Guards, Preconditions and Refinement in Z

Ralph Miarka, Eerke Boiten, John Derrick

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

Email: {rm17,E.A.Boiten,J.Derrick}@ukc.ac.uk

Abstract. In the common Z specification style operations are, in general, partial relations. The domains of these partial operations are traditionally called preconditions, and there are two interpretations of the result of applying an operation outside its domain. In the traditional interpretation anything may result whereas in the alternative, guarded, interpretation the operation is blocked outside its precondition.
In fact these two interpretations can be combined, and this allows representation of both refusals and underspecification in the same model. In this paper we explore this issue, and we extend existing work in this area by allowing arbitrary predicates in the guard.
To do so we adopt a non-standard three valued interpretation of an operation by introducing a third truth value. This value corresponds to a situation where we don't care what effect the operation has, i.e. the guard holds but we may be outside the precondition.
Using such a three valued interpretation leads to a simple and intuitive semantics for operation refinement, where refinement means reduction of undefinedness or reduction of non-determinism. We illustrate the ideas in the paper by means of a small example.

1 Introduction

In the states-and-operations (abstract data type) specification style in Z, operations are in general partial relations. The domains of these partial relations are traditionally called *preconditions*. Depending on which context the abstract data types are used in, there are two interpretations of the result of applying an operation outside its domain. In the traditional interpretation [11], anything may happen outside the precondition (including divergence); in the blocking (guarded) interpretation the operation is not possible. The latter interpretation is the common one when modelling reactive systems or combining Z with process algebra, and also in Object-Z. It is also called 'firing condition' or 'enabling condition' interpretation [9].

It has been observed that it is often convenient to use a combination of these two interpretations, which allows both modelling of refusals and underspecification. One way of doing this is by having explicit guards as in B [1] or in Fischer's work [5]. In this paper we generalise existing work by allowing arbitrary predicates in

the guard. Furthermore, we give a model of refinement, refining both guard and precondition.

Our inspiration comes from a non-standard semantics of operations, viz. an interpretation in three-valued logic. The third logic value is called "don't care", denoted $\perp$. We do occasionally refer to "undefinedness", although this should probably be distinguished from the kind of undefinedness discussed by Valentine [15] and solved by VDM's third logic value. Using a three-valued logic leads to a simple and intuitive notion of (operation) *refinement*: refinement is reduction of undefinedness or reduction of non-determinism (or both). It would even allow an alternative definition of refinement which preserves "required non-determinism" [10, 12].

However, such an interpretation of operations requires a more expressive notation than normal operations with explicit guards. In that notation, we take the operation to be false (impossible) outside its guard, and undefined where the guard holds but not the precondition. Clearly this allows us to state that, for certain before states, *any* after state "is undefined", but not that some after states are undefined, and others possible or impossible. We will define a syntax which *is* sufficiently expressive for this semantics, and define operation refinement rules for this which generalise the traditional ones.

The remainder of this work is structured as follows. In Section 2, we will demonstrate by means of an example (a simple money transaction system) that a combination of the traditional and blocking interpretations is sometimes required. Then, in Section 3, we define a schema notation including both guards and effect schemas. Based on that we define regions of operation behaviour, i.e. whether an operation is within or without the guard, or within or without the precondition.These regions can also be defined in a three valued interpretation, which we will give in Section 4. Using such a three valued interpretation leads to a simple and intuitive notion of refinement that generalises classical operation refinement. We introduce the rules in Section 5 and show their compatibility to the classical ones. Finally, we discuss related work (Section 6), as well as summarise our work including a discussion of possible future research (Section 7).

2 Guards and Preconditions in Z

2.1 Example

Consider the following example of a simple money transaction system. It allows to transfer a positive amount of money to a person's bank account. Therefore, we need a set of bank account holders

$$[PID]$$

Each bank account is characterised by its holder and the amount of money in it. Of course, we allow negative amounts in the account as well. On the other

hand, not every person in the above set has to have a bank account, therefore, a collection of accounts is a partial function. Further, *total* is a derived state component which calculates the amount of money in our bank by taking the sum of the money in all accounts.

Bank

$$account : PID \pfun \mathbb{Z}$$
$$total : \mathbb{Z}$$
$$total = \sum x : \mathrm{dom}\, account \bullet account(x)$$

We describe a transaction that will transfer a given amount of money to someone's bank account. Clearly the amount transfered has to be positive, because we do not want to be able to decrease someone else's account.

transfer

$$\Delta Bank$$
$$a? : \mathbb{Z}$$
$$p? : PID$$
$$a? \geq 0$$
$$p? \in \mathrm{dom}(account)$$
$$account' = account \oplus \{p? \mapsto account(p?) + a?\}$$

2.2 Classical Precondition and Guarded Interpretation

In the above example, two conditions have to be fulfilled for a transfer to be successful. On the one hand, the amount must be positive and on the other hand the receiving person must have an account. These conditions are expressed in the following schema:

pre *transfer*

$$Bank$$
$$a? : \mathbb{Z}$$
$$p? : PID$$
$$a? \geq 0 \land p? \in \mathrm{dom}\, account$$

which is obtained as usual by existentially quantifying over the after state in *transfer*.

But what happens if we try to apply the operation outside of these conditions? There are two possible interpretations: the precondition interpretation, allowing the operation, and the guarded interpretation, preventing it. A related issue is refinement, the development from a specification towards a more concrete representation. How do both interpretations deal with it?

In the classical Z interpretation a precondition represents the set of states where the operation is defined, i.e. guaranteed to produce the specified result. Outside the precondition the operation is considered to be undefined which means that the operation can do anything including non-termination ("divergence"). Therefore, refinement can, apart from reduction of non-determinism, weaken a precondition, allowing one to widen the scope of the operation and thereby reduce the area of undefinedness.

Other specification languages, like Object-Z, treat the precondition differently. There the precondition is considered as a guard, blocking the operation if the precondition is not fulfilled. Such an interpretation is occasionally used in Z as well, for example, when modelling reactive systems (see for example [9, 13]). Refinement of guards is treated differently. In Object-Z, for example, one is not allowed to change the guard. However, other approaches, like [10] where preconditions and guards are combined, allow strengthening of guards, i.e. the reduction of the applicability of the operation. They also allow to weaken any precondition. However, the precondition is the upper bound for strengthening the guard and the guard is the lower bound for weakening the preconditions.

2.3 Refinement

In the precondition interpretation, the following two refinements of *transfer* would be possible, each of them weakening one of the constraints of pre *transfer*. First, we could allow the creation of accounts:

C$_1$*_transfer*

$$\Delta Bank$$
$$a? : \mathbb{Z}$$
$$p? : PID$$

$$a? \geq 0$$
$$p? \notin \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto a?\}$$
$$p? \in \mathrm{dom}(account) \Rightarrow account' = account \oplus \{p? \mapsto account(p?) + a?\}$$

This appears a sensible refinement, however, in the guarded interpretation it would be forbidden.

The guarded interpretation also forbids the more dangerous

C$_2$*_transfer*

$$\Delta Bank$$
$$a? : \mathbb{Z}$$
$$p? : PID$$

$$p? \in \mathrm{dom}(account)$$
$$account' = account \oplus \{p? \mapsto account(p?) + a?\}$$

which, by removing the requirement that $a? \geq 0$ suddenly allows withdrawal of someone else's money. In the precondition interpretation this is still a valid refinement, though.

Apparently, the two predicates in pre *transfer* have a different status: $a? \geq 0$ is more like a guard, whereas $p? \in \mathrm{dom}(account)$ is more like a precondition. Our example shows that each interpretation alone is not always sufficient. Therefore, we want to have both guards and preconditions in the same specification.

2.4 Combining Guards and Preconditions

The idea is not new and there are a number of essentially identical approaches. For example, Fischer [4, 5] provides a solution to this problem by using an "enabled" schema to denote the guard and an "effect" schema for the classical operation schema with its precondition interpretation. Using this approach the *transfer* operation in our example evolves to

F_transfer

`enable`*_transfer*

$a? : \mathbb{Z}$

$a? \geq 0$

`effect`*_transfer*

$\Delta Bank$
$a? : \mathbb{Z}$
$p? : PID$

$p? \in \mathrm{dom}(account)$
$account' = account \oplus \{p? \mapsto account(p?) + a?\}$

where `enable` refers to the guard of the operation and `effect` to the effect of the operation. Now the operation *F_transfer* is blocked whenever $a?$ is negative. However, the update of someone's account is only guaranteed if the account already exists. In case it does not divergence may occur.

With this notation we are able to develop refinement rules which deal with the guards and preconditions in an appropriate fashion. Such refinement rules would allow one to weaken the precondition of *F_transfer* (i.e. `effect`*_transfer*), reduce any non-determinism in the specification, and potentially strengthen the guard (i.e. `enable`*_transfer*). With these rules in place we are able to weaken the precondition $p? \in \mathrm{dom}(account)$ provided we do preserve the guard $a? \geq 0$.

However, according to Fischer [5] the guard "must contain unprimed state variables only". Unfortunately, this would still allow undesired refinements, as the after state is completely unconstrained for before states satisfying the guard but not the precondition. Sensible restrictions like

$$\{p?\} \lhd account' = \{p?\} \lhd account$$
and $total' = total + a?$

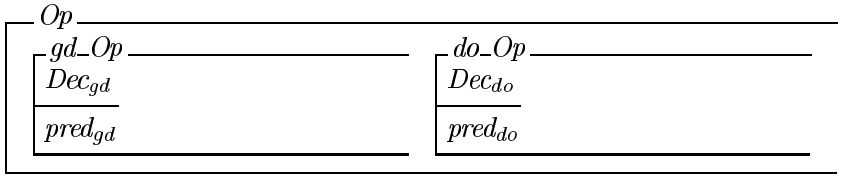
which express that no one else's account changes and that the total amount of money cannot exceed the previous amount plus the newly added, cannot be imposed. Adding this restriction to `effect_`*transfer* would have no effect, because it can be derived from `effect_`*transfer* already. However, for states currently outside the precondition but within the guard, we have no way of imposing this as a postcondition.

3 A Syntax for Using Generalised Guards

In this section we introduce the syntax to describe an operation in terms of guards and preconditions. We then use this characterisation to define the different regions of definition that an operation can have. The operation syntax we introduce again splits an operation into two parts consisting of its guard and its effect in a way similar to that described in Section 2.4.

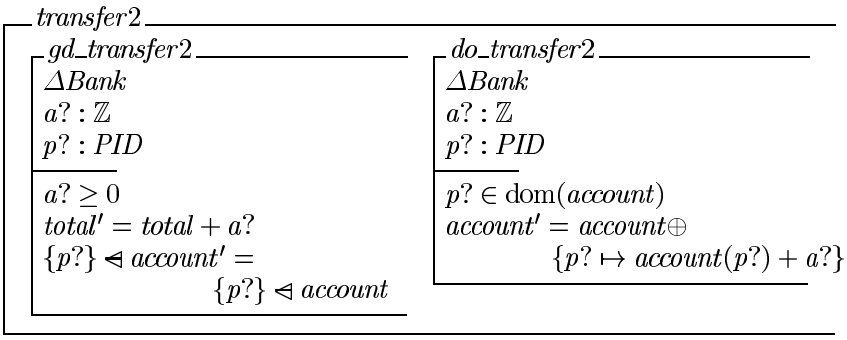
3.1 Operations with Guards and Preconditions

An operation Op is a defined as a pair (gd_Op, do_Op), where gd denotes the guard of the operation and do the classical operation itself, and it is given by a schema



such that Dec denote the declarations of the guard and operation respectively, and, we require that Dec_{gd} are (textually) contained in Dec_{do}. One could require that $do\_Op \Rightarrow gd\_Op$, though we avoid such a restriction by using $gd\_Op \wedge do\_Op$ rather than just do_Op in any situation where such a restriction would be relevant. In particular, when we refer to just Op in an expression, this is taken to be an abbreviation for $gd\_Op \wedge do\_Op$. Note, that whenever $\neg\ gd\_Op$ holds we do not care about do_Op anymore. Note as well, that in gd_Op we allow any arbitrary predicate which may involve after states (S') too, and indeed, the signature reflects this.

For example, the previous discussed operation *transfer* with the desired extension of the guard can now be expressed as



Having primed state variables in the guard causes the guard not to be executable, because we cannot test the after state beforehand. However, we may consider specifications that contain undefined areas as not implementable anyway, because some refinement is still missing. For refinement rules which remove undefinedness see Section 5 (and 6.4). Primed state variables in the guard do not limit implementations in general, they just give us more expressiveness.

3.2 Regions of Before States

Using such a notation, we can describe (at least) three different possibilities for a particular pair of before/after states:

- *gd_Op* holds and *do_Op* holds: the states belong to the operation.
- *gd_Op* holds but *do_Op* does not hold: the states may or may not belong to the operation, we don't care.
- *gd_Op* does not hold: we do not wish the states to belong to the operation. (Note, that this makes *do_Op* for this pair of states redundant information.)

Based on this description, we can define a number of regions of before states that are of interest. The next section then will formalise this description in a three-valued logic, and Section 5 will present a refinement relation that conforms with the above intuition.

For simplicity, let us take $Dec_{do} = Dec_{gd} = \Delta S$ in the following definitions.

Impossible. The impossible region is the set of states where the operation is blocked, i.e. it is always going to fail.

$$\text{impo}(Op) \mathrel{\widehat=} [S \mid \neg\, \exists\, S' \bullet gd\_Op]$$

Analysing our example, we identify that the operation *transfer*2 is always rejected when the amount $a?$ is negative, i.e. $\text{impo}(transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid a? < 0]$.

Precondition. The precondition region is the area where the operation is possible and well defined. It is defined by

$$\mathrm{pre}(Op) \mathrel{\widehat=} [S \mid \exists S' \bullet gd\_Op \land do\_Op]$$

Observe that this is consistent with our convention of Op denoting $gd\_Op \land do\_Op$. Then this results in the following precondition for our example: $\mathrm{pre}(transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid p? \in \mathrm{dom}(account) \land a? \geq 0]$.

Guard. The guarded region is simply the complement to the impossible region, i.e. it is the area where the blocking predicate holds.

$$\mathrm{guard}(Op) \mathrel{\widehat=} [S \mid \exists S' \bullet gd\_Op]$$

This, however, is the same as calculating the precondition of the guarded part of the operation, i.e. $\mathrm{guard}(Op) = \mathrm{pre}(gd\_Op)$. Then it holds for our example $\mathrm{guard}(transfer2) = \mathrm{pre}(gd\_transfer2) = [Bank, a? : \mathbb{Z}, p? : PID \mid a? \geq 0]$.

Here it is clear that our approach is strictly more expressive than Fischer's: $\mathrm{guard}(Op)$ contains an abstraction of the information in our approach, whereas in his $pre(\texttt{enable}) = \texttt{enable}$. In $transfer2$ the guard is $a? \geq 0$, loosing the information that any widening of the precondition should respect $\{p?\} \lhd account' = \{p?\} \lhd account$ and $total' = total + a?$.

Undefined. Given the regions defined by guard and precondition we could define a "completely undefined" region as the difference between guard and precondition. This would be

$$\mathrm{undef}(Op) \mathrel{\widehat=} [S \mid \exists S' \bullet gd\_Op \land (\neg\ \exists S' \bullet gd\_Op \land do\_Op)]$$

In the initial $transfer$ operation it is $[Bank, a? : \mathbb{Z}, p? : PID \mid a? \geq 0 \land p? \notin \mathrm{dom}(account)]$ whereas in $transfer2$ this region is empty.

4 Three Valued Interpretation

In the last section we defined several regions according to pairs of before/after states. We distinguished three different possibilities: First, the region where gd_Op does not hold, i.e. where the operation should be impossible. Second, the region where both gd_Op and do_Op hold, i.e. where after states belong to the operation. Third, the remaining region where gd_Op holds but do_Op does not hold. In that case the outcome of the operation is undefined. These three regions are depicted in Figure 1 and can be naturally described using a set of three truth values $\{\mathbf{f}, \mathbf{t}, \bot\}$ respectively.

4.1 Semantical Description of the Regions

We want to define val Op to be a mapping from the regions into the three truth values given above. Therefore, we define first the relational representation of an operation schema in the obvious way, such that if Op is an operation on state S

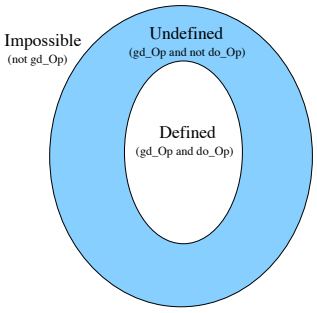


Fig. 1. Guard and Precondition

with input and output, rel Op is a binary relation between bindings of type S plus input and bindings of type S plus output.[1]

Further, we define a three valued boolean-like type by

$$bool3 ::= \mathbf{t} \mid \mathbf{f} \mid \bot$$

Now the three valued interpretation of an operation $Op = (gd\_Op, do\_Op)$ can be defined as follows:

$$\begin{aligned} \text{val } Op = &\{x;\ y \mid (x,y) \in \text{rel } Op \bullet (x,y) \mapsto \mathbf{t}\} \\ &\cup \{x;\ y \mid (x,y) \notin \text{rel } gd\_Op \bullet (x,y) \mapsto \mathbf{f}\} \\ &\cup \{x;\ y \mid (x,y) \in \text{rel } (gd\_Op \wedge \neg do\_Op) \bullet (x,y) \mapsto \bot\} \end{aligned}$$

We use a table style notation to relate before states and after states of an operation by means of the possible outcome, i.e. by val Op. For example, given an operation

Filter

gd_Filter	*do_Filter*
$a? : \mathbb{Z}$	$a? : \mathbb{Z}$
$b! : \mathbb{Z}$	$b! : \mathbb{Z}$
$a? > 0$	$even(a?)$
	$b! \leq a?$

which takes only a positive number as input and returns any number less or equal to it if the given number is even. Then the table representation is

[1] Cf. Appendix A for a definition of rel, and a fully typed version of val.

$a?$ \ $b!$	... -1	0	1	2	3	4	5 ...
⋮							
-1	**f**	**f**	**f**	**f**	**f**	**f**	**f**
0	**f**	**f**	**f**	**f**	**f**	**f**	**f**
1	⊥	⊥	⊥	⊥	⊥	⊥	⊥
2	**t**	**t**	**t**	**t**	⊥	⊥	⊥
3	⊥	⊥	⊥	⊥	⊥	⊥	⊥
4	**t**	**t**	**t**	**t**	**t**	**t**	⊥
5	⊥	⊥	⊥	⊥	⊥	⊥	⊥
⋮							

4.2 Meaning of Refinement

Operation refinement is defined as removal of undefinedness as well as non-determinism. Taking our three valued interpretation and the above representation then we can explain refinement intuitively as replacing any $\perp$ by **t** which may enlarge the precondition region or by replacing any $\perp$ by **f** which in turn may reduce the guarded region. Furthermore, we can replace multiple **t** in a line by **f** (as long as one **t** remains), in order to reduce non-determinism. Note that the later step does not change either the precondition nor the guarded region.

Let us consider our *Filter* operation from above in order to clarify the presented notion of refinement. Therefore, we introduce a possible refinement *C_Filter*.

C_Filter

gd_C_Filter

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$

$a? > 0$
$b! < a?$

do_C_Filter

$a? : \mathbb{Z}$
$b! : \mathbb{Z}$

$even(a?)$
$b! = a?/2$

The following refinement took place. First, we ensure that $b!$ is always less than $a?$. This is done by strengthening the guard and corresponds to changing $\perp$ to **f** for all cases where $b! \geq a?$. Note, that this refinement step also strengthens the postcondition of *Filter* in some cases. Second, we remove non-determinism by providing a more concrete representation of the output in case that $a?$ is even. This is done by replacing multiple **t** by **f**. Weakening of the precondition did not take place but we may define an output for the case that $a?$ is an odd number in another refinement step. However, the result will always be bound by the newly introduced predicate in the guard. The outcome of this refinement step is illustrated in the following table.

$a?$ \ $b!$	... -1	0	1	2	3	4	5 ...
⋮							
-1	**f**	**f**	**f**	**f**	**f**	**f**	**f**
0	**f**	**f**	**f**	**f**	**f**	**f**	**f**
1	⊥	⊥	**f**	**f**	**f**	**f**	**f**
2	⊥	⊥	**t**	**f**	**f**	**f**	**f**
3	⊥	⊥	⊥	⊥	**f**	**f**	**f**
4	⊥	⊥	⊥	**t**	⊥	**f**	**f**
5	⊥	⊥	⊥	⊥	⊥	⊥	**f**
⋮							

5 Operation Refinement

In this work, we will restrict ourselves to operation refinement. Our work is intended to generalise the classical approach of refinement. In this section, we first present our generalised rules of refinement which we then apply to the *transfer* example. Finally, we show that our new refinement conditions indeed generalise both the guarded and the preconditioned approach.

5.1 Rules for Operation Refinement

Given an abstract operation $AOp = (gd\_AOp, do\_AOp)$ and a concrete operation $COp = (gd\_COp, do\_COp)$ both over the same state *State* with input $x? : X$ and output $y! : Y$, then *COp* refines *AOp*, denoted $AOp \sqsubseteq COp$, if and only if applicability (1) and correctness (2) hold:

(1) $\forall State;\ x? : X \bullet \text{pre}\, AOp \vdash \text{pre}\, COp$
(2) $\forall State;\ State';\ x? : X;\ y! : Y \bullet \text{pre}\, AOp \wedge COp \vdash AOp$

The first condition allows to weaken the precondition and the second condition ensures that the refined operation does at least what the abstract operation did.

Additionally, we allow strengthening of guards:

(3) $\forall State;\ State';\ x? : X;\ y! : Y \bullet gd\_COp \vdash gd\_AOp$

Conditions (1) and (3) together ensure that the precondition is the upper bound for strengthening the guard and that the guard is the lower bound for weakening the precondition.

We observe that the correctness rule can be formally weakened using (3):

$$\begin{aligned} & \text{pre}\, AOp \wedge COp \Rightarrow AOp \\ \equiv\ & \{\text{definition of Op}\} \end{aligned}$$

$$
\begin{aligned}
&\quad \mathrm{pre}(gd\_AOp \wedge do\_AOp) \wedge gd\_COp \wedge do\_COp \Rightarrow gd\_AOp \wedge do\_AOp \\
&\equiv \{\text{using } gd\_COp \Rightarrow gd\_AOp\} \\
&\quad \mathrm{pre}(gd\_AOp \wedge do\_AOp) \wedge gd\_COp \wedge do\_COp \Rightarrow do\_AOp \\
&\equiv \{\text{definition of Op}\} \\
&\quad \mathrm{pre}\, AOp \wedge COp \Rightarrow do\_AOp
\end{aligned}
$$

However, it turns out nice that the shape of the classical refinement rules is preserved when we use the introduced abbreviation.

5.2 Example

In Section 2 we introduced a simple money transaction system that allows to put money into the account of an existing customer. We showed via an example that using only the guarded or precondition interpretation limits the expressiveness, and also perhaps allows unintended refinement. In our combined approach we solved these problems. Therefore, we are now able to express the following refinement of the *transfer* operation:

C_transfer

gd_C_transfer

$\Delta Bank$
$a? : \mathbb{Z}$
$p? : \mathbb{Z}$

$a? \geq 0$
$total' = total + a?$
$\{p?\} \lhd account' =$
$\{p?\} \lhd account$

do_C_transfer

$\Delta Bank$
$a? : \mathbb{Z}$
$p? : PID$

$p? \notin \mathrm{dom}(account) \Rightarrow$
$account' = account \oplus \{p? \mapsto a?\}$
$p? \in \mathrm{dom}(account) \Rightarrow$
$account' = account \oplus$
$\{p? \mapsto account(p?) + a?\}$

First, we strengthened the guard *gd_transfer*. Now, the money to be transfered has to be positive and we are not permitted to change another person's bank account, no matter what future refinement will do to the precondition. Second, we also refined the *do_transfer* operation. We weakened the precondition of *transfer* to handle the case that the receiving user does not have an account. In this case we allow the creation of a new bank account which will have the amount $a?$ as initial input.

5.3 Generalisation of Traditional Refinement Rules

Our concept of refinement is a valid generalisation of the traditional operation refinement rules in both the guarded and the preconditioned approach. Taking $gd\_Op = \mathrm{pre}\, Op$ and $do\_Op = Op$ or $gd\_Op = \mathrm{true}$ and $do\_Op = Op$, respectively, we show that our refinement rules reduce to the traditional ones.

Guarded Approach. In the guarded interpretation the guard is the precondition of the operation. Therefore, we use $gd\_Op = \text{pre}\, Op$ and $do\_Op = Op$.

Let $Op_1 = (gd\_Op_1, do\_Op_1) = (\text{pre}\, AOp, AOp)$ and $Op_2 = (gd\_Op_2, do\_Op_2) = (\text{pre}\, COp, COp)$. We show that for this choice of Op_1, Op_2 it holds $Op_1 \sqsubseteq Op_2 \equiv AOp \sqsubseteq COp$ in the guarded approach.

(1) Applicability.

$$
\begin{aligned}
& \text{pre}\, Op_1 \vdash \text{pre}\, Op_2 \\
\equiv\ & \{Op = (gd\_Op \wedge do\_Op)\} \\
& \text{pre}(gd\_AOp \wedge do\_AOp) \vdash \text{pre}(gd\_COp \wedge do\_COp) \\
\equiv\ & \{gd\_Op = \text{pre}\, Op \,\text{and}\, do\_Op = Op\} \\
& \text{pre}(\text{pre}\, AOp \wedge AOp) \vdash \text{pre}(\text{pre}\, COp \wedge COp) \\
\equiv\ & \{\text{simplification: pre}\, Op \wedge Op \equiv Op\} \\
& \text{pre}\, AOp \vdash \text{pre}\, COp
\end{aligned}
$$

(2) Correctness.

$$
\begin{aligned}
& \text{pre}\, Op_1 \wedge Op_2 \vdash Op_1 \\
\equiv\ & \{Op = (gd\_Op \wedge do\_Op)\} \\
& \text{pre}(gd\_AOp \wedge do\_AOp) \wedge (gd\_COp \wedge do\_COp) \vdash (gd\_AOp \wedge do\_AOp) \\
\equiv\ & \{gd\_Op = \text{pre}\, Op \,\text{and}\, do\_Op = Op\} \\
& \text{pre}(\text{pre}\, AOp \wedge AOp) \wedge (\text{pre}\, COp \wedge COp) \vdash (\text{pre}\, AOp \wedge AOp) \\
\equiv\ & \{\text{simplification: pre}\, Op \wedge Op \equiv Op\} \\
& \text{pre}\, AOp \wedge COp \vdash AOp
\end{aligned}
$$

(3) Strengthening.

$$
\begin{aligned}
& gd\_Op_2 \vdash gd\_Op_1 \\
\equiv\ & \{gd\_Op_1 = \text{pre}\, AOp, gd\_Op_2 = \text{pre}\, COp\} \\
& \text{pre}\, COp \vdash \text{pre}\, AOp
\end{aligned}
$$

Applicability and strengthening together result in the fact the $\text{pre}\, COp = \text{pre}\, AOp$, i.e. the classical condition in Object-Z that a guard cannot be strengthened nor weakened. The correctness rule is as in classical refinement as well.

Precondition Approach. In order to show that our approach is a generalisation of the precondition approach, we consider that the guard of the operation is the weakest possible, i.e. $gd\_Op = \text{true}$. Then our notation coincides with the classical one where $do\_Op = Op$. Using the fact that we consider $Op = gd\_Op \wedge do\_Op$ it is easy to show that applicability (1) and correctness (2) hold. The rule for strengthening (3) evaluates to $\forall State;\ State';\ x? : X;\ y! : Y \bullet \text{true}$ which means there is no strengthening at all. Therefore, in the case of no guards our refinement rules are equivalent to the classical ones.

6 Related and Further Work

6.1 Strulo's Work

In [13] Strulo attempts to unify both the precondition and the guarded interpretation in order to model passive and active behaviour in Z accordingly. In his work, Strulo uses the term firing condition rather than guard. An operation is then described by a single state schema, plus a label indicating whether the operation is either active or passive. A distinction is made between active operations being impossible or divergent, by interpreting before states which allow all possible after states as divergent. This encoding extends the guarded approach but is somewhat artificial. In particular, addition or removal of state invariants has subtle consequences for which states belong to the "impossible" or "divergent" regions.

6.2 The (R, A)-Calculus

Doornbos' (R, A)-calculus [3] separates well-definedness of an operation from its effect, in an abstract setting of binary relations and sets. An operation (R, A) consists of a set A essentially representing its precondition, and a relation R specifying its effect. This is substantially different from having a relation with an explicit *guard*, in particular it allows the specification of "miracles". The fragment of the calculus satisfying $A \subseteq \operatorname{dom} R$ (i.e., the "law" of the excluded miracle), is generalised by our calculus, viz. $(gd\_Op, do\_Op) \mathrel{\widehat{=}} (R, A \lhd R)$. Doornbos also draws a parallel between the (R, A) calculus and weakest (liberal) preconditions which suggests a similar exercise would be possible for our calculus.

6.3 Hehner and Hoare's Predicative Approach to Programming

In [6–8] the authors consider a specification to be a predicate of the form $P \Rightarrow Q$ meaning that if P is satisfied, then the computation terminates and satisfies Q. A specification S is refined by a specification T if all computations satisfying T also satisfy S, i.e. the reverse implication $S \Leftarrow T$ ($T \sqsupseteq S$). This allows weakening of the precondition P as well as strenghtening of the postcondition Q.

Within this approach, the predicate $guard \wedge (pre \Rightarrow post)$ in a schema body would express nearly the desired effect under the guarding interpretation of Z schemas. In this interpretation, a false *guard* causes the specification to be false, i.e. impossible, and a false precondition *pre* leads to the specification being true, which in turn allows any output.

However, the advantage of our approach with two schemas *gd* and *do* is a certain independence of the guard and precondition. Even when the precondition is false, not every output is permitted: it is still restricted by the guard.

6.4 Refinement Rules for Required Non-Determinism

A different interpretation is possible for the operations in three-valued logic that we have described. Various authors (e.g. [10, 12]) have argued that for behavioural specifications, the traditional identification of non-determinism with implementation freedom is unsatisfactory. They would like the opportunity to specify *required* non-determinism, which implies a need for additional specification operators to express implementation freedom. Refinement rules should then remove implementation freedom but not non-determinism. Steen et al [12] describe such a calculus, obtained by adding a disjunction operator to LOTOS.

We could give a similar calculus in Z by reinterpreting the three-valued operations described above. As before, when the operation evaluates to **f** for a particular before and after state, it denotes an impossibility. However, the collection of after states that are related by **t** to a particular before state represent *required* non-determinism. As a consequence, none of these **t** values may be removed in refinement. Finally, the collection of after states that are related by $\bot$ to a particular before state represent an implementation choice, i.e. at least one of those after states will need to be related by **t** in a final refinement.

As a consequence, expressed in terms of the tabular representation used before, refinement rules for required non-determinism and disjunctive specification are:

- if a line contains a single $\bot$, it is equivalent to **t** (required choice from a singleton set);
- if a line contains multiple occurrences of $\bot$, some but not all of them may be changed to **f** (reducing possibility of choice);
- any $\bot$ may be changed to **t** (in particular, an implementation choice between several after states may be refined to a non-deterministic choice between some of them).

This approach generalises only the guarded approach – the precondition just characterises those before states for which possible after states have been determined already. It also prevents some undesired interaction between removing undefinedness and increasing determinism.

7 Conclusion and Future Work

In this work we presented the idea of using a three-valued interpretation of operations to combine and extend the guarded and precondition approaches. Using this non-standard interpretation we were able to present a simple and intuitive notion of operation refinement, which generalizes the traditional refinement relations.

A full theory of refinement would also include a notion of data refinement. However, when the retrieve relation is a two-valued predicate the extension becomes

natural. It remains an open question what might be represented by a three-valued retrieve relation.

In our interpretation of pairs of schemas (gd_Op, do_Op) we identified only three regions. Clearly, we could further distinguish the areas $\neg\ gd\_Op \wedge \neg\ do\_Op$ and $\neg\ gd\_Op \wedge do\_Op$. The latter area might be regarded as representing "miracles" or inconsistency. Detecting and managing inconsistency between the guarded and the preconditioned region is another of our topics for future research.

Further, we would like to develop a schema calculus for the operators of three-valued logic.

Acknowledgement

We like to thank all the anonymous referees for their corrections and helpful suggestions in order to improve this work.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Jonathan P. Bowen, Andreas Fett, and Michael G. Hinchey, editors. *ZUM '98: The Z Formal Specification Notation, Proceedings of the 11th International Conference of Z Users.* Lecture Notes in Computer Science 1493. Springer Verlag, Berlin Heidelberg New York, September 1998.
3. H. Doornbos. A relational model of programs without the restriction to Egli-Milner constructs. In E.-R. Olderog, editor, *PROCOMET '94*, pages 357–376. IFIP, 1994.
4. Clemens Fischer. CSP-OZ: A Combination of Object-Z and CSP. Technical Report TRCF-97-2, Universität Oldenburg, Fachbereich Informatik, PO Box 2503, 26111 Oldenburg, Germany, April 1997. Online: `http://theoretica.Informatik.Uni-Oldenburg.DE/~fischer/techreports.html` (last access 10/01/2000).
5. Clemens Fischer. How to Combine Z with Process Algebra. In Bowen et al. [2], pages 5–23.
6. Eric C. R. Hehner. *A practical theory of programming.* Springer Verlag, 1993.
7. Eric C. R. Hehner. Specifications, programs, and total correctness. *Science of Computer Programming*, 34(3):191–205, July 1999. Online `http://www.elsevier.