end{array} \\ \hline
\end{array}$$

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.

$$Status ::= open \mid closed \mid stop$$

LiftSys

$req : \mathbb{P}\, Floor$
$pos : Floor$
$door : Status$

INIT

$req = \varnothing \land pos = minFloor \land door = open$

Request

$\Delta(req)$
$f? : Floor$

$req' = req \cup \{f?\}$

CloseD

$\Delta(door)$

$req \neq \varnothing$
$door = open \land door' = closed$

OpenD

$\Delta(door)$

$door = stop \land door' = open$

Move

$\Delta(req, pos, door)$
$f! : Floor$

$door = closed \land door' = stop$
$pos' \in req$
$req' = req \setminus \{pos'\}$
$pos' = f!$

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 [?], 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 *history*, that is from the Object-Z semantic model of a class [?].

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 parameter 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 upon 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 [**?**]. 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 [**?**,**?**]. 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 $\forall AState;\ CState \bullet Abs \implies (\mathrm{pre}\, AOp \iff \mathrm{pre}\, COp)$
DS.2 $\forall AState;\ CState;\ CState' \bullet Abs \land COp \implies (\exists AState' \bullet Abs' \land AOp)$
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 *LiftSys*2. 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.

*LiftSys*2

⋮

Move

$\Delta(req, pos, door)$
$f! : Floor$

$door = closed \land door' = stop$
$pos' \in req \land \neg(\exists p : req \bullet \mid pos - p \mid < \mid pos - pos' \mid)$
$req' = req \setminus \{pos'\}$
$pos' = f!$

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, $(|||_{n:Name}\, User_n) || LiftSys2$ is a refinement of $(|||_{n:Name}\, User_n) || LiftSys$.

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.

Lift

$$pos, target : Floor$$
$$door : Status$$
$$target\_received : \mathbb{B}$$

INIT

$$pos = minFloor \land door = open$$

SetTarget

$$\Delta(target, target\_received)$$
$$f?, pos! : Floor$$

$$f? = target' \land pos! = pos$$
$$door = open$$
$$\neg\, target\_received \land target\_received'$$

CloseD

$$\Delta(door)$$

$$target\_received$$
$$door = open \land door' = closed$$

OpenD

$$\Delta(door)$$

$$door = stop \land door' = open$$

MoveOneFloor

$$\Delta(pos)$$

$$door = closed$$
$$pos > target \Rightarrow pos' = pos - 1$$
$$pos < target \Rightarrow pos' = pos + 1$$

Move

$$\Delta(door, target\_received)$$
$$f! : Floor$$

$$door = closed \land door' = stop$$
$$\neg\, target\_received'$$
$$pos = target \land f! = pos$$

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 whether the lift has stopped, and if so 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.

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

Con

$$req : \mathbb{P}\, Floor$$
$$target\_sent : \mathbb{B}$$

INIT

$$req = \varnothing \land \neg close$$

Request

$$\Delta(req)$$
$$f? : Floor$$
$$req' = req \cup \{f?\}$$

SetTarget

$$\Delta(target\_sent)$$
$$f!, pos? : Floor$$
$$\neg target\_sent \land target\_sent'$$
$$f! \in req$$
$$\neg(\exists\, p : req \bullet \mid pos? - p \mid < \mid pos? - f! \mid)$$

CloseD

$$\Delta(target\_sent)$$
$$target\_sent \land \neg target\_sent'$$

Move

$$\Delta(req, target\_sent)$$
$$f? : Floor$$
$$req \neq \varnothing$$
$$req' = req \setminus \{f?\}$$
$$\neg\ target\_sent'$$

It is then possible to show (e.g. by a calculation of failures-divergences) that *LiftSys* is refined by $(Lift \| Con) \setminus \{SetTarget, MoveOneFloor\}$[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 $(Lift \| Con) \setminus \{SetTarget, MoveOneFloor\}$ is a refinement of *LiftSys*, 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 $E \setminus \{Op\}$ to denote the hiding of all events corresponding to the operation *Op*. In general, the names of these events will consist of a mapping from the parameters of the operation to their values, as well as the name of the operation.

