Abstract specification in Object-Z and CSP

Graeme Smith[1] and John Derrick[2]

[1]Software Verification Research Centre, University of Queensland 4072, Australia
phone: +61 7 3365 1625 fax: +61 7 3365 1533 smith@svrc.uq.edu.au
[2]Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
J.Derrick@ukc.ac.uk

Abstract. A number of integrations of the state-based specification language Object-Z and the process algebra CSP have been proposed in recent years. In developing such integrations, a number of semantic decisions have to be made. In particular, what happens when an operation's precondition is not satisfied? Is the operation *blocked*, i.e., prevented from occurring, or can it occur with an undefined result? Also, are outputs from operations *angelic*, satisfying the environment's constraints on them, or are they *demonic* and not influenced by the environment at all? In this paper we discuss the differences between the models, and show that by adopting a blocking model of preconditions together with an angelic model of outputs one can specify systems at higher levels of abstraction.

1 Introduction

One strand of recent work on integrating formal methods has been the area concerned with combining state-based languages such as Object-Z [11, 4] and Z [14] with process algebras such as CSP [6, 9] and CCS [8]. A canonical example of this are the integrations of Object-Z with CSP. In such an integration, of which there have been a number of proposals [10, 5, 7, 13], Object-Z is used to specify the components of a system, and CSP is used to describe how the components interact and communicate.

In developing such an integration, one is faced with a choice in terms of how *preconditions* and *outputs* are dealt with, and this choice effects the design approach one takes in the language.

The specific choice in terms of preconditions is whether to adopt a *blocking* or *non-blocking* model of an operation. That is, outside the stated or calculated precondition, is the operation unable to occur (blocking) or is it able to occur but its effect undefined (non-blocking)?

Z adopts the non-blocking approach and Object-Z the blocking approach (which is closer to an object-oriented and process algebraic interpretation). Different integrations of Object-Z and CSP have also taken differing interpretations. For example, [10, 13] adopt a blocking model whereas [5, 7] adopt a non-blocking model[1].

[1] Both non-blocking approaches [5, 7] also extend the syntax of Object-Z to include *guards* so that operations can also be blocked.

The specific choice in terms of outputs is whether to adopt a *demonic* or *angelic* model of outputs. To understand this, consider a specification $A \| B$ where the components synchronise on an operation Op. In A, this operation has a non-deterministic output, matched by an input in the corresponding operation in B. The question is "Should B be able to control the output of A?".

In a demonic model of outputs the answer is "No". That is, the non-determinism of the value output by Op in A is entirely internal to A. So, if the chosen output is incompatible with the expected input in B, the composite operation's precondition is not satisfied (resulting in blocking or undefined behaviour). The contrasting angelic model of outputs, however, allows B to affect the non-determinism in A by choosing a value to synchronise on, if one can be found, thus allowing the operation to proceed normally if at all possible.

The majority of work on integrating Object-Z and CSP has used the demonic model of outputs. This work includes that of [5, 7, 13]. This is also the main approach appearing in other work on semantics of value-passing communication, e.g., Butler [1] places conditions on composed value-passing action systems which ensure that outputs are always accepted by the environment. The main reason for this bias is that since outputs can be controlled by the environment in the angelic model, they are effectively the same as inputs and hence cannot be strengthened during refinement. As a consequence, standard state-based refinement [2], which allows such strengthening, is not compositional.

In this paper, we show, however, that adopting an angelic model of outputs, together with a blocking model of operations, as is done in [10], allows us to specify at a higher level of abstraction. Consequently, specifications are simpler and therefore easier to understand and reason about. Furthermore, we show how the limitations of refinement for this interpretation of outputs can be overcome.

The structure of the paper is as follows. In Section 2, we illustrate the different interpretations of preconditions and outputs via a simple example. In Section 3 we show how, by adopting a blocking plus angelic model of operations, we are able to specify systems more abstractly. The consequences for refinement are dealt with in Section 4. Finally, we conclude in Section 5.

2 Illustrative example

The differences between the approaches to the modelling of preconditions and outputs is best illustrated by considering a simple example. We assume the reader is familiar with both CSP and Object-Z, and the setting of our work is the Object-Z/CSP approach first proposed by Smith [10] and further developed by Smith and Derrick [13].

