Formal Aspects of Computing (2003) 15: 1-27

Structural refinement of systems specified in Object-Z and 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. This paper is concerned with methods for refinement of specifications written using a combination of Object-Z and CSP. Such a combination has proved to be a suitable vehicle for specifying complex systems which involve state and behaviour, and several proposals exist for integrating these two languages. The basis of the integration in this paper is a semantics of Object-Z classes identical to CSP processes. This allows classes specified in Object-Z to be combined using CSP operators.

It has been shown that this semantic model allows state-based refinement relations to be used on the Object-Z components in an integrated Object-Z / CSP specification. However, the current refinement methodology does not allow the structure of a specification to be changed in a refinement, whereas a full methodology would, for example, 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 single components to be refined to more complex specifications involving CSP operators. The soundness of these rules is verified against the common semantic model and they are illustrated via a number of examples.

Keywords: Integrated formal methods, Object-Z, CSP, refinement.

1. Introduction

There have been a number of proposals recently for *integrated* specification languages which seek to combine two or more existing notations into a single formalism. Of particular interest here are proposals which combine state-based languages such as Z [32] and Object-Z [29] with process algebras such as CSP [19] and CCS [25]. Such combinations of languages are particularly useful for the description of concurrent or distributed systems since their specification requires use of both concurrency and non-trivial data structures.

The particular combination of languages we consider here is the use of Object-Z together with CSP. This combination of languages has been investigated by a number of researchers including Smith [28], Fischer [11] and Mahony and Dong [23]. In this paper, we work with the integration described by Smith [28] and Smith and Derrick [30, 31], however the methodology we describe could be adapted to other combinations of these two languages.

The approach to specification we consider consists of a number of phases whereby Object-Z is used to

Correspondence and offprint requests to: John Derrick, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK

specify the component processes of a system, and CSP is used to describe how the components synchronise and interact. In order to give these integrated specifications a meaning, a common semantic model is used, based upon the failures-divergences semantics of CSP. This is achieved by deriving the failures-divergences of an Object-Z component from the Object-Z history semantics [27] in an intuitive fashion, allowing the two languages to be used together without altering their meaning.

Having a common semantic basis for the two languages also 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 (based upon failures-divergences) as the refinement relation for their combination. However, as a means to verify a refinement it is more convenient to be able to use state-based refinement relations for the Object-Z components, rather than having to directly calculate their failures-divergences. Of course, to do so, one needs to use refinement relations which are compatible with CSP refinement, and in particular, at minimum the refinement relations must be sound with respect to CSP refinement.

Existing work on refinement relations for state-based systems which are sound and complete with respect to CSP refinement includes that of Josephs [21] (similar work due to He [20] and Woodcock and Morgan [37] also appeared around the same time). These results have been applied to combined Object-Z / CSP specifications by Smith and Derrick [30, 31], who develop state-based refinement relations for use on the Object-Z components within an integrated specification.

However, the rules presented in [30, 31] do not allow the *structure* of the specification to be changed in a refinement. By this, we mean that 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.

Consider as an example the process of withdrawing money from a cash point machine. This might be described in the integrated notation as $(|||_{n:Name} Customer_n) \| CashService_0$ where $Customer_n$ and $CashService_0$ are given as Object-Z classes describing a customer and the bank respectively, and CSP is used to describe the interaction between these components. State-based refinement rules can be used to refine $CashService_0$ into another single component *CashService*, say, but there is no tractable method for refining $CashService_0$ into, for example, a number of communicating components. It would clearly be desirable to be able to change the structure of specifications like these under a refinement, and the purpose of this paper is to develop refinement rules to be able to do so. An earlier version of this paper [10] discussed a particular combination of hiding and parallel composition. Here we expand upon those results to consider the use of arbitrary CSP expressions.

To do this, we consider each of the major CSP operators in turn, and show how each operator can be introduced into a refinement. Thus, for example, we give rules for the refinement of a single class A into a parallel composition of classes $B_1 \| B_2$, as well as rules that allow hiding, choice and interleaving to be introduced. We also discuss how such operators could be removed during a refinement step.

The paper is structured as follows. In Section 2, we discuss the integration of Object-Z and CSP into a single specification notation. Section 3 goes on to discuss the refinement of specifications written using these two languages. Subsequent sections (Sections 4 to 7) discuss refinement where we introduce particular CSP operators. The removal of CSP operators during refinement is covered in Section 8. Finally, related work and some conclusions are presented in Section 9.

2. An integration of Object-Z and CSP

In this section, we discuss the specification of systems described using a combination of Object-Z and CSP. We assume the reader is familiar with both Object-Z [29] and CSP [19] for the remainder of this paper.

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.

In the component specification, a restricted subset of Object-Z is used because all the necessary interaction of system components is specified using CSP operators. In particular, we have no need for object instantiation, polymorphism, object containment nor any need for the parallel or enrichment schema operators. For similar reasons, 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. An example - A cash point machine

As an illustrative example, we consider the specification of a bank and an associated cash point machine. At an abstract level the components of our system will be the customers of the bank and the cash service, and both are specified by Object-Z classes.

Let *Name* denote the names of all possible customers of the system. An individual customer is capable of a number of operations, here we just specify one operation: the process of opening an account. The other operations are elided.

Customer

$name : Name$

$account : \mathbb{N}$
$pin : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

INIT

$account = 0$

OpenAcct

$\Delta(account)$
$account? : \mathbb{N}$
$pin! : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$account' = account?$
$pin! = pin$

$\vdots$

The initial specification also contains a single class $CashService_0$ which describes the cash withdrawal component. This is given as an Object-Z class consisting of six operations.

Accounts are opened by the *OpenAcct* operation[1]. Customers can *Insert* a card, the identifier of which is given by *card?*. A four-digit PIN is then given in the *Passwd* operation, and if this matches the account then the customer is able to make a *Request* for money and then *Withdraw* some cash. Requests and withdraws can continue until the card is ejected (via the *Eject* operation). Other operations, such as a deposit facility, are also assumed to be available but we elide their definition.

The partial function *accts* from account numbers to amounts is used to model bank accounts, and $pins(m)$ gives the PIN for a given account m. We need to represent the card currently inside the machine, and we use *card* to do this where 0 represents no card being present. We also record the *amount* of money requested during a transaction and *log* the number of password operations that have been invoked. Finally, we use a variable *mode* to determine whether a transaction is allowed to proceed.

$Mode ::= passwd \mid request \mid withdraw$

[1] We assume that $max(\varnothing) = 0$.

CashService$_0$

$accts : \mathbb{N} \pfun \mathbb{N}$
$pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$card, amount, log : \mathbb{N}$
$mode : Mode$

$\mathrm{dom}\, accts = \mathrm{dom}\, pins$
$0 \notin \mathrm{dom}\, accts$

INIT

$accts = \varnothing \land pins = \varnothing \land card = 0 \land mode = passwd \land log = 0$

OpenAcct

$\Delta(accts, pins)$
$account! : \mathbb{N}$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$account! = (max\, \mathrm{dom}\, accts) + 1$
$accts' = accts \cup \{(account!, 0)\}$
$pins' = pins \cup \{(account!, pin?)\}$

Insert

$\Delta(card)$
$card? : \mathbb{N}$

$card = 0 \land card? \in \mathrm{dom}\, pins$
$card' = card?$

Passwd

$\Delta(mode, log)$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$log' = log + 1$
$card \neq 0 \land mode = passwd$
$(pins(card) = pin?) \Rightarrow mode' = request$
$(pins(card) \neq pin?) \Rightarrow mode' = passwd$

Request

$\Delta(mode, amount)$
$amount? : \mathbb{N}$

$mode = request \land mode' = withdraw$
$amount' = amount?$

Withdraw

$\Delta(accts, mode)$
$money! : \mathbb{N}$

$mode = withdraw \land mode' = request$
$accts' = accts \oplus \{card \mapsto accts(card) - money!\}$
$money! = max\{amount, accts(card)\}$

Eject

$\Delta(card, mode)$

$card' = 0 \land mode' = passwd$

To describe the complete system, we combine the components together in a way which captures their interaction. To do so we define $Customer_n$ to be the *Customer* class with *name* instantiated to n [28], then the required interaction is given by

$$BankSys = (|||_{n:Name} Customer_n) \| CashService_0$$

which describes a single cash point machine with which a number of customers can independently interact.

2.2. A combined semantic model

A specification such as *BankSys* is meaningless unless a semantics has been given to the combined Object-Z and CSP notation. Such a semantics is described in more depth in [28, 31], and here we briefly review the essential features.

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 integrate Object-Z components in a failures-divergences semantics, the failures of an Object-Z class are derived from its *histories* [27]. A history is a non-empty sequence of states with a corresponding sequence of operations. The histories of a class are thus modelled as a set $H \subseteq S^\omega \times O^\omega$, where S is the set of states of the class, and O the set of operations[2]. The operations are instances of the class's operation schemas, represented by the name of the operation together with an assignment of values to its parameters. For example, one history of the class *Customer* is the following (where $n \in Name$):

$$(\langle \{name \mapsto n, account \mapsto 0, pin \mapsto (1,2,3,4)\}, \{name \mapsto n, account \mapsto 7, pin \mapsto (1,2,1,2)\rangle \ , \\ \langle (OpenAcct, \{account? \mapsto 7, pin! \mapsto (1,2,1,2)\})\rangle \ )$$

To relate Object-Z classes and CSP processes, we map Object-Z operations to CSP events. The function *event* relates the two: $event((n,p)) = n.\beta(p)$ where n is an operation name, and p an assignment of values to parameters. The meta-function β replaces parameter names in p by their *basenames*, i.e., the name with the ? or ! decoration removed. 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 the operation in the history above is $OpenAcct.\{`account' \mapsto 7, `pin' \mapsto (1,2,1,2)\}$.