4 Operation Refinement

In this section, we present simulation rules that allow us to prove operation refinements between Object-Z classes and CSP expressions involving more than one Object-Z class. The rules are extended to data refinement in Section 5.

In Smith and Derrick [**?**,**?**], 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 $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$. To do so we show that $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ is failures-divergences equivalent to the Object-Z class C, and then derive simulation rules to show that C is a downward simulation of A. The usefulness of the approach is that these simulation 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.

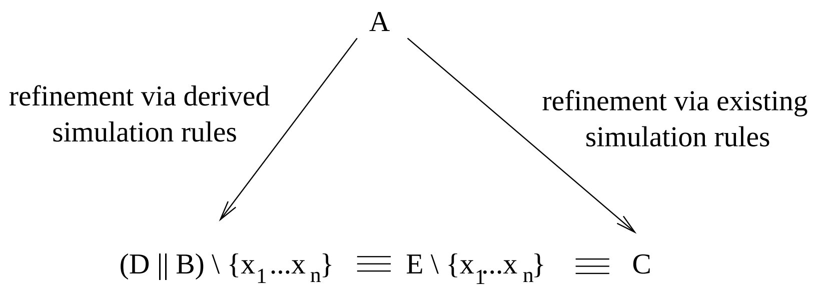

Figure 1. Approaches to operation refinement

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 4.3 we prove the refinement of the Lift example from Section 3. The ideas are extended to the CSP interleaving operator $|||$ in Section 4.4.

4.1 Constructing an equivalent class

Our approach works for refinements of a class A into specifications of the form $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ where classes D and B and events $x_1, \ldots, x_n$ 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, $i \in 1 \mathinner{\ldotp\ldotp} n$, may 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 $x_1, \ldots, x_n$ 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.

The reason for the final restriction can be seen if we consider the following same-named operations from classes D and B.

Op	*Op*
$x? : \mathbb{N}$	$x! : \mathbb{N}$
$x? \leqslant 5$	$x! \leqslant 10$

The two operations are intended to 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 operations can occur with the communicated value less than or equal to 5.

Now consider refining the operation in B to the following.

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

This is possible since refinement allows conditions on outputs to be strengthened [**?**]. However, now the synchronisation of the operations in D and B cannot occur since there is no value of the communicated variable which satisfies both. Hence, despite the individual classes D and B being refined, the resulting composed system 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 $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ by considering first parallel composition and then hiding.

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

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

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

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

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

where for each $n : 1 \mathinner{..} i-j$, Op_n is defined as in D, and for each $n : i+1 \mathinner{..} k$, Op_n is defined as in B, and for each $n : i-j+1 \mathinner{..} i$, Op_n is the associative parallel composition [**?**] of the definition in D with the definition in B, i.e. $D.Op_n \parallel_! B.Op_n$.

The associative parallel composition operator of Object-Z, $\parallel_!$, 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.

Therefore, for each $n : i-j+1 \mathinner{..} i$, due to the $DState$ and $BState$ declaring distinct variables (restriction 1), Op_n in C 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 C has parameters with identical basenames to those in Op_n of D and B. Therefore, the alphabet of C is the union of the alphabets of D and B.