In this approach, which is similar in essence to other work in the area [5], Object-Z is used to describe the individual components which are combined with CSP operators by using, for example, parallel composition or interleaving.

Thus, in the following trivial example (more realistic examples are contained in [10, 12, 13, 3]), two components A and B are given as Object-Z classes:

A

Op

$x! : \mathbb{N}$

$x! \in \{1, 2\}$

B

$y : \mathbb{Z}$

Op

$\Delta(y)$

$x? : \mathbb{N}$

$x? = 1$

$y \geq 0 \land y' = y + x?$

To illustrate the difference between the blocking and non-blocking model, let us first consider the component B on its own. The blocking model takes the view that outside its precondition ($y \geq 0 \land x? = 1$) the operation is not enabled and cannot occur. On the other hand, the non-blocking model views the operation to be enabled but the outcome is undefined, i.e., any after-state might result, including divergence or non-termination.

The complete Object-Z/CSP specification is given by the parallel composition of the two components:

$$Spec = A \| B$$

The effect of this specification is a system with one operation *Op*. Adopting the blocking model (as is done in Object-Z/CSP), how does it behave? The operations in A and B must synchronise, with agreement on the value communicated. Therefore, if this event does occur the value 1 will be communicated from A to B.

But does this event occur? Here, in addition to the issue of the precondition, we have a choice, which is the crux of the issue concerning outputs highlighted in the introduction. In particular, if A can non-deterministically output 2, then under these circumstances the precondition of *Op* will not hold. This interpretation is the one taken in [13] and [3] where the operation would be blocked. We shall call such an interpretation demonic.

The alternative is to view A and B as cooperating components, and allow the event to proceed *if it can*. That is, the non-determinism in A is restricted by what the environment is prepared to accept. This interpretation is less popular, but is the one taken in [10] and [12]. We shall call such an interpretation angelic.

3 Abstraction

The non-blocking interpretation of operations was developed (in Z, for example) predominantly for the specification of sequential systems. Such systems perform operations in a particular order. In concurrent systems where operations can occur at any time, specifiers using a non-blocking approach need to be more explicit about the exceptional behaviour that occurs when a precondition is not met. In many cases, this information is missing in a specification, leading to unintentional underspecification. For example, consider the following specification of a network node.

$$[\mathit{Message}, \mathit{Address}]$$

$$\mathit{to}, \mathit{from} : \mathit{Message} \rightarrow \mathit{Address}$$

Node

$$\mathit{address} : \mathit{Address}$$

$$\mathit{received} : \mathrm{seq}\, \mathit{Message}$$

$\textit{I}\textsc{nit}$

$$\mathit{received} = \langle\, \rangle$$

$\mathit{Receive}$

$$\begin{array}{l} \Delta(\mathit{received}) \\ \mathit{msg}? : \mathit{Message} \\ \hline \mathit{to}(\mathit{msg}?) = \mathit{address} \\ \mathit{received}' = \mathit{received} \frown \langle \mathit{msg}? \rangle \end{array}$$

Send

$$\begin{array}{l} \mathit{msg}! : \mathit{Message} \\ \hline \mathit{from}(\mathit{msg}!) = \mathit{address} \end{array}$$

The receive operation has a precondition that the incoming message is addressed to the node. In the case where the incoming message is not addressed to the node, the precondition is not met. Generally, we would not want our network to behave chaotically whenever a node detects a message which is not for it. A more suitable behaviour would be to ignore the message. Under a non-blocking interpretation, this would need to be specified explicitly as follows.

$\mathit{Receive}$

$$\begin{array}{l} \Delta(\mathit{received}) \\ \mathit{msg}? : \mathit{Message} \\ \hline \mathit{to}(\mathit{msg}?) = \mathit{address} \land \mathit{received}' = \mathit{received} \frown \langle \mathit{msg}? \rangle \\ \lor \\ \mathit{to}(\mathit{msg}?) \neq \mathit{address} \land \mathit{received}' = \mathit{received} \end{array}$$

The blocking model, on the other hand, provides a way of specifying exceptional behaviour implicitly. This is only true when the exceptional behaviour is of the form that nothing happens. Other types of exceptional behaviour have to be specified explicitly as in the non-blocking model. As the node example illustrates, however, doing nothing is often what is required in concurrent systems.