In the following, the function *events* returns the sequence of events associated with an operation sequence, and the function ι returns the assignment of values to the input parameters of an operation.

Based on the approach of [31], the failures of a class C are derived from the histories H of the class as follows. (t, X) is a failure of C if there is a finite history in H such that:

- 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 assignments 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.

Formally, we define[3]:

$$\begin{array}{l} failures(C) = \\ \quad \{(t,X) \mid \exists (s,o) \in H \bullet \\ \qquad s \in S^* \land t = events(o) \land \\ \qquad (\forall\, e \in X \bullet \\ \qquad\quad (\nexists st \in S, op \in O \bullet \\ \qquad\qquad e = event(op) \land (s \frown \langle st \rangle, o \frown \langle op \rangle) \in H) \\ \qquad\quad \lor \\ \qquad\quad (\exists (n,p) \in O, (n,q) \in O \bullet \\ \qquad\qquad \iota(p) = \iota(q) \land e = event((n,p)) \land \\ \qquad\qquad (\exists\, st \in S \bullet \\ \qquad\qquad\quad (s \frown \langle st \rangle, o \frown \langle (n,q) \rangle) \in H) \land \\ \qquad\qquad event(n,q) \notin X))\} \end{array}$$

Since Object-Z does not allow hiding of operations (hiding is only possible at the CSP level), divergence

[2] S^ω denotes the set of (possibly infinite) sequences of elements from the set S.
[3] S^* denotes the set of finite sequences of elements from the set S.

is not possible within a component. Therefore, a class is represented by its failures together with empty divergences.

Although divergence is not possible within a component specified as an Object-Z class, divergence can arise as the result of using hiding in the CSP expression. There is also a related issue due to the possible presence of unbounded nondeterminism.

This arises when, for example, a process can choose from an infinite set of options, and is not supported by the failures-divergences semantics of CSP. This is a problem when using this semantics for Object-Z classes since unbounded nondeterminism can arise naturally.

The issue with unbounded nondeterminism is that, when hidden, unbounded nondeterminism can create divergence which is not captured in the failures-divergences semantics which assumes that the infinite sequences of events can be extrapolated from the finite sequences [26].

There are a number of possible solutions to this issue when combining Object-Z and CSP, and these are discussed in more depth in [31]. The solution we adopt here is to place a well-definedness condition on the hiding operator as is done in [21]. That is, given a process P with failures F, $P \setminus C$ is well-defined only if

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

This prevents unbounded sequences of events being hidden, and therefore stops any divergence being introduced into the specification, whether as a result of unbounded nondeterminism or otherwise. When we refine specifications involving hiding, we will use two conditions which are sufficient to guarantee that this predicate on failures holds.

A representation in terms of failures and divergences can thus be given for any combined Object-Z and CSP specification. This in turn opens the way for a coherent theory of refinement to be developed, and we discuss this in the next section.

3. Refinement in Object-Z and CSP

Refinement is a well-established technique for developing specifications towards an eventual implementation. Refinement is based upon the idea that valid developments are those that reduce non-determinism present in an abstract specification. Thus, refinement is concerned with making implementation choices in parts of a specification which are under-specified.

Different paradigms use different methods to verify that a refinement is correct. In CSP, refinement is defined in terms of failures and divergences [5], and since we have used a failures-divergences semantics for our integrated notation we can define refinement between two specifications as follows. A specification C is a refinement of a specification A if

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

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

In this framework refinements are verified directly by calculating and comparing the failures of the specifications (or if the specifications have the same structure, comparing the failures of the components). However, due to the complexity of the process, calculating the failures of a specification is not practical for any non-trivial specification, so alternative techniques are needed.

The standard approach to making verification tractable is to use techniques borrowed from state-based languages, and in particular the use of upward and downward simulations used in Z and Object-Z [36, 6]. The advantage of these simulation methods is that they allow 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 [30, 31]. In an upward or downward simulation, a retrieve relation Abs links the abstract state ($AState$) and the concrete state ($CState$), and requires that the concrete specification simulates the abstract specification. The simulation is verified on a step-by-step basis, i.e., there are individual conditions for the operations and the initialisation and, in particular, consideration of the traces and failures of the complete specification is not required.

The conditions for a downward simulation are 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 $\forall\, CInit \bullet (\exists\, AInit \bullet Abs)$

DS.2 $\forall\, AState;\ CState \bullet Abs \implies (\mathrm{pre}\, AOp \iff \mathrm{pre}\, COp)$

DS.3 $\forall\, AState;\ CState;\ CState' \bullet Abs \land COp \implies (\exists\, AState' \bullet Abs' \land AOp)$

The conditions for an upward simulation are similar, see for example [3, 9]. The downward and upward simulations are sound and jointly complete with respect to failures-divergences refinement. Although this result originally was due to He, Hoare and Sanders [17], the particular form of the result we use is due to Josephs [21], whose work was set in the context of refinement methods for state-based concurrent systems defined by transition systems where processes do not diverge. Josephs defines downward and upward simulations between CSP processes, and shows that, if we do not have divergence, these are sound and jointly complete with respect to failures-divergences refinement. Since we have restricted ourselves in our language integration to Object-Z classes without internal operations, the Object-Z components we use represent processes which do not diverge as required by [21].

The simulation rules allow a single Object-Z class to be refined by another class. As an example, we might refine the $CashService_0$ component to another class $CashService$ where we model accounts using a sequence as opposed to a partial function. The class would look something like the following (with obvious corresponding changes to the operations):

CashService

$accts : \mathrm{seq}\, \mathbb{N}$
$pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$card, amount, log : \mathbb{N}$
$mode : Mode$

$\mathrm{dom}\, accts = \mathrm{dom}\, pins$

$\vdots$

Such a 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, $(|||_{n:Name} Customer_n) \| CashService$ is a refinement of $(|||_{n:Name} Customer_n) \| CashService_0$.

Refinements that do not change the state space are called *operation* refinements (as opposed to *data* refinements which do potentially alter the state space). Operation refinements can be verified with a retrieve relation which is the identity, thus simplifying the refinement rules.

3.1. Example

In reality, the customer's branch of a bank is not necessarily co-located with the cash point machine being used. Thus we will alter our description to one that is defined in terms of two separate components (a bank and a cash point machine), and we would like to be able to verify this as a refinement of our initial description. The *Bank* component will control the accounts and the transfer of money to individual machines, and the *CashPoint* component will deal with the insertion of the card into the machine and subsequent withdrawal of cash from it. The two separate classes are given as follows, and these will communicate with each other in order to process requests on behalf of the customer.

$Trans ::= idle \mid begin \mid middle \mid end$

Bank

$accts : \mathbb{N} \pfun \mathbb{N}$
$log : \mathbb{N}$

INIT

$accts = \varnothing \land log = 0$

Passwd

$\Delta(log)$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$log' = log + 1$

OpenAcct

$\Delta(accts)$
$account! : \mathbb{N}$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$account! = (max \, \mathrm{dom} \, accts) + 1$
$accts' = accts \cup \{(account!, 0)\}$

Transfer

$\Delta(accts)$
$amt?, acct?, transfer! : \mathbb{N}$

$accts' = accts \oplus \{acct? \mapsto (accts(acct?) - transfer!)\}$
$transfer! = max\{amt?, accts(acct?)\}$

CashPoint

$pins : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$card, funds, amount : \mathbb{N}$
$trans : Trans$

INIT

$pins = \varnothing \land card = 0 \land trans = idle \land funds = 0$

Eject

$\Delta(card, trans)$

$card' = 0 \land trans' = idle$

OpenAcct

$\Delta(pins)$
$account? : \mathbb{N}$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$pins' = pins \cup \{(account?, pin?)\}$

Insert

$\Delta(card)$
$card? : \mathbb{N}$

$card = 0 \land account? \in \mathrm{dom} \, pins$
$card' = card?$

Passwd

$\Delta(trans)$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$card \neq 0 \land trans = idle$
$(pins(card) = pin?) \Rightarrow trans' = begin$
$(pins(card) \neq pin?) \Rightarrow trans' = idle$

Request

$\Delta(trans, amount)$
$amount? : \mathbb{N}$

$trans = begin \land trans' = middle$
$amount' = amount?$

Transfer

$\Delta(funds, trans)$
$amt!, acct!, transfer? : \mathbb{N}$

$trans = middle \land trans' = end$
$acct! = card \land amt! = amount$
$funds' = transfer?$

Withdraw

$\Delta(trans, funds)$
$money! : \mathbb{N}$

$trans = end \land trans' = begin$
$money! = funds$
$funds' = 0$

In the cash machine class, requests are made to withdraw amounts from specified accounts. To perform the withdrawal, the two classes communicate by synchronising on the *Transfer* operation, which passes over some funds allowing the *Withdraw* operation to take place.