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

- s is a trace comprising events corresponding to operations $Op_1 \ldots Op_k$,
- 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,

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

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

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

$$\begin{array}{|l} \underline{\ E\ } \\ EState \\ EINIT \\ Op_1 \\ Op_2 \\ Op_3 \\ \hline \end{array}$$

The CSP expression which hides Op_2 in E, i.e., $E \setminus \{Op_2\}$, is semantically equivalent to the following class. ('Op_3' denotes the name of the operation Op_3 in E, as opposed to its definition denoted simply by Op_3.)

$$\begin{array}{|l} \underline{\ C\ } \\ EState \\ EINIT \\ Op_1 \\ `Op_3\text{'} \mathrel{\widehat=} Op2 \setminus (p_1, \ldots, p_n) \mathbin{;} \ldots \mathbin{;} Op2 \setminus (p_1, \ldots, p_n) \mathbin{;} Op_3 \\ \hline \end{array}$$

where $p_1, \ldots, p_n$ are the parameters of Op_2.

Consider deriving the failures of the constructed class C by the approach outlined in [?,?]. 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,

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

Since an Object-Z class has no divergences, this is equivalent to the failures of $E \setminus \{Op_2\}$ as given by Hoare [?]. (The definition can be extended for hiding of multiple operations in the obvious way.)

4.2 Deriving the simulation rules

Given a refinement $(D \parallel B) \backslash \{x_1, \ldots, x_n\}$ 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 [?]. 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:

$$
\begin{array}{l}
\mathbf{DS.1}' \ \forall\, AState;\ DState;\ BState \bullet \mathrm{pre}\ \ AOp \iff \mathrm{pre}\ \ (DOp \parallel_! BOp) \\
\mathbf{DS.2}' \ \forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet \\
\qquad\qquad (DOp \parallel_! BOp) \implies (\exists\, AState' \bullet AOp)
\end{array}
$$

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).

$$
\begin{array}{|l}
\multicolumn{1}{l}{DOp} \\
\hline
\Delta(x) \\
x, x' : X \\
z? : Z \\
\hline
p \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{BOp} \\
\hline
\Delta(y) \\
y, y' : Y \\
z! : Z \\
\hline
q \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{DOp \parallel_! BOp} \\
\hline
\Delta(x, y) \\
x, x' : X \\
y, y' : Y \\
z! : Z \\
\hline
p[z!/z?] \wedge q \\
\hline
\end{array}
$$

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:

$$\begin{aligned}
\mathrm{pre}(DOp \parallel_! BOp) &\equiv \exists\, x' : X, y' : Y, z! : Z \bullet p[z!/z?] \land q \\
&\equiv \exists\, x' : X, y' : Y, z! : Z, w! : Z \bullet p[w!/z?] \land q \\
&\qquad \text{[by restriction 4]} \\
&\equiv (\exists\, x' : X, w! : Z \bullet p[w!/z?]) \land (\exists\, y' : Y;\ z! : Z \bullet q) \\
&\qquad \text{[since } p[w!/z?] \text{ and } q \text{ refer to distinct variables]} \\
&\equiv \mathrm{pre}\, DOp[w!/z?] \land \mathrm{pre}\, BOp
\end{aligned}$$

In addition, we have

$$DOp \parallel_! BOp \equiv DOp[z!/z?] \land BOp$$

Extrapolating to the general case, we have the following.

$$\begin{aligned}
\mathrm{pre}(DOp \parallel_! BOp) &\equiv \mathrm{pre}\, DOp[w_1!/z_1?, \ldots, w_n!/z_n?] \\
&\qquad \land \\
&\qquad \mathrm{pre}\, BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?] \\
DOp \parallel_! BOp &\equiv DOp[z_1!/z_1?, \ldots, z_n!/z_n?] \\
&\qquad \land \\
&\qquad BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?]
\end{aligned}$$

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 1-4 (above) and the following hold.

PS.1 $\forall\, AState;\ DState;\ BState \bullet$
$$\mathit{pre}\, AOp \iff \mathit{pre}\, DOp[w_1!/z_1?, \ldots, w_n!/z_n?] \land \mathit{pre}\, BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?]$$
PS.2 $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet$
$$DOp[z_1!/z_1?, \ldots, z_n!/z_n?] \land BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?] \implies (\exists\, AState' \bullet AOp)$$
PS.3 $\forall\, DInit \land BInit \bullet (\exists\, AInit \bullet \mathit{true})$

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, 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 $n : \mathbb{N}$ occurrences of a hidden operation Op_2 (as in the example of Section 4.1), we replace Op_3 in the simulation rules by $Op_2 \setminus (p_1, \ldots, p_n) \fatsemi \ldots \fatsemi Op_2 \setminus (p_1, \ldots, p_n) \fatsemi Op_3$.