This feature of the blocking model proves useful in the Object-Z and CSP context where it can be used to abstract away from implementation detail. For example, given a set of addresses $A : \mathbb{P}\,Address$, we can define a network as follows[2]. (The renaming of *Send* events ensures they communicate with *Receive* events with the same message.)

$$Network = \|_{a:A}\ Node_{\{address \mapsto a\}}[Receive/Send]$$

Adopting the blocking model for classes, the nodes only engage in events which concern them (i.e., where they are the sender or receiver). The specification abstractly models that such messages are not seen by other nodes (due to some routing mechanism in the network) or are simply ignored by them. The actual mechanism by which the network operates is not of interest at this level of abstraction.

It is not possible to similarly model at this level of abstraction with the non-blocking model since all events are in a given node process's alphabet regardless of the sender and recipient. Therefore, the network's operating mechanism needs to be explicitly specified as, for example, in the modified *Receive* schema above.

This ability to abstract from implementation details is further enhanced when we adopt the angelic model of outputs. This allows us to easily model cooperation between processes to produce an output. This allows us to abstract away from the actual cooperation mechanism which in general would require additional message passing. For example, a group of nodes can elect a "leader" by synchronising on an *ElectLeader* operation of the form

```
┌─ ElectLeader ──────────────────────
│ address! : Address
└────────────────────────────────────
```

which would be included in each *Node*. The leader is chosen by the identification of the *address*! outputs.

In the demonic model where outputs are chosen without reference to the environment, it cannot be guaranteed that all processes choose the same output. Hence, if the *ElectLeader* operation is required to occur, deadlock is possible. To avoid this, the actual protocol to elect the leader, which would typically comprise a series of communications, needs to be specified.

In general, the non-blocking interpretation of operations and demonic model of outputs reflect more closely the situation in an implementation: exceptional behaviour must be dealt with and outputs cannot be influenced by the environment. The penalty for this is that specifications tend to be less abstract, including additional details which may not be necessary to describe the essential functionality of the specified system. Introducing these details at an early

[2] $Node_{\{address \mapsto a\}}$ denotes the process corresponding to the class *Node* when its constant *address* is instantiated to a [10].

stage can complicate both the understandability and the ease of analysis of a specification.

To further illustrate these ideas, we specify a simple hotel booking system. Customers of this system may book a room of a particular type for a particular date. They may also cancel a booking they have previously made.

We introduce a given type for dates

$$[Date]$$

and define the type of a room to be single, double or twin.

$$RoomType ::= single \mid double \mid twin$$

A customer is specified as having a name and a set of bookings for particular types of rooms on particular dates. Initially, this set is empty and operations to make and cancel bookings are provided.

Customer

$$name : Name$$

$$bookings : RoomType \leftrightarrow Date$$

Init

$$bookings = \varnothing$$

Book

$$\begin{array}{l} \Delta(bookings) \\ n! : Name \\ r! : RoomType \\ d! : Date \\ \hline n! = name \\ bookings' = bookings \cup \{(r!, d!)\} \end{array}$$

Cancel

$$\begin{array}{l} \Delta(bookings) \\ n! : Name \\ r! : RoomType \\ d! : Date \\ \hline n! = name \\ (r!, d!) \in bookings \\ bookings' = bookings \setminus \{(r!, d!)\} \end{array}$$

Assuming we are adopting the blocking model of operations and angelic model of outputs, the hotel booking system with which such customers interact can be specified as follows.

Hotel

$rooms : \mathrm{seq}\, RoomType$
$booked : Date \rightarrow (\mathbb{N} \leftrightarrow Name)$

$\mathrm{dom}(\mathrm{ran}\, booked) \subseteq \mathrm{dom}\, rooms$

Init

$\forall\, d : Date \bullet booked(d) = \varnothing$

Book

$\Delta(booked)$
$n? : Name$
$r? : RoomType$
$d? : Date$

$$\begin{array}{l} \exists\, i : \mathrm{dom}\, rooms \bullet \\ \quad rooms(i) = r? \wedge \\ \quad i \notin \mathrm{dom}\, booked(d?) \wedge \\ \quad booked' = booked \oplus \{d? \mapsto booked(d?) \cup \{(i, n?)\}\} \end{array}$$