Notice that since CSP parallel composition requires identical events in any synchronisation, we have to insert 'dummy' input and output parameters in some of the Object-Z operations such as *Passwd* in *Bank* in order for the synchronisation to proceed.

As an aside, note that in such a description, where two classes are synchronising on a number of operations, neither class necessarily has to contain the complete temporal ordering of operations since this will be determined by the final synchronisation between the two component classes.

Having written our description, we now wish to show that the original $CashService_0$ is refined by $(Bank \| CashPoint) \setminus \{Transfer\}$, since this represents the composition of *Bank* and *CashPoint* with the internal *Transfer* operation hidden from the environment. (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.)

We cannot use the simulation conditions as they stand to verify this refinement since they are defined between two Object-Z components, and in our more concrete specification we have changed the structure of the Object-Z classes and introduced both parallel composition and hiding. In addition, clearly $CashService_0$ is not refined by either of the individual components *Bank* or *CashPoint* since, for example, they are not even *conformal*, i.e., neither *Bank* nor *CashPoint* contain all the operations in $CashService_0$, and they also contain additional operations such as *Transfer*.

However, it should be clear that $CashService_0$ is refined by $(Bank \| CashPoint) \setminus \{Transfer\}$, and the purpose of this paper is to derive state-based techniques that allow us to verify such refinements without having to calculate the failures of the complete specification.

3.2. Deriving structural simulation rules

In order to derive simulation rules between a specification S and a structurally different specification T, we construct Object-Z classes U and V equivalent to S and T respectively. We then re-express the standard simulation rules between U and V in terms of the original specifications S and T, and in particular in terms

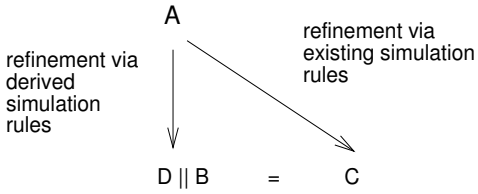


Fig. 1. Approaches to refinement

of their components. This allows us to verify the refinement between S and T in terms of the component classes without having to construct the equivalent classes U and V.

For example, suppose a specification is given as a single Object-Z component A and we wish to refine this to a specification which is given as the parallel composition of two components $D \parallel B$. Our aim is to derive a simulation method that works in terms of A, D and B. To do so we construct an Object-Z class C which is semantically equivalent to $D \parallel B$. A simulation can then be used to verify the refinement between A and C. Finally, we re-express this simulation as a simulation between A, D and B. Figure 1 illustrates the process.

In the remainder of this paper we consider a number of structural refinements, working through the details for each of the major CSP operators. The following table shows the decomposition together with the section that derives the appropriate simulation rules.

A	$D \mid\mid\mid B$	Section 4
A	$D \Box B$, $D \sqcap B$,	Section 5
A	$B \setminus L$	Section 6
A	$D \parallel B$	Section 7
$B_1 \parallel B_2$	$C_1 \parallel C_2$	Section 8

Some of these will require additional restrictions on the components, and we will discuss these in the appropriate section. Structural refinements where we remove CSP operators (e.g., refining $D \mid\mid\mid B$ into A) can also be verified in a similar manner, and in Section 8 we discuss how to do this.

4. Interleaving

To begin, we show how we can refine a class A into a specification involving interleaving, as in $D \mid\mid\mid B$. To do so, we place some syntactic restrictions on D and B as follows.

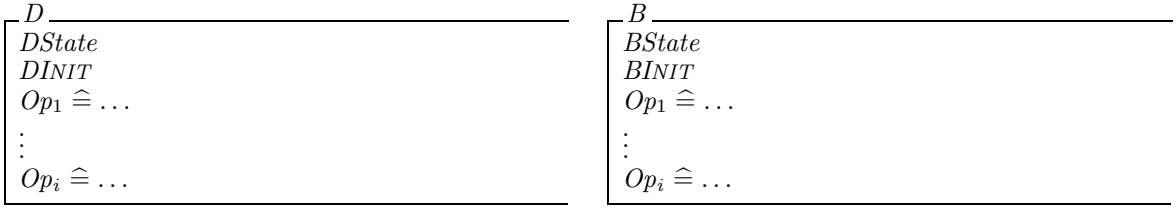
R1 The variables declared in the state schema of class D are distinct from those declared in the state schema of class B.

R2 D and B have the same operation names and, for each common-named operation, the same parameters.

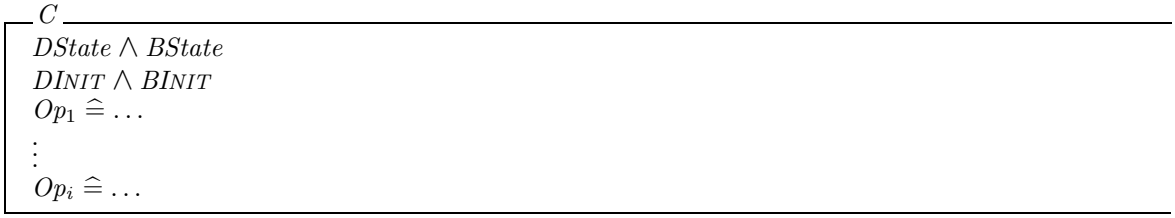
Restriction 1 allows us to derive simulation rules expressed as rules on the two separate classes, and restriction 2 is required since, in CSP, any two interleaved processes must have the same alphabet.

With these restrictions in place, we can now consider how to construct an equivalent class C for the CSP expression $(D \mid\mid\mid B)$ by combining same-named operations with the Object-Z choice operator [] [29]. 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.

Consider classes D and B below where $i \in \mathbf{N}$.



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



where for each $n : 1 \mathinner{..} i$, Op_n is the choice of the definition in D or the definition in B, i.e., $DOp_n [] BOp_n$.

Therefore, for each $n : 1 \mathinner{..} i$, 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 leaving the variables in *BState* unchanged, or transforms those variables in *BState* according to the operation Op_n of B leaving the variables in *DState* unchanged. Furthermore, Op_n in C has the same parameters as those in Op_n of D and B. Therefore, the alphabet of C is the same as the alphabets of D and B.

To see why the constructed class C is equivalent to $D \mid\mid\mid B$, consider deriving the failures of C by the approach outlined in Section 2.2 (see also [31]). The failures of C are all traces s and refusal sets X where

- s is a trace comprising events corresponding to operations $Op_1, \ldots, Op_i$,
- s can be viewed as the interleaving of two traces t and u (denoted s *interleaves*(t, u)), such that
  - t comprises those events where D's operation is chosen,
  - u comprises those events where B's operation is chosen,
- since *DState* is only changed by events in t (due to *DState* and *BState* declaring distinct variables), X includes only events that can be refused by D after undergoing trace t,
- similarly, X includes only events that can be refused by B after undergoing trace u.

Hence,

$$\begin{aligned} \mathit{failures}(C) = \{(s, X) \mid \exists\, t, u \bullet\ & s\ \mathit{interleaves}(t, u) \\ & \land (t, X) \in \mathit{failures}(D) \\ & \land (u, X) \in \mathit{failures}(B)\} \end{aligned}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $(D \mid\mid\mid B)$ as given by Hoare [19].

4.1. Deriving the simulation rules

We now derive simulation rules between A and $(D \mid\mid\mid B)$ in terms of the component classes. We could simply use the downward simulation rules **DS.2** and **DS.3** which, in this case, require that:

$$\textbf{DS.2}' \ \forall\, \mathit{AState};\ \mathit{DState};\ \mathit{BState} \bullet \mathit{Abs} \implies (\text{pre}\ \ AOp \iff \text{pre}\ \ (DOp [] BOp))$$

$$\begin{aligned} \textbf{DS.3}' \ \forall\, & \mathit{AState};\ \mathit{DState};\ \mathit{BState};\ \mathit{DState}';\ \mathit{BState}' \bullet \\ & \mathit{Abs} \land (DOp [] BOp) \implies (\exists\, \mathit{AState}' \bullet AOp \land \mathit{Abs}') \end{aligned}$$

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

These rules still involve an operation, $DOp [] BOp$, 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) :

DOp	BOp
$\Delta(x)$ $x, x' : X$ $z? : Z$	$\Delta(y)$ $y, y' : Y$ $z? : Z$
p	q

where x and y are the state variables of the two component classes and are distinct (by restriction 1). The operation in the equivalent constructed class is

$DOp[]BOp$
$\Delta(x, y)$ $x, x' : X$ $y, y' : Y$ $z? : Z$
$(p \wedge y' = y)$ $\vee$ $(q \wedge x' = x)$

Hence, we can simplify preconditions as follows.

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

In addition, we have

$$DOp[]BOp \equiv (DOp \wedge [\, y' = y \,]) \vee (BOp \wedge [\, x' = x \,])$$

Extrapolating to the general case, we have the following.

$$DOp[]BOp \equiv (DOp \wedge [\, y'_1 = y_1 \wedge \ldots \wedge y'_n = y_n \,]) \vee (BOp \wedge [\, x'_1 = x_1 \wedge \ldots \wedge x'_m = x_m \,])$$

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

Definition 2. Interleaving downward simulation
Let D and B be Object-Z classes containing operations DOp and BOp respectively. Suppose, further, that the Δ-list of DOp is $x_1, \ldots, x_m$ and that of BOp is $y_1, \ldots, y_n$. Then a CSP expression $D \,|||\, B$ is an interleaved downward simulation of the Object-Z class A if D and B satisfy restrictions R1 and R2 and the following hold for some retrieve relation Abs.