There are two cases to consider. The first when one of the hidden events x_i occurs in both classes D and B, and the second when it occurs in just one class. Our lift example illustrates both: *SetTarget* occurs in *Lift* and *Con* whereas *MoveOneFloor* only occurs in *Lift*.

Case 1. Suppose, without loss of generality, an event x occurs in just one class D in $(D \| B) \setminus \{x\}$. Let us denote this operation by Dx. Then Dx is not involved in any communication, and we therefore have to show a simulation between AOp and $Dx \fatsemi (DOp \; \|_! \; BOp)$.

Since the state spaces of D and B are disjoint this can be re-written as $(Dx \fatsemi DOp) \; \|_! \; BOp$, and thus the simulation rules for the operations require that (eliding the quantification over the state spaces):

$$
\begin{array}{l}
\textbf{PS.1} \; \mathrm{pre}\, AOp \iff \mathrm{pre}(Dx \fatsemi DOp[w_1!/z_1?, \ldots, w_n!/z_n?]) \\
\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad \mathrm{pre}\, BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?] \\
\textbf{PS.2} \; (Dx \fatsemi DOp[z_1!/z_1?, \ldots, z_n!/z_n?]) \\
\qquad \wedge \\
\qquad BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?] \\
\qquad\qquad\qquad \implies (\exists\, AState' \bullet AOp)
\end{array}
$$

Case 2. Suppose, without loss of generality, an event x occurs in both classes (e.g. in order to perform a communication as in *SetTarget*) in D in $(D \| B) \setminus \{x\}$ and that it communicates over channel p. Let Dx denote the operation x in class D etc. Then we have to show a simulation between AOp and $((Dx \; \|_! \; Bx) \setminus \{p!\}) \fatsemi (DOp \; \|_! \; BOp)$.

Since the state spaces of D and B are disjoint this can be re-written as $((Dx \fatsemi DOp) \; \|_! \; (Bx \fatsemi BOp)) \setminus \{p!\}$. Since hiding distributes through pre, the simulation requirements become:

$$
\begin{array}{l}
\textbf{PS.1} \; \mathrm{pre}\, AOp \iff \exists\, p! \bullet \mathrm{pre}(Dx \fatsemi DOp[w_1!/z_1?, \ldots, w_n!/z_n?]) \\
\qquad\qquad\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad\qquad\qquad \mathrm{pre}(Bx \fatsemi BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?]) \\
\textbf{PS.2} \; \exists\, p! \bullet ((Dx \fatsemi DOp[z_1!/z_1?, \ldots, z_n!/z_n?]) \\
\qquad\qquad \wedge \\
\qquad\qquad (Bx \fatsemi BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?]) \\
\qquad\qquad\qquad\qquad \implies (\exists\, AState' \bullet AOp)
\end{array}
$$

These rules easily generalise 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. This approach is illustrated in the example of the next section.

4.3 Example

Using these rules we can now show that *LiftSys* is refined by $(Lift \| Con) \setminus \{SetTarget, MoveOneFloor\}$. 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:

$$
\begin{array}{l}
\exists\, pos, target, door, pos', target', door' \bullet \\
\qquad door = closed \land door' = stop \\
\qquad pos = target \land f! = pos \\
\Rightarrow \\
\exists\, req, close, req', close' \bullet req \neq \varnothing \land req' = req \setminus \{f!\}
\end{array}
$$