Cancel

$\Delta(booked)$
$n? : Name$
$r? : RoomType$
$d? : Date$

$$\begin{array}{l} \exists\, i : \mathrm{dom}\, rooms \bullet \\ \quad rooms(i) = r? \wedge \\ \quad (i, n?) \in booked(d?) \wedge \\ \quad booked' = booked \oplus \{d? \mapsto booked(d?) \setminus \{(i, n?)\}\} \end{array}$$

This class models the hotel's rooms by a sequence of room types (*rooms*), and the current bookings by a function from dates to pairs of room numbers, denoted by a room's position in *rooms*, and customer names (*booked*). Initially, no rooms are booked on any date. Operations *Book* and *Cancel* allow bookings to be made and cancelled respectively.

The system can then be specified as follows.

$$System = (|||_{n:Name}\ Customer_{\{name \mapsto n\}}) \mathbin{||} Hotel$$

The parallel composition ensures any customer performing a book or cancel event synchronises with the hotel performing the same event.

Under the blocking model of operations and the angelic model of outputs, the synchronisation corresponding to a booking being made is abstractly modelling a sequence of communications. One possible implementation is that the customer provides a preference for room type and date, and the hotel either acknowledges the booking or indicates that the booking cannot be made and asks for a second preference. In the latter case, the hotel may provide information about why the booking was unsuccessful, e.g., no room of the desired type available on the date, and may make suggestions for alternatives, e.g. booking a twin rather than a double room. Another possible implementation is that the customer simply indicates that he or she is interested in making a booking and is presented with a table showing all available rooms and dates from which to make a choice. Using the blocking model and angelic outputs, the actual implementation need not be specified.

This is not the case, however, if either the non-blocking model of operations or demonic model of outputs is used. Both of these require additional information to be passed between a customer and the hotel before a booking can be made.

Adopting the non-blocking model, the hotel would accept any inputs for a booking (since preconditions do not need to be met) and hence bookings will proceed even when requested room type/date combinations are unavailable. An exceptional behaviour modelling the hotel indicating that a request cannot be met would need to be specified in order to capture the correct behaviour in this case.

With demonic outputs, additional information would need to be communicated to the customer (using one of the implementation strategies above, for example) before a successful booking could be made. Otherwise, there is the possibility of customer processes deadlocking when their booking choice is unavailable. Once again, this is not the correct behaviour.

4 Refinement

In this section, we examine the issue of refining specifications under the blocking plus angelic model of operations.

At first sight, this seems to provide a theory of refinement which is *not* compositional. Because Object-Z/CSP specifications are given a semantics identical to that of CSP specifications (i.e., a failures-divergences semantics) [10], we use the notion of refinement adopted for CSP (i.e., failures-divergences inclusion) [6]. Consider classes A and B from Section 2. Under the angelic model of output non-determinism the operation in $A \| B$ is always enabled since A and B cooperate on the choice of value communicated. Now consider refining A to the following class C:

```
┌─ C ──────────────────────────────
│ ┌─ Op ───────────────────────────
│ │ x! : ℕ
│ ├──────────
│ │ x! = 2
│ └────────────────────────────────
└──────────────────────────────────
```

where, as in standard approaches to state-based refinement [2], we have reduced the non-determinism in the output.

Now, when we form the composition $C \| B$ we find the operation Op is blocked. We have thus introduced a *deadlock* into the specification, and therefore $C \| B$ is *not* a refinement of $A \| B$. We might conclude, therefore, that refinement is not compositional. However, closer inspection reveals a sleight of hand at play. The unstated assumption in this deduction was that the standard Object-Z downward simulation rule [2] (see Definition 1) is sound with respect to CSP refinement.

Definition 1 *Downward simulation in Object-Z*
An Object-Z class $C = (C.\textsc{State}, C.\textsc{Init}, COp_i)_{i \in I}$ *is a downward simulation of the class* $A = (A.\textsc{State}, A.\textsc{Init}, AOp_i)_{i \in I}$ *if there is a retrieve relation* R *on* $A.\textsc{State} \land C.\textsc{State}$ *such that the following hold for all* $i \in I$.