IS.1 $\forall\, DInit \wedge BInit \bullet (\exists\, AInit \bullet Abs)$
IS.2 $\forall\, AState;\ DState;\ BState \bullet Abs \implies (\mathrm{pre}\, AOp \iff \mathrm{pre}\, DOp \vee \mathrm{pre}\, BOp)$
IS.3 $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet$
$Abs \wedge ((DOp \wedge [\, y'_1 = y_1 \wedge \ldots \wedge y'_n = y_n \,]) \vee (BOp \wedge [\, x'_1 = x_1 \wedge \ldots \wedge x'_m = x_m \,]))$
$\implies (\exists\, AState' \bullet Abs' \wedge AOp)$

4.2. Example - A Vending Machine

As an example of applying the simulation rules consider the decomposition of 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.

2 *VM*

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

INIT

$money = 0$

Coin	*Coffee*
$\Delta(money)$	$\Delta(money)$
$money' = money + 1$	$money' = money - 1$

We now refine $2\,VM$ into $D|||B$ where $D == VM$ and $B == VM$ and VM is a simple vending machine that allows just one drink to be dispensed at a time.

VM

$coin : \mathbb{B}$

INIT

$\neg\ coin$

Coin	*Coffee*
$\Delta(coin)$	$\Delta(coin)$
$\neg\ coin \land coin'$	$coin \land \neg\ coin'$

In order to verify the refinement, we need to use the following retrieve relation.

Abs

$2\,VMState$
$DState \land BState$

$money = 0 \Leftrightarrow (\neg\ D.coin \land \neg\ B.coin)$
$money = 1 \Leftrightarrow (\neg\ D.coin \land B.coin) \lor (D.coin \land \neg\ B.coin)$
$money = 2 \Leftrightarrow (D.coin \land B.coin)$

Then, to verify the refinement, we have to prove conditions **IS.1 - 3**; these are straightforward. For example, **IS.2** for the *Coin* operation requires that we show that

$$Abs \land (money \in \{0, 1\}) \Longleftrightarrow (\neg\ D.coin \lor \neg\ B.coin)$$

The other conditions are equally trivial.

5. Choice

In this section, we show how the CSP choice operators $\sqcap$ (nondeterministic choice) and $\Box$ (general choice) can be introduced into a specification. In this case, we require only restriction 2 of Section 4. This restriction is required since in CSP the choice operators can only be used with processes which have the same alphabet.

Nondeterministic choice

Consider classes D and B of Section 4. The nondeterministic choice of classes D and B, $D \sqcap B$, is semantically equivalent to the following class.

C

$DState \wedge BState \wedge [\, choice : \{1, 2\} \,]$
$DINIT \wedge BINIT$
$Op_1 \mathrel{\widehat=} \ldots$
$\vdots$
$Op_i \mathrel{\widehat=} \ldots$

where for each $n : 1 \,..\, i$, Op_n is the choice of the definition in D when $choice = 1$ and the definition in B when $choice = 2$:

$$([\, choice = 1 \,] \wedge DOp_n) \,[]\, ([\, choice = 2 \,] \wedge BOp_n)$$

Since the value of choice is not specified initially, it may be either 1 or 2. Furthermore, since it is not changed by any operation, only the events of one of the component classes can ever occur.

Depending on the (fixed) value of *choice*, for each $n : 1 \,..\, i$, Op_n in C transforms those variables from *DState* according to the operation Op_n of D or transforms those variables in *BState* according to the operation Op_n of B. Furthermore, Op_n in C has the same parameters as those in Op_n of D and B. Therefore, the alphabet of C is the same as the alphabets of D and B.

To see why the constructed class C is equivalent to $D \sqcap B$, we construct the failures of C by the approach given in Section 2.2. The failures of C are all traces s and refusal sets X where

- s is a trace comprising events corresponding to operations $Op_1, \ldots, Op_i$,
- when $choice = 1$, since *DState* is only changed by events in s, X includes only events that can be refused by D after undergoing trace s,
- similarly, when $choice = 2$, X includes only events that can be refused by B after undergoing trace s.

Hence,

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

Since an Object-Z class has no divergences, this is equivalent to the failures of $(D \sqcap B)$ as given by Hoare [19].

General choice

Consider again classes D and B of Section 4. The general choice of classes D and B, $D \Box B$, is semantically equivalent to the following class.

C

$DState \wedge BState \wedge [\, choice : \{0, 1, 2\} \,]$
$DINIT \wedge BINIT \wedge [\, choice = 0 \,]$
$Op_1 \mathrel{\widehat=} \ldots$
$\vdots$
$Op_i \mathrel{\widehat=} \ldots$

where for each $n : 1 \,..\, i$, Op_n is the choice of the definition in D when $choice \in \{0, 1\}$ or the definition in B when $choice \in \{0, 2\}$:

$$[\, \Delta(choice) \mid choice \in \{0, 1\} \wedge choice' = 1 \,] \wedge DOp_n$$
$$[]$$
$$[\, \Delta(choice) \mid choice \in \{0, 2\} \wedge choice' = 2 \,] \wedge BOp_n$$

Initially, $choice = 0$ and either D's definition or B's definition can be chosen. Once an operation has occurred, however, *choice* is changed to 1 if D's definition was chosen or 2 if B's definition was chosen. Therefore, all subsequent operations choose the definition corresponding to the same component, i.e., D or B, as the original operation.

Depending on the value of *choice* after the first operation, for each $n : 1 \,..\, i$, Op_n in C transforms those variables from *DState* according to the operation Op_n of D or transforms those variables in *BState* according to the operation Op_n of B. Furthermore, Op_n in C has the same parameters as those in Op_n of D and B. Therefore, the alphabet of C is the same as the alphabets of D and B.

To see why the constructed class C is equivalent to $D \Box B$, we derive the failures of C as before. The failures of C are all traces s and refusal sets X where

- s is a trace comprising events corresponding to operations $Op_1, \ldots, Op_i$,
- initially, since $choice = 0$, X includes only events that can be refused by both B and D,
- when $choice = 1$, since *DState* is only changed by events in s, X includes only events that can be refused by D after undergoing trace s,
- similarly, when $choice = 2$, X includes only events that can be refused by B after undergoing trace s.

Hence,

$$\begin{aligned} failures(C) = \{(s, X) \mid\ & s = \langle\rangle \wedge (s, X) \in failures(D) \cap failures(B) \\ & \vee (s \neq \langle\rangle \wedge (s, X) \in (failures(D) \cup failures(B)))\} \\ = \{(s, X) \mid\ & (\langle\rangle, X) \in failures(D) \cap failures(B) \\ & \vee (s \neq \langle\rangle \wedge (s, X) \in (failures(D) \cup failures(B)))\} \end{aligned}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $(D \Box B)$ as given by Hoare [19].

5.1. Deriving the simulation rules

Following the discussion in Section 4.1 the simulation rules for the nondeterministic choice operator can be expressed as follows.

Definition 3. Nondeterministic choice downward simulation
Let D and B be Object-Z classes containing operations *DOp* and *BOp* respectively. Suppose, further, that the Δ-list of *DOp* is $x_1, \ldots, x_m$ and that of *BOp* is $y_1, \ldots, y_n$. Then a CSP expression $D \sqcap B$ is a nondeterministic choice downward simulation of the Object-Z class A if D and B satisfy restriction R2 and the following hold for a retrieve relation *Abs*.

NCS.1 $\forall\, DInit \wedge BInit \bullet (\exists\, AInit \bullet Abs)$
NCS.2 $\forall\, AState;\ DState;\ BState \bullet$
$Abs \implies (\mathrm{pre}\, AOp \iff (\mathrm{pre}([\, choice = 1\,] \wedge DOp) \vee \mathrm{pre}([\, choice = 2\,] \wedge BOp)))$
NCS.3 $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet$

$$\begin{aligned} Abs \wedge ((&[\, choice = 1\,] \wedge DOp \wedge [\, y_1' = y_1 \wedge \ldots \wedge y_n' = y_n\,]) \\ &\vee \\ &([\, choice = 2\,] \wedge BOp \wedge [\, x_1' = x_1 \wedge \ldots \wedge x_m' = x_m\,])) \\ &\implies (\exists\, AState' \bullet Abs' \wedge AOp) \end{aligned}$$

Similarly, the simulation rules for the general choice operator can be expressed as follows.

Definition 4. General choice downward simulation
Let D and B be Object-Z classes containing operations *DOp* and *BOp* respectively. Suppose, further, that the Δ-list of *DOp* is $x_1, \ldots, x_m$ and that of *BOp* is $y_1, \ldots, y_n$. Then a CSP expression $D \Box B$ is a general choice downward simulation of the Object-Z class A if D and B satisfy restriction 2 and the following hold

for a retrieve relation Abs.