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 $MoveOneFloor^n \mathbin{;}$ *Move* where $MoveOneFloor^n$ represents n sequential compositions. 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 (for some n):

$$\mathrm{pre}\; LiftSys.Move \Leftrightarrow \mathrm{pre}\; Lift.(MoveOneFloor^n \mathbin{;} Move) \land \mathrm{pre}\; Con.Move$$

and this boils down to the trivial

$$door = closed \land req \neq \varnothing \Leftrightarrow (door = closed) \land (req \neq \varnothing)$$

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

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

This amounts to showing that

$$\begin{aligned}&\quad f! \in req\\ &\quad \neg(\exists\, p : req \bullet \mid pos? - p \mid < \mid pos? - f! \mid)\\ &\quad door = open \land door' = closed\\ &\quad f! = target' \land pos! = pos\\ &\Rightarrow\\ &\quad \exists\, LiftSysState \bullet \;\; req \neq \varnothing \land door = open \land door' = closed\end{aligned}$$

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 $(Lift \| Con) \setminus \{SetTarget, MoveOneFloor\}$, and since the hidden events do not occur in $User_n$ we can conclude that $(|||_{n:Name}\, User_n) \| LiftSys$ is refined by $(|||_{n:Name}\, User_n) \| (Lift \| Con) \setminus \{SetTarget, MoveOneFloor\}$.

4.4 Rules for introducing interleaving

We can also derive rules which allow us to refine a class A into $D \; ||| \; B$, by a similar derivation to the above. Given a CSP expression involving interleaving of the form $D \; |||$ B, an equivalent class can be constructed following the approach for parallel composition except that the Object-Z choice operator[**?**], denoted 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 any of the restrictions that were 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

$$
\begin{array}{|l}
\hline
\textit{DOpBOp} \\
\Delta(x, y) \\
x, x' : X \\
y, y' : Y \\
z?, z! : Z \\
\hline
(p \land y' = y) \\
\lor \\
(q \land x' = x) \\
\hline
\end{array}
$$

Hence, we can simplify preconditions as follows.

$$
\begin{aligned}
\mathrm{pre}(DOpBOp) &\equiv \exists\, x' : X;\ y' : Y \bullet (p \land y' = y) \lor (q \land x' = x) \\
&\equiv (\exists\, x' : X;\ y' : Y \bullet p \land y' = y) \lor (\exists\, x' : X;\ y' : Y \bullet q \land x' = x) \\
&\equiv (\exists\, x' : X \bullet p) \lor (\exists\, y' : Y \bullet q) \\
&\qquad [\text{since } y \text{ is not free in } p \text{ and } x \text{ is not free in } q] \\
&\equiv \mathrm{pre}\, DOp \lor \mathrm{pre}\, BOp
\end{aligned}
$$

In addition, we have

$$
\begin{aligned}
DOpBOp &\equiv (DOp \land [\, y' = y\,]) \lor (BOp \land [\, x' = x\,]) \\
&\equiv DOp \lor BOp \\
&\qquad [\text{since } y \text{ is not in the } \Delta\text{-list of } DOp \text{ and } x \text{ is not in the } \Delta\text{-list of } BOp]
\end{aligned}
$$

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

Definition 3 *Interleaving downward simulation*
A CSP expression $D \mid\mid\mid B$ is an operation downward simulation of the Object-Z class A if the following hold.

IS.1 $\forall\, AState;\ DState;\ BState \bullet pre\, AOp \iff pre\, DOp \lor pre\, BOp$
IS.2 $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet DOp \land BOp \implies (\exists\, AState' \bullet AOp)$
IS.3 $\forall\, DInit \land BInit \bullet (\exists\, AInit \bullet true)$

Should that be [] and not $\land$ in IS.2?

These rules can be modified for hiding in exactly the same way as the parallel composition rules.

5 Data Refinement

In this section we generalise the results from Section 4 to cover data refinement. That is, we consider the case when the state space of A is changed when refining this class to $(D \mid\mid B) \setminus \{x_1, \ldots, x_n\}$, 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 Definitions 2 and 3.

In taking a change of data into account we first note that the construction of the single class C which is semantically identical to $(D \parallel B) \setminus \{x_1, \ldots, x_n\}$ 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.

Next we note that for parallel composition restriction 1 allows us to describe the retrieve relation Abs from A to C as $Abs_D \wedge Abs_B$, where Abs_D is a retrieve relation from D to A, and similarly for Abs_B.

With this in place the rules for parallel composition are easily expressed as follows.

PS.1 $\forall AState; DState; BState \bullet$
$\quad Abs_D \wedge Abs_B \implies (\text{pre } AOp \iff \text{pre } DOp[w_1!/z_1?, \ldots, w_n!/z_n?]$
$\quad \wedge$
$\quad \text{pre } BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?])$

PS.2 $\forall AState; DState; BState; DState'; BState' \bullet$
$\quad Abs_D \wedge DOp[z_1!/z_1?, \ldots, z_n!/z_n?]$
$\quad \wedge$
$\quad Abs_B \wedge BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?]$
$\qquad \implies (\exists AState' \bullet AOp \wedge Abs'_D \wedge Abs'_B)$

PS.3 $\forall DInit \wedge BInit \bullet (\exists AInit \bullet Abs_D \wedge Abs_B)$

For interleaving we place no restrictions on the classes D and B, and therefore use a single retrieve relation linking the state of A to $DState \wedge BState$. The requirements then become:

IS.1 $\forall AState; DState; BState \bullet Abs \implies (\text{pre } AOp \iff \text{pre } DOp \vee \text{pre } BOp)$

IS.2 $\forall AState; DState; BState; DState'; BState' \bullet$
$\quad Abs \wedge DOp \wedge BOp \implies (\exists AState' \bullet AOp \wedge Abs')$

IS.3 $\forall DInit \wedge BInit \bullet (\exists AInit \bullet Abs)$

5.1 Example - A Vending Machine

This example illustrates both interleaving and data refinement, and we decompose a single vending machine into two vending machines acting in parallel. The initial specification allows, for the sake of illustration, up to two coins to be entered and for a coffee to be served for each input.

$2VM$

$\quad money : \{0, 1, 2\}$

$\quad INIT$
$\quad money = 0$

$\quad Coin$
$\quad \Delta(money)$
$\quad money' = money + 1$

$\quad Coffee$
$\quad \Delta(money)$
$\quad money' = money - 1$

We now refine $2\,VM$ into $DVM|||BVM$ where each component DVM and BVM is a simple vending machine that allows just one drink to be dispensed at a time. Both components are in fact given by the same class definition:

$coin : \mathbb{B}$

INIT

$\neg\ coin$

Coin

$\Delta(coin)$

$\neg\ coin \wedge coin$

Coffee

$\Delta(money)$

$coin \wedge \neg\ coin$

In order to verify the refinement we need to use data refinement techniques, and in this example the retrieve relation we use is:

Abs

$2\,VMState$

$DVMState \wedge BVMState$

$money = 0 \Leftrightarrow (\neg\ D.coin \wedge \neg\ B.coin)$

$money = 1 \Leftrightarrow (\neg\ D.coin \wedge B.coin) \vee (D.coin \wedge \neg\ B.coin)$

$money = 2 \Leftrightarrow (D.coin \wedge B.coin)$

To verify the refinement we have to prove conditions **IS.1 - 3**, these are straightforward. For example, **IS.1** for the *Coin* operation requires that we show that

$$Abs \wedge (money \in \{0,1\}) \Leftrightarrow (\neg\ D.coin \vee \neg\ B.coin)$$

The other conditions are equally trivial.

Note that when introducing interleaving in the case of data refinement it is not always possible to split Abs into two retrieve relations $Abs_D \wedge Abs_B$.

6 Conclusions

References