DS.1 $\forall\, C.\textsc{Init} \bullet \exists\, A.\textsc{Init} \bullet R$
DS.2 $\forall\, A.\textsc{State};\ C.\textsc{State} \bullet R \implies (\text{pre}\, AOp_i \iff \text{pre}\, COp_i)$
DS.3 $\forall\, A.\textsc{State};\ C.\textsc{State};\ C.\textsc{State}' \bullet$
$R \land COp_i \implies (\exists\, A.\textsc{State}' \bullet R' \land AOp_i)$

However, this rule is sound with respect to the blocking plus *demonic* model, but it is not sound with respect to the blocking plus *angelic* model. Confirmation of this is found by noting that initially C cannot perform operation Op with $x! = 1$ whereas A can. Refinement in CSP requires that the events which a process can refuse to perform at any stage of its evolution are a subset of those of any process it refines [6]. Thus, C is not, in fact, a valid refinement of A with respect to the angelic model of outputs.

The solution here is to adapt the simulation rules given above to produce rules which *are* sound with respect to the blocking plus angelic model. In fact, this adaption is easy. As detailed in [12], one has just to change the meaning of pre Op to include existential quantification of the after state only (and not the output), since we wish to exclude reduction of non-determinism of the output.

Thus, if we define $\text{Pre} \mathrel{\widehat{=}} \exists\, State' \bullet Op$ for an operation Op defined over state space $State$, and use Pre in place of pre in the definition above, we produce a set of simulation rules sound with respect to the blocking plus angelic model. We call these rules (ba)-simulation rules for sake of easy reference.

Refinements in this model behave exactly as before, except non-determinism in outputs cannot be resolved. In fact, this is a by-product of how outputs are being used in this specification style. Specifically, the non-deterministic selection of outputs represents some sort of *required non-determinism* in the description (like external choice in CSP). Therefore, it would be an unacceptable refinement to reduce this non-determinism which was explicitly needed as part of the modelling paradigm.

We still do, however, achieve a compositional theory of refinement. That is, using the (ba)-simulation rules (where Pre has replaced pre), if C refines A then $C \| B$ refines $A \| B$.

4.1 Introducing an explicit communication mechanism

Part of the motivation for using the blocking model with an angelic model of outputs was to model, at a suitable level of abstraction, cooperative communication between components. This it does successfully, allowing components to agree on values without having to describe explicitly how this agreement is achieved.

However, in an implementation the actual agreement mechanism used will need to be made explicit, and the natural question to ask, therefore, is whether the abstract description of the communicating components can be refined to an implementation-oriented view. In fact, we can perform this refinement, and we illustrate now how it can be achieved.

Consider a specification $Sys_1 = A_1 \| B_1$, where the cooperating communicating part of the components are as follows.

A_1

$\quad$ *Choose*

$\quad x! : \mathbb{N}$

$\quad x! \in \{1, 2\}$

B_1

$\quad$ *Choose*

$\quad x! : \mathbb{N}$

$\quad x! \in \{2, 3\}$

With the blocking model plus angelic model of outputs, the synchronisation of *Choose* in $A_1 \| B_1$ models agreement on a particular value for communication. We wish to implement this design with an explicit mechanism which models finding the agreed value. In particular, we will refine Sys_1 to Sys_2, where

$$Sys_2 = (C \ {}_X\|_Y (A_2 ||| B_2)) \setminus \{\!| \ Send \ |\!\}$$

where $X = \{\!| \ Send, Choose \ |\!\}$, $Y = \{\!| \ Send \ |\!\}$ and the components are given as:

A_2

$\quad$ *Send*

$\quad v! : \{1\}$

$\quad y! : \mathbb{P}\,\mathbb{N}$

$\quad y! = \{1, 2\}$

B_2

$\quad$ *Send*

$\quad v! : \{2\}$

$\quad y! : \mathbb{P}\,\mathbb{N}$

$\quad y! = \{2, 3\}$

C

$nodes : \mathbb{P}\{1,2\}$
$s : \mathbb{P}\,\mathbb{N}$

$\text{I}\textsc{nit}$
$nodes = \{1,2\} \wedge s = \mathbb{N}$

$Send$
$\Delta(nodes, s)$
$v? : \mathbb{N}$
$y? : \mathbb{P}\,\mathbb{N}$

$v? \in nodes$
$nodes' = nodes \setminus \{v?\}$
$s' = s \cap y?$

$Choose$
$x! : \mathbb{N}$

$nodes = \varnothing$
$x! \in s$

In this description A and B now communicate with C via an operation *Send*, and C records, in the variable s, those values which are acceptable to both A and B. When both components have sent their preferences, C will communicate an acceptable chosen value via *Choose*. This operation can then be synchronised with a component taking in as input the values agreed by A and B.

Clearly, Sys_1 and Sys_2 have the same observable behaviour, and using *structural simulation rules* [3] which allow the structure of an integrated Object-Z and CSP specification to be altered in a refinement, we can show that Sys_1 is refined by Sys_2.

The structural simulation rules allow refinements to be verified even if the overall CSP structure of the integrated specification has been altered in a development step, and rules have been derived that allow components to be introduced and removed using each of the commonly used CSP operators. For example, there are simulation rules to refine a specification E into a specification $F \| G$ since, even though there is no correspondence between individual components, the simulation rules check whether the overall observable behaviour in $F \| G$ is consistent with that defined in E.

To verify the refinement above we proceed using four steps which, as we see in the following, introduce intermediate classes in order to verify the refinement

($\sqsubseteq$ denotes “is refined by”):

$$\begin{aligned} Sys_1 = A_1 \| B_1 &\sqsubseteq D_1 \\ &\sqsubseteq D_2 \setminus \{\!|\ Send\ |\!\} \\ &\sqsubseteq (C\ {}_X\|_Y\ D_3) \setminus \{\!|\ Send\ |\!\} \\ &\sqsubseteq (C\ {}_X\|_Y\ (A_2 ||| B_2)) \setminus \{\!|\ Send\ |\!\} \end{aligned}$$

Full details of the form of structural simulation rules can be found in [3]. Here our purpose is to illustrate their use, and we do not give all the rules in full. First of all we note that the intermediate classes we need in the refinement are as follows.

D_1

$Choose$
$x! : \mathbb{N}$

$x! \in \{1, 2\} \cap \{2, 3\}$

D_3

$Send$
$v! : \{1, 2\}$
$y! : \mathbb{P}\,\mathbb{N}$

$(v! = 1 \wedge y! = \{1, 2\})$
$\vee$
$(v! = 2 \wedge y! = \{2, 3\})$

D_2

$nodes : \mathbb{P}\{1, 2\}$
$s : \mathbb{P}\,\mathbb{N}$

$\textit{INIT}$
$nodes = \{1, 2\} \wedge s = \mathbb{N}$

$Send$
$\Delta(nodes, s)$
$v? : \mathbb{N}$
$y? : \mathbb{P}\,\mathbb{N}$

$(1 \in nodes \wedge nodes' = nodes \setminus \{1\}$
$\wedge s' = s \cap \{1, 2\})$
$\vee$
$(2 \in nodes \wedge nodes' = nodes \setminus \{2\}$
$\wedge s' = s \cap \{2, 3\})$

$Choose$
$x! : \mathbb{N}$

$nodes = \varnothing$
$x! \in s$

Let us consider a few of the steps involved. To verify the step $D_2 \sqsubseteq (C\ {}_X\|_Y\ D_3)$, and hence the step $D_2 \setminus \{\!|\ Send\ |\!\} \sqsubseteq (C\ {}_X\|_Y\ D_3) \setminus \{\!|\ Send\ |\!\}$, we use the structural simulation rule shown in Definition 2 for introducing a parallel composition. This simplified form of the rule assumes that if an operation Op is shared between components F and G, then FOp has input $z?$ and GOp has corresponding output $z!$.

Definition 2 *Parallel downward simulation*
A CSP expression $F\ {}_A\|_B\ G$ is a downward simulation of the Object-Z class E if F and G satisfy the following for some retrieve relation R and each Op in both A and B.

PS.1 $\forall F.\textsc{Init} \wedge G.\textsc{Init} \bullet \exists E.\textsc{Init} \bullet R$
PS.2 $\forall E.\textsc{State};\ F.\textsc{State};\ G.\textsc{State} \bullet$
$R \implies (\text{Pre } EOp \iff \text{Pre } (FOp[z!/z?] \wedge GOp))$
PS.3 $\forall E.\textsc{State};\ F.\textsc{State};\ G.\textsc{State};\ F.\textsc{State}';\ G.\textsc{State}' \bullet$
$R \wedge FOp[z!/z?] \wedge GOp \implies (\exists E.\textsc{State}' \bullet EOp \wedge R')$

(The derivation of a similar rule for demonic outputs can be found in [3]. Note however that by using the blocking plus angelic model, we do not place any restrictions on the outputs of the refinement as is done in [3].)

Application of this rule requires its verification for the initialisation and operations *Send* and *Choose*. To do so, we will use the identity retrieve relation. Since *Choose* does not appear in Y, the conditions with respect to that operation are easily discharged by the fact that *CChoose* is identical to $D_2 Choose$ [3]. For *Send*, we are required to verify conditions such as **PS.2**:

$$\text{Pre } D_2 Send \iff \text{Pre } (CSend[v!/v?, y!/y?] \wedge D_3 Send)$$

The predicate of Pre $D_2 Send$ simplifies to $nodes \neq \varnothing$, as does the predicate of Pre $(CSend[v!/v?, y!/y?] \wedge D_3 Send)$. This condition is therefore easily discharged. Verification of correctness (**PS.3**) for *Send* is done in a similar fashion.

The structural rules allowing the introduction of an interleaving, as in $D_3 \sqsubseteq A_2 ||| B_2$, are similar and we omit the verification here, as we similarly do for the step $A_1 \| B_1 \sqsubseteq D_1$.

The refinement step $D_1 \sqsubseteq D_2 \setminus \{\!|\ Send\ |\!\}$ which involves the introduction of a hidden operation, is slightly more involved. Here, under the assumption $D_2 \setminus \{\!|\ Send\ |\!\}$ contains no divergence, the relevant rule is (for our particular specifications):

Definition 3 *Weak downward simulation*
The CSP expression $D_2 \setminus \{\!|\ Send\ |\!\}$ is a weak downward simulation of the Object-Z class D_1 if there is a retrieve relation R such that the following holds.

WS.1 $\forall D_2.\textsc{State} \bullet D_2.\textsc{Init} \mathbin{⨾} Int_{Send} \implies (\exists D_1.\textsc{Init} \wedge R)$
WS.2 $\forall D_1.\textsc{State};\ D_2.\textsc{State} \bullet R \implies$
$(\text{Pre } D_1 Choose \iff \text{Pre } (Int_{Send} \mathbin{⨾} D_2 Choose))$
WS.3 $\forall D_1.\textsc{State};\ D_2.\textsc{State};\ D_2.\textsc{State}' \bullet$
$R \wedge (Int_{Send} \mathbin{⨾} D_2 Choose \mathbin{⨾} Int_{Send}) \implies$
$(\exists D_1.\textsc{State}' \bullet R' \wedge D_1 Send)$

Here Int_{Send} represents the effect of the hidden event Send and is found by taking zero or more occurrences of Send, i.e., $Int_{Send} \mathrel{\widehat=} \Xi D_2 State \vee Send \vee (Send \mathbin{⨾} Send) \vee \ldots$ (This can, in fact, be written using the schema calculus, see [3, 2].)

The verification of this involves comparing the effect of *Choose* in D_1 with $Int_{Send} \mathbin{⨟} Choose \mathbin{⨟} Int_{Send}$ in D_2. In calculating $Int_{Send} \mathbin{⨟} Choose \mathbin{⨟} Int_{Send}$ in D_2 we note that *Send* can occur up to two times, then *Choose* will definitely be enabled. We then easily see that the effect of *Choose* in D_1 is the same as the effect of *Choose* in $D_2 \setminus \{\!| \mathit{Send} |\!\}$, and the conditions can be formally verified if necessary.

Putting the pieces together, we find that all the refinement steps in the sequence can be verified and, therefore, $A_1 \| B_1 \sqsubseteq (C\ {}_X\|_Y (A_2 ||| B_2)) \setminus \{\!| \mathit{Send} |\!\}$.

In summary, what we have shown is that the blocking model with an angelic model of outputs has not constrained the design to necessarily adopt the abstract communication mechanism that motivated it. By using structural refinements, we can introduce explicit mechanisms to communicate and negotiate between the components. We can, therefore, move smoothly between an abstract view and a more implementation-oriented view of the same behaviour and, furthermore, this is achieved using the same angelic model of outputs throughout.

4.2 Hotel example

To further illustrate this approach, consider the specification of the hotel booking system in Section 3. We can refine the specification to reflect more closely one of the strategies for realising the communication specified abstractly.

For example, the implementation of the second strategy discussed at the end of Section 4 would involve introduction of an additional operation *MakeBooking*, which on the customer's behalf indicates that they wish to make a booking, and allows the hotel to pass over a table of available rooms and dates. To ensure a double booking is not attempted, the hotel will give this information to one customer at a time (implemented by a *lock*). When making a *Booking*, a customer will select a room and date from those that he/she knows for certain are available (since they are in the table). Then, in the synchronisation of the booking operation, the inputs and outputs will always be able to synchronise, allowing implementation where outputs are demonic.

The specification of such a strategy can be given as follows (where we use inheritance to shorten the description of the classes).

Customer$_1$

Customer

$table : RoomType \leftrightarrow Date$

INIT

$table = \varnothing$

MakeBooking

$\Delta(table)$

$table? : RoomType \leftrightarrow Date$

$table' = table?$

Book

$\Delta(table)$

$(r!, d!) \in table$

$table' = \varnothing$

$Hotel_1$

$Hotel$

$lock : \mathbb{B}$

$\textsc{Init}$

$\neg lock$

$MakeBooking$

$\Delta(lock)$
$table! : RoomType \leftrightarrow Date$

$\neg lock \land lock'$
$(r, d) \in table! \Leftrightarrow$
$\quad (\exists\, i : \operatorname{dom} rooms \bullet$
$\qquad r = rooms(i) \land$
$\qquad i \notin \operatorname{dom} booked(d))$

$Book$

$\Delta(lock)$

$lock \land \neg lock'$

The complete specification being

$$System_1 = ((|||_{n:Name}\ Customer_{1\{name \mapsto n\}}) \parallel Hotel_1) \setminus \{MakeBooking\}$$

We can show that $System_1$ is a refinement of $System$ by employing structural refinements to show

$$(|||_{n:Name}\ Customer_{\{name \mapsto n\}}) \| Hotel$$
$$\sqsubseteq$$
$$((|||_{n:Name}\ Customer_{1\{name \mapsto n\}}) \| Hotel_1) \setminus \{MakeBooking\}$$

The details are omitted.

5 Conclusion

This paper has investigated a particular semantic interpretation of preconditions and outputs in the context of integrations of Object-Z and CSP. It has shown that by adopting a *blocking* model of operations, where operations cannot occur outside their preconditions, and an *angelic* model of outputs, where outputs may be influenced by their environment, we can specify concurrent systems at a higher level of abstraction. In particular, exceptional behaviours need not be specified in many cases and mechanisms for communication between processes can be largely ignored.

We have also shown that adopting an angelic model of outputs does not preclude compositional refinement, nor refinement to an implementation in which outputs are no longer influenced by the environment, i.e., are *demonic*. The latter is achieved by introducing an internal operation which "chooses" a value to output before the operation which outputs it. This approach can also be used when initially specifying systems providing the internal non-determinism associated with demonic outputs within the angelic interpretation.

References

1. MJ. Butler. Refinement and decomposition of value-passing action systems. In E. Best, editor, *International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 217–232. Springer-Verlag, 1993.
2. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2001.
3. J. Derrick and G. Smith. Structural refinement in Object-Z/CSP. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *2nd International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 194–213. Springer-Verlag, 2000.
4. R. Duke and G. Rose. *Formal Object-Oriented Specification using Object-Z*. MacMillan, 2000.
5. C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.
6. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. B.P. Mahony and J.S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.
8. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
9. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
10. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.
11. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
12. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M.G. Hinchey and Shaoying Lui, editors, *First International Conference on Formal Engineering Methods (ICFEM '97)*, pages 293–302. IEEE Computer Society Press, 1997.
13. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems – an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2000.
14. J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.