GCS.1 $\forall DInit \land BInit \bullet (\exists AInit \bullet Abs)$
GCS.2 $\forall AState;\ DState;\ BState \bullet$
$$Abs \implies (\mathrm{pre}\, AOp \iff (\mathrm{pre}([\,\Delta(choice) \mid choice \in \{0,1\} \land choice' = 1\,] \land DOp) \lor \mathrm{pre}([\,\Delta(choice) \mid choice \in \{0,2\} \land choice' = 2\,] \land BOp)))$$
GCS.3 $\forall AState;\ DState;\ BState;\ DState';\ BState' \bullet$
$$Abs \land (([\,\Delta(choice) \mid choice \in \{0,1\} \land choice' = 1\,] \land DOp \land [\,y_1' = y_1 \land \ldots \land y_n' = y_n\,]) \lor ([\,\Delta(choice) \mid choice \in \{0,2\} \land choice' = 2\,] \land BOp \land [\,x_1' = x_1 \land \ldots \land x_m' = x_m\,])) \implies (\exists AState' \bullet Abs' \land AOp)$$

These rules can be applied in a fashion similar to those in Definition 2, which were illustrated in Section 4.2.

6. Hiding

In this section, we discuss how hiding can be introduced, i.e., consider how a class A can be refined to the specification $B \setminus L$. In doing this, it turns out that the derived simulation rules we need are, unsurprisingly, rules developed for use with state-based specifications containing internal, i.e., unobservable, operations. Such rules are known as *weak refinement* rules [7, 8].

To understand these rules, suppose the specification $B \setminus L$ refines a class A, where B is the class

B
$BState$
$BINIT$
$Op_1 \mathrel{\widehat=} \ldots$
$\vdots$
$Op_i \mathrel{\widehat=} \ldots$

and L is the set $\{x_1, \ldots, x_m\}$. Then the behaviour of the specification $B \setminus L$ is one where events in the set L are internal and thus can occur before and after any of the remaining (i.e., non-hidden) operations. To represent this as a single class we shall, for the moment, assume there is no divergence due to infinite sequences of hidden events. The single class C equivalent to $B \setminus L$ will use an operation Int_L which represents all possible finite hidden evolution due to events in L. This is constructed as follows [8, 6].

The set of all finite sequences of operations in L is given by $\mathrm{seq}\, L$. The effect of such a sequence is obtained using the operator $\circ$ on a sequence defined, using distributed schema composition, by

$$\overset{\circ}{\langle\,\rangle} = \Xi State$$
$$\overset{\circ}{ops} = \fatsemi_{i : \mathrm{dom}\, ops} \bullet ops(i) \setminus (ops\_param(i)) \qquad \text{for } ops \neq \langle\,\rangle$$

where $ops\_param(i)$ is the list of parameters of the operation $ops(i)$ (such parameters are hidden)[4].

Every possible finite hidden evolution due to events in L is now described by the schema disjunction of the effects of all possible finite sequences of such operations, i.e., $Int_L \mathrel{\widehat=} \exists x : \mathrm{seq}\, L \bullet \overset{\circ}{x}$. So, two states are related by Int_L if and only if there exists a sequence of hidden events x such that the combined effect $\overset{\circ}{x}$ of these operations relates the states.

If $B \setminus L$ contains no divergence, then the CSP expression $B \setminus L$, is semantically equivalent to the following class.

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

C

$BState$
$BINIT \fatsemi Int_L$
$\vdots$
$Op_n \mathrel{\widehat=} Int_L \fatsemi BOp_n \fatsemi Int_L$
$\vdots$

for every $Op_n \notin L$. If $Op_n \in L$ then C contains no defining occurrence of that operation (because its effects are now contained in Int_L).

Now consider deriving the failures of the constructed class C as in Section 2.2. The failures of C are all traces s and refusal sets X where

- there exists a failure (t, Y) of B such that t restricted to the events of C is s
- Y includes, as well as the events in X, all events corresponding to those in L.

Hence,

$$failures(C) = \{(t \rhd alphabet(C), X) \mid (t, X \cup L) \in failures(B)\}$$

Since an Object-Z class has no divergences, this is equivalent to the failures of $B \setminus L$ as given by Hoare [19].

6.1. Deriving the simulation rules

The simulation rules can now be written down for a downward simulation between A and C, and these are as follows[5].

Definition 5. Weak downward simulation
The CSP expression $B \setminus L$ is a weak downward simulation of the Object-Z class A if there is a retrieve relation Abs such that the following hold for every operation not in L.

WS.1 $\forall BState \bullet BInit \fatsemi Int_L \implies (\exists AInit \land Abs)$
WS.2 $\forall AState;\ BState \bullet Abs \implies (\mathrm{pre}\ AOp \iff \mathrm{pre}(Int_L \fatsemi BOp))$
WS.3 $\forall AState;\ BState;\ BState' \bullet$
$Abs \land (Int_L \fatsemi BOp \fatsemi Int_L) \implies \exists AState' \bullet Abs' \land AOp$

where Int_L represents the effect of the hidden events in L as defined above.

These conditions were derived on the basis that $B \setminus L$ contains no divergence, and we discuss this point now. The single class A does not have any hidden events, and thus contains no divergence. Therefore any failures-divergences refinement of A cannot contain any divergence, so $B \setminus L$, as a refinement of A, cannot diverge. Hence, we must ensure that we don't use a class B that could introduce divergence. To do this, two additional conditions are introduced into Definition 5 that prevent the introduction of divergence.

We use a well-founded set WF with a partial order $<$, and a variant Γ which is an expression in the state variables. The variant should always be an element of the set WF, and it should be decreased by each internal operation in the concrete operation. These two conditions can be formulated as:

D1 $Abs \vdash \Gamma \in WF$
D2 $\forall x \in L \bullet Abs \land x \vdash \Gamma' < \Gamma$

Hidden events decrease the variant. However, since there are no constraints on events which are not hidden, a hidden event can occur an infinite number of times, but not in an infinite sequence between observable events.

There are a number of special cases for which the conditions **D1** and **D2** automatically hold. For example, the following restriction is sufficient to ensure no divergence is introduced.

[5] This is identical to the definition of weak downward simulation between two Object-Z specifications when A contains no internal operations, see [7, 8].

H1 Each hidden event $x \in L$ must occur a finite number of times immediately before a visible event y corresponding to one particular operation and not at any other time.

As indicated in Section 2.2 this condition is sufficient to deal with divergence due to hidden events arising from sequences of observable events being hidden.

6.2. Example

As an illustration of these weak simulation rules, we return to our cash point example. One particular refinement we wish to verify is that from $CashService_0$ to $(Bank \| CashPoint) \setminus \{Transfer\}$, and to do this we proceed in two stages (which in fact illustrates a general strategy for this sort of refinement).

The first stage is to refine $CashService_0$ to $CashService_1 \setminus \{Transfer\}$, and then the second stage is to refine this to $(Bank \| CashPoint) \setminus \{Transfer\}$. The component $CashService_1$ will be almost identical to $CashService_0$ except that we introduce operations into the class that will be used as the basis for the subsequent communication between *Bank* and *CashPoint*. That is, $CashService_1$ will contain the operation *Transfer*, but this will be hidden so that the complete specification is a refinement of $CashService_0$. The class $CashService_1$ we use is as follows:

$CashService_1$

$accts_1 : \mathbb{N} \pfun \mathbb{N}$
$log_1 : \mathbb{N}$
$pins_1 : \mathbb{N} \pfun \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$card_1, funds_1, amount_1 : \mathbb{N}$
$trans_1 : Trans$

INIT

$pins_1 = \varnothing \land card_1 = 0 \land trans_1 = idle$
$funds_1 = 0 \land accts_1 = \varnothing \land log_1 = 0$

Passwd

$\Delta(log_1, trans_1)$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$

$log_1' = log_1 + 1$
$card_1 \neq 0 \land trans_1 = idle$
$(pins_1(card_1) = pin?) \Rightarrow trans_1' = begin$
$(pins_1(card_1) \neq pin?) \Rightarrow trans_1' = idle$

OpenAcct

$\Delta(accts_1, pins_1)$
$pin? : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
$account! : \mathbb{N}$

$account! = (max \operatorname{dom} accts_1) + 1$
$pins_1' = pins_1 \cup \{(account!, pin?)\}$
$accts_1' = accts_1 \cup \{(account!, 0)\}$

Request

$\Delta(trans_1, amount_1)$
$amount? : \mathbb{N}$

$trans_1 = begin \land trans_1' = middle$
$amount_1' = amount?$

Transfer

$\Delta(trans_1, accts_1, funds_1)$
$amt!, acct!, transfer! : \mathbb{N}$

$trans_1 = middle \land trans_1' = end$
$acct! = card_1 \land amt! = amount_1$
$funds_1' = transfer!$
$accts_1' = accts_1 \oplus \{acct! \mapsto accts_1(acct!) - transfer!\}$
$transfer! = max\{amt!, accts_1(acct!)\}$

Withdraw

$\Delta(trans_1, funds_1)$
$money! : \mathbb{N}$

$trans_1 = end \land trans_1' = begin$
$money! = funds_1$
$funds_1' = 0$

Insert

$\Delta(card_1)$
$card? : \mathbb{N}$

$card_1 = 0 \land card? \in \mathrm{dom}\, pins_1$
$card_1' = card?$

Eject

$\Delta(card_1, trans_1)$

$card_1' = 0 \land trans_1' = idle$

To verify the refinement we apply Definition 5. This is a simple data refinement here with *Abs* equating a majority of the variables in $CashService_1$ with those of $CashService_0$. The difference between the two state spaces occurs because we have inserted a state to deal with the (internal) *Transfer* operation. Hence the after state of *Withdraw* in $CashService_0$ corresponds to the after state of *Withdraw* in $CashService_1$, but we do not link up the after state of *Transfer* in $CashService_1$ to anything in $CashService_0$.

To see why this is the case, and hence understand the effect of the hidden events, consider the behaviour of the specifications as represented by simple labelled transition systems, as in Figure 2. Here we have elided the behaviour of *OpenAcct*, which can happen at every state, and represented the retrieve relation *Abs* as a dotted line between the states that are identified.

The retrieve relation *Abs* thus simply identifies those states as linked in the figure. For example, the first states in the figure are linked, i.e.,

$$(card = 0 \land mode = passwd) \iff (card_1 = 0 \land trans_1 = idle)$$

and the corresponding after states of *Request* are linked, and so on.

Let us first consider divergence. The variable $trans_1$ has been used to ensure that *Transfer* happens once, but only once, before each non-hidden operation. There is therefore no possibility of divergence being introduced.

The initialisation condition we need to verify is trivial since *Transfer* cannot be applied straight after an initialisation. In order to check the other conditions, we need to consider the effect of the hidden events, i.e., calculate *Int* for this specification. For this particular description this is fairly simple since it is easy to see that *Transfer* can never happen more than one time in a row. Thus:

$$Int \mathrel{\widehat=} \Xi CashService_1\, State \lor Transfer$$

For the operations we have to verify applicability and correctness for all the non-hidden operations (i.e., *Request*, *Eject*, etc). For those operations which cannot be proceeded or followed by an internal event (i.e., occurrence of *Transfer*) applicability and correctness reduce to the same applicability and correctness conditions for systems without internal operations (i.e., the standard definitions). This leaves applicability and correctness for *Request* and *Withdraw*.

For example, applicability for *Withdraw* is

$$Abs \Rightarrow (\mathrm{pre}\ CashService_0.Withdraw \iff \mathrm{pre}\ CashService_1.((\Xi State \lor Transfer) \fatsemi Withdraw))$$

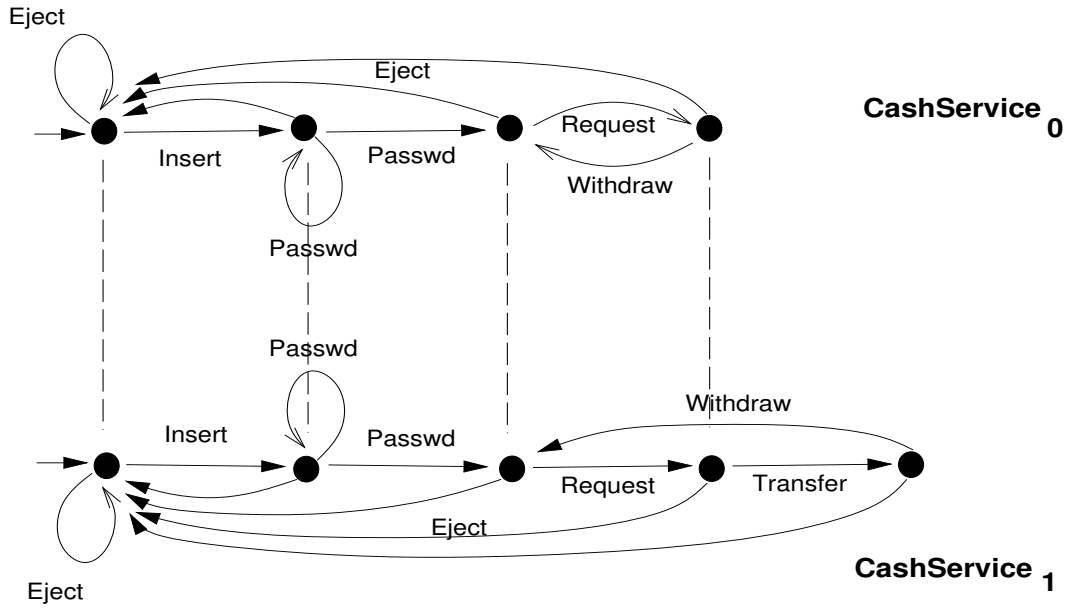


Fig. 2. The behaviour of $CashService_0$ and $CashService_1$.

which holds by virtue of the chosen retrieve relation. Correctness is similar and involves noticing that in *Withdraw* in $CashService_0$

$$money! = max\{amount, accts(card)\}$$

which is equivalent to the effect of *Withdraw* in $CashService_1$:

$$money! = funds_1 \wedge funds_1 = max\{amount, accts(card)\}$$

The remaining conditions are verified in a similar manner.

This example has hidden only one operation, the behaviour of which was relatively simple. An example where the behaviour of events being hidden is more complex is given in [10]. There, some of the hidden events can be invoked an arbitrarily, but finite, number of times between observable operations. This makes the calculation of Int_L more complex in the verification of the weak simulation, and [10] discusses the conditions required.

7. Parallel Composition

We now show how we can refine a class A into a parallel composition $(D \parallel B)$. In addition to restriction 1 from Section 4, we modify restriction 2, and place an additional restriction on D and B as follows.

R2 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).

R3 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. Therefore, given that Op in D has input $x?$ and predicate p and Op in B has output $x!$ and predicate q and $y!$ is not free in p and q, the following must hold.
$p[x!/x?] \wedge q \iff (\exists\, y! \bullet p[y!/x?] \wedge q)$
where $p[x!/x?]$ and $p[y!/x?]$ are the predicates formed from p by replacing all free occurrences of $x?$ with $x!$ and $y!$ respectively.

This generalises to the situation when the operation in D has input $x?$ and output $z!$, and the operation in B has input $z?$ and output $x!$, when the following must hold.

$$p[x!/x?] \land q \iff (\exists\, y! \bullet p[y!/x?] \land q)$$
$$\land$$
$$p \land q[z!/z?] \iff (\exists\, w! \bullet p \land q[w!/x?])$$

Restrictions 2 and 3 allow the construction of an equivalent class by combining same-named operations with the Object-Z associative parallel composition operator $\|_!$ [29]. 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 restriction 3 is needed, consider the following same-named operations from classes D and B.

DOp

$x? : \mathbb{N}$

$x? \leqslant 5$

BOp

$x! : \mathbb{N}$

$x! \leqslant 10$

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 3 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.

BOp

$x! : \mathbb{N}$

$5 < x! \leqslant 10$

This is possible since refinement allows conditions on outputs to be strengthened [31, 6]. 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 3 prevents this situation from occurring.

With these restrictions in place, we can now consider how to construct an equivalent class C for the CSP expression $(D \parallel B)$. The approach is similar to that for $|||$ except we use the associative parallel composition operator [29] in place of the choice operator to combine common-named operations. Consider classes D and B below.

D

DState
DInit
$Op_1 \mathrel{\widehat=} \ldots$
$\vdots$
$Op_i \mathrel{\widehat=} \ldots$

B

BState
BInit
$Op_{i-j+1} \mathrel{\widehat=} \ldots$
$\vdots$
$Op_k \mathrel{\widehat=} \ldots$

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 $\land$ *BState*
DInit $\land$ *BInit*
$Op_1 \mathrel{\widehat=} \ldots$
$\vdots$
$Op_k \mathrel{\widehat=} \ldots$

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

Therefore, for each $n : i-j+1..i$, 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 $D \| B$, consider deriving the failures of C by the approach outlined in Section 2.2. 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 [19].

7.1. Deriving the simulation rules

We now derive simulation rules between A and $(D \parallel B)$ in terms of the component classes. 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 operations which are shared (i.e., occur in both D and B) 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.2** and **DS.3** which, for the communicating operations, require that:

DS.2′ $\forall\, AState;\ DState;\ BState \bullet Abs \implies (\text{pre}\ AOp \iff \text{pre}\ (DOp \parallel_! BOp))$
DS.3′ $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet$
$Abs \wedge (DOp \parallel_! BOp) \implies (\exists\, AState' \bullet AOp \wedge Abs')$

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).

DOp
$\Delta(x)$
$x, x' : X$
$z? : Z$
---
p

BOp
$\Delta(y)$
$y, y' : Y$
$z! : Z$
---
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

DOp $\|_!$ *BOp*

$\Delta(x, y)$
$x, x' : X$
$y, y' : Y$
$z! : Z$

$p[z!/z?] \land q$

where $p[z!/z?]$ is the predicate p with all free occurrences of $z?$ renamed to $z!$.

Therefore we can simplify the precondition calculation as follows:

$$\begin{aligned}
\text{pre}(DOp \|_! BOp) &\equiv \exists x' : X, y' : Y, z! : Z \bullet p[z!/z?] \land q \\
&\equiv \exists x' : X, y' : Y, z! : Z, w! : Z \bullet p[w!/z?] \land q \\
&\quad \text{[by restriction 3]} \\
&\equiv (\exists x' : X, w! : Z \bullet p[w!/z?]) \land (\exists y' : Y;\ z! : Z \bullet q) \\
&\quad \text{[since } p[w!/z?] \text{ and } q \text{ refer to distinct variables]} \\
&\equiv \text{pre}\, DOp[w!/z?] \land \text{pre}\, BOp
\end{aligned}$$

In addition, we have

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

Extrapolating to the general case, we have the following.

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

The requirements for a downward simulation can now be re-expressed as requirements between the components instead of the equivalent classes, leading to the following definition.

Definition 6. Parallel downward simulation
Let D and B be Object-Z classes containing operations DOp and BOp respectively. Suppose, further, that DOp has inputs $z_1?, \ldots z_n?$ and outputs $z_{n+1}!, \ldots z_{n+m}!$ and BOp has outputs $z_1!, \ldots z_n!$ and inputs $z_{n+1}?, \ldots z_{n+m}?$. Then a CSP expression $D \parallel B$ is a downward simulation of the Object-Z class A if D and B satisfy the restrictions above and the following hold.

PS.1 $\forall\, DInit \land BInit \bullet (\exists\, AInit \bullet Abs)$

PS.2 $\forall\, AState;\ DState;\ BState \bullet$

$$\begin{aligned}
Abs \implies (&\text{pre}\, AOp \\
&\iff \\
&\text{pre}\, DOp[w_1!/z_1?, \ldots, w_n!/z_n?] \land \text{pre}\, BOp[w_{n+1}!/z_{n+1}?, \ldots, w_{n+m}!/z_{n+m}?])
\end{aligned}$$

PS.3 $\forall\, AState;\ DState;\ BState;\ DState';\ BState' \bullet$

$$\begin{aligned}
&Abs \land DOp[z_1!/z_1?, \ldots, z_n!/z_n?] \land BOp[z_{n+1}!/z_{n+1}?, \ldots, z_{n+m}!/z_{n+m}?] \\
&\quad \implies (\exists\, AState' \bullet AOp \land Abs')
\end{aligned}$$

Since hiding is monotonic with respect to refinement in CSP, to show that $CashService_0$ will be refined by $(Bank \| CashPoint) \setminus \{Transfer\}$, it suffices to show that $CashService_1$ is refined by $(Bank \| CashPoint)$. To show the refinement of $CashService_1$ to $(Bank \| CashPoint)$, we use the parallel downward simulation rules from Definition 6.

The retrieve relation Abs is straightforward: equating variables with the same name modulo subscript 1, e.g., *card* and $card_1$. To begin, we must show that the decomposition satisfies the restrictions outlined above.

Restriction 1 clearly holds. For restriction 2 we have to compare parameters in *OpenAcct*, *Transfer* and *Passwd* between the two classes *Bank* and *CashPoint*. In each case, the basenames in the pairs of operations are the same (e.g., *pin* in *Passwd*). In addition, we have to check the restrictions on the predicates given by restriction 3. For example, for the *OpenAcct* operation we have to check that in the environment given by the signature of *OpenAcct*:

$$
\begin{array}{l}
pins' = pins \cup \{(account!, pin?)\} \land \\
account! = (max\,\mathrm{dom}\,accts) + 1 \land \\
accts' = accts \cup \{(account!, 0)\} \text{ iff} \\
(\exists\, y! \bullet \\
\quad pins' = pins \cup \{(y!, pin?)\} \land \\
\quad account! = (max\,\mathrm{dom}\,accts) + 1 \land \\
\quad accts' = accts \cup \{(account!, 0)\})
\end{array}
$$

Since no constraints are being placed on the inputs *account*? and *pin*? (*account*? has been renamed to an output on the left-hand side above) this is trivially satisfied.

Simulation rule PS.1 This amounts to showing that together the initialisations of *Bank* and *CashPoint* imply the initialisation in $CashService_1$, which is trivial since, modulo the retrieve relation, they are equal.

Simulation rule PS.2 For each operation in $CashService_1$ we have to show that either it is a standard refinement if it occurs in just one class, or show that **PS.2** holds if it occurs in both classes. For the former, *Request*, *Insert*, *Eject* and *Withdraw* are identical in the refinement, and both operations appear in just one class. For the remaining operations (*Passwd*, *OpenAcct* and *Transfer*) we have to demonstrate that **PS.2** holds and this is relatively straightforward.

Simulation rule PS.3 In a similar fashion we must show **PS.3** holds. This again is straightforward.

Then given the refinement of $CashService_1$ to $(Bank \| CashPoint)$ we can conclude that $CashService_1 \setminus \{Transfer\}$ is refined by $(Bank \| CashPoint) \setminus \{Transfer\}$, and thus that the original specification $CashService_0$ is also refined by $(Bank \| CashPoint) \setminus \{Transfer\}$.

A comment is in order here about the strategy that we have adopted to verify this refinement. In order to verify the refinement we had to introduce hidden events before we decomposed the component $CashService_0$ into its two component classes. To do so we need to know not just the names of the operations to be hidden, but also their parameters, since in CSP events are hidden (i.e., in terms of Object-Z we hide both the operation name and also the inputs and outputs). For this reason appropriate outputs in *Transfer* in $CashService_1$ were introduced and these were chosen on the basis of the decomposition we were aiming for.

8. Further simulation rules

In addition to introducing a parallel composition, an existing parallel composition can be refined in a component-wise fashion. We already know that if one component, B_1 say, is refined to C_1, then the composition $B_1 \| B_2$ is refined by $C_1 \| B_2$. However, it is also possible that one specification $B_1 \| B_2$ is refined by another, $C_1 \| C_2$, without either pair of components being related by refinement. This can happen when the behaviour of an operation is moved between the components in a parallel composition. The effect of this is that, within a single component, preconditions and postconditions can seemingly be weakened, or indeed ultimately operations can be removed or added. All of this is permissible if the overall synchronisation of the new components delivers a refinement of the original ones.

As in Definition 6, we can produce a set of simulation rules to verify refinements of this sort. Their derivation is similar to those in Section 7.1, and we do not give the derivation in detail since the essence is much the same.

Suppose that $B_1 \| B_2$ is refined by $C_1 \| C_2$, and that both sets of components satisfy the restrictions outlined at the start of Section 7.1. As before, for operations names occurring in only one component class, the simulation rules are unchanged.

So consider an operation appearing in both components B_1 and B_2, which is refined to an operation also appearing in both components C_1 and C_2. Suppose further that, similarly to the operations *DOp* and *BOp* on page 22, the operation $B_1\,Op$ has declaration $z? : Z$ and operation $B_2\,Op$ has declaration $z! : Z$, and that these operations are refined to $C_1\,Op$ and $C_2\,Op$ respectively.

Then, we can derive simulation rules between two equivalent constructed classes and re-write these simulation rules in terms of the original components as follows.

Definition 7. Parallel downward simulation for components
A CSP expression $C_1 \| C_2$ is a downward simulation of the CSP expression $B_1 \| B_2$ if components satisfy the restrictions above and the following hold.

$$\textbf{CPS.1}\ \forall C_1 Init \wedge C_2 Init \bullet (\exists B_1 Init \wedge B_2 Init \bullet Abs)$$
$$\textbf{CPS.2}\ \forall B_1 State;\ B_2 State;\ C_1 State;\ C_2 State \bullet$$
$$Abs \implies (\text{pre}\, B_1 Op[w!/z?] \wedge \text{pre}\, B_2 Op$$
$$\iff$$
$$\text{pre}\, C_1 Op[w!/z?] \wedge \text{pre}\, C_2 Op)$$
$$\textbf{CPS.3}\ \forall B_1 State;\ B_2 State;\ C_1 State;\ C_2 State;\ C_1 State';\ C_2 State' \bullet$$
$$Abs \wedge C_1 Op[z!/z?] \wedge C_2 Op$$
$$\implies (\exists B_1 State' \wedge B_2 State' \bullet B_1 Op[z!/z?] \wedge B_2 Op \wedge Abs')$$

The extrapolation to the general case can be made in a similar fashion to before.

A structural refinement does not necessarily have to introduce CSP operators into a specification, but might also reasonably remove CSP operators. For example, one might want to verify that $(Bank \| CashPoint) \setminus \{Transfer\}$ is refined by a further specification given, for example, as $(Bank_2 \| CashPoint_2)$ or as $CashService_3 \setminus \{Transfer\}$.

To verify such refinements, simulation rules can be given which are, in many cases, the simple reverse of the rules presented above. We give just two examples here. One example is when we wish to remove hiding, which can be done via application of the following.

Definition 8. Weak downward simulation (removal)
The CSP expression A is a weak downward simulation of the CSP expression $B \setminus L$ if there is a retrieve relation Abs such that the following hold for every operation not in L.

$$\textbf{WS(R).1}\ \forall BState \bullet BINIT \Rightarrow \exists AState \bullet AINIT \fatsemi Int_L \wedge Abs$$
$$\textbf{WS(R).2}\ \forall AState;\ BState \bullet Abs \Rightarrow (\text{pre}(Int_L \fatsemi AOp) \iff \text{pre}\, BOp)$$
$$\textbf{WS(R).3}\ \forall AState;\ BState;\ BState' \bullet$$
$$Abs \wedge BOp \implies \exists AState' \bullet Abs' \wedge (Int_L \fatsemi AOp \fatsemi Int_L)$$

where Int_L represents the effect of the hidden events in L as defined in Section 6.

Note that since there is no hiding in B, rules **D1** and **D2** are not necessary.

In a similar fashion we can remove a parallel composition by applying the following.

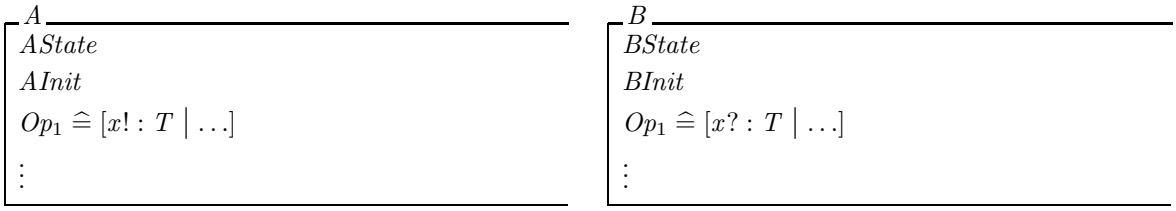
Definition 9. Parallel downward simulation (removal)
Let D and B be Object-Z classes containing operations DOp and BOp respectively. Suppose, further, that DOp has inputs $z_1?, \ldots z_n?$ and outputs $z_{n+1}!, \ldots z_{n+m}!$ and BOp has outputs $z_1!, \ldots z_n!$ and inputs $z_{n+1}?, \ldots z_{n+m}?$. Then a CSP expression A is a downward simulation of $D \parallel B$ if D and B satisfy the restrictions above and the following hold.

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

The simulation rules that we have introduced enable one to verify a range of structural refinements in a manner illustrated by the examples. With a full set of rules for removing operators, our approach is also complete in that all structural refinements can be verified by their application. For example, one might refine a specification $A_1 \| B_1$ by introducing another operation (subsequently hidden) to perform part of the communication, resulting in a specification of the form $(A_2 \| B_2) \setminus L$. Such a refinement can be verified by removing the parallel operator from the original specification and then refining the resulting "flattened" specification to the structure of the second specification. In cases like these, it clearly would be possible to derive appropriate structural simulation rules allowing us to by-pass the need to flatten the specification, but they become ever more complex according to how specialised the situation is.

One can, however, develop structural refinements to implement particular commonly occurring situations.

For example, suppose A and B synchronise on an operation Op_1 in order to communicate a data value, say, as in the following (where the elided operations in the classes A and B might differ):



A common implementation of this might be to use a one place buffer given as a class *Buf* with operations to *Push* and *Pop*. This can be achieved by the following composition $((A[Push/Op_1] \;|||\; B) \parallel Buf[Op_1/Pop]) \setminus \{Push\}$ where A and B now communicate directly only with *Buf* which implements the channel between them. Indeed, this composition has behaviour identical to that in the original description $A \| B$, and this can easily be checked using applications of the appropriate structural refinement rules. Once verified this design pattern can be re-used as and when required.

In situations that are more complex than the cases discussed above it is not clear as to the merits of such rules over and above calculation of the two equivalent classes, followed by a standard simulation verification between them. Therefore, each case must be judged on its relative merits based upon complexity of verification, and the rules presented above merely serve to simplify some commonly occurring cases.

9. Conclusions

This paper has presented an approach to verifying refinements for specifications written in a combination of CSP and Object-Z, with special emphasis on structural refinements where CSP operators can be introduced or removed.

Since neither CSP or Object-Z have been modified in the integration, following the approach of Josephs [21], standard state-based simulation techniques can be used to verify the refinements. The use of simulations between single Object-Z components is straightforward, and the purpose of this paper was to extend these simulation techniques to situations where, for example, parallel composition or hiding might be introduced as part of the refinement step.

The simulation rules we have developed are sound, since we have verified them with respect to failures-divergences refinement, but they are not complete. There is a balance to be struck between ever more complex simulation rules that can be used to verify arbitrary refinements versus placing restrictions on the components involved in order that refinements can more easily be verified.

Further work is therefore needed to determine whether these rules are the ones that prove useful in practice, and whether they really do reduce the effort required to verify a refinement. Clearly in some refinements the interaction between the components is so great that in practice you end up calculating the single equivalent class when verifying a refinement. Whether these are the majority of cases remains to be seen.

As indicated in Section 1, work on refinement for state-based concurrent systems goes back to the late '80s, with papers by Josephs [21], He [20] and Woodcock and Morgan [37] being relevant. More recent work in the same vein includes that by Bolton et al [4, 2] and that by Boiten and Derrick [9].

There has been recent renewed interest in applications of this work due to interest in combining state-based and concurrent specification languages. Approaches to combining Z or Object-Z with CSP or CCS include the work of Fischer [11], Galloway [16], Mahony and Dong [23], Sühl [33] and Taguchi and Araki [34]. A survey of some of these approaches is given in [12]. Applications of this work include use of these combinations for the specification of interactive systems [22] and embedded systems [24].

The work with most similarities to that presented here is that of Fischer. He also combines Object-Z with CSP by using a failure-divergence semantics as the basis for the integration. However, the integration defined is more complex since he extends the basis of integration by adding channel definitions and CSP processes to Object-Z classes. The simple use of Object-Z as a means to describe components is therefore lost. In addition, both the precondition and guard of an operation are defined and events can either be atomic or have duration (and therefore have a start and end). A more complex integration into the semantics therefore has to be used.

Fischer does, however, discuss refinement for his language. In [13] Fischer derives downward and upward simulations for use on the components in a CSP-OZ specification, but in [13] he is mainly concerned with the basic definition of simulation in the integrated language as opposed to transformations which change the structure of the specification. Changes of structure are consider by Fischer and Wehrheim in [14]. They develop model checking techniques based upon the CSP model checking tool FDR [15] to verify simulations where the structure of the specification changes, and their work complements our approach.

Further work could also be undertaken to determine whether the simulation techniques presented in this paper can usefully be combined with model checking approaches such as those discussed by Fischer and Wehrheim. Our work could also be extended to other combinations of Object-Z and CSP, such as those described in [13, 23], and also to integrations involving other state-based languages such as B [35].

References

[1] K. Araki, A. Galloway, and K. Taguchi, editors. *International conference on Integrated Formal Methods 1999 (IFM'99)*. Springer, York, July 1999.

[2] C. Bolton and J. Davies. A Singleton Failures Semantics for Communicating Sequential Processes. Submitted for publication, 2002.

[3] C. Bolton and J. Davies. Refinement in Object-Z and CSP. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, 2002.

[4] C. Bolton, J. Davies, and J. C. P. Woodcock. On the refinement and simulation of data types and processes. In Araki et al. [1], pages 273–292.

[5] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In *Pittsburgh Symposium on Concurrency*, volume 197 of *Lecture Notes in Computer Science*, pages 281–305. Springer-Verlag, 1985.

[6] J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z*. Springer-Verlag, 2001.

[7] 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, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 369–388. Springer-Verlag, April 1997.

[8] J. Derrick, E. A. Boiten, H. Bowman, and M. W. A. Steen. Specifying and Refining Internal Operations in Z. *Formal Aspects of Computing*, 10:125–159, December 1998.

[9] J. Derrick and E.A. Boiten. Unifying concurrent and relational refinement. In *REFINE - a FLoC'02 FME workshop*, Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[10] J. Derrick and G. Smith. Structural refinement in Object-Z/CSP. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *International conference on Integrated Formal Methods 2000 (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 194–213. Springer, November 2000.

[11] C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Second IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 423–438. Chapman & Hall, July 1997.

[12] C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23. Springer-Verlag, September 1998.

[13] C. Fischer. *Combination and implementation of processes and data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[14] C. Fischer and H. Wehrheim. Model checking CSP-OZ specifications with FDR. In Araki et al. [1], pages 315–334.

[15] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR 2*, October 1997. FDR2 User Manual.

[16] A. Galloway and W. Stoddart. An operational semantics for ZCCS. In Hinchey and Liu [18], pages 272–282.

[17] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.

[18] M. G. Hinchey and Shaoying Liu, editors. *First International Conference on Formal Engineering Methods (ICFEM'97)*, Hiroshima, Japan, November 1997. IEEE Computer Society Press.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[20] He Jifeng. Process refinement. In J. McDermid, editor, *The Theory and Practice of Refinement*. Butterworths, 1989.

[21] M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

[22] I. MacColl. Specifying interactive systems in Object-Z and CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods (IFM'99)*, pages 335–352. Springer-Verlag, 1999.

[23] 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, editors, *20th International Conference on Software Engineering (ICSE'98)*. IEEE Press, 1998.

[24] B.P. Mahony and J.S. Dong. Sensors and actuators in TCOZ. In J.M.Wing, J.C.P. Woodcock, and J. Davies, editors, *World Congress on Formal Methods (FM'99)*, pages 1166–1185. Springer-Verlag, 1999.

[25] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[26] A. W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice-Hall, 1998.

[27] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
[28] 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, *FME'97: Industrial Application and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, September 1997.
[29] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, 2000.
[30] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In Hinchey and Liu [18], pages 293–302.
[31] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.
[32] J. M. Spivey. *The Z notation: A reference manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.
[33] C. Sühl. RT-Z: An integration of Z and timed CSP. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods (IFM'99)*, pages 29–48. Springer-Verlag, 1999.
[34] K. Taguchi and K. Araki. The state-based CCS semantics for concurrent Z specification. In Hinchey and Liu [18], pages 283–292.
[35] H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods 1999 (IFM'99)*, pages 437–456, York, July 1999. Springer.
[36] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.
[37] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!- Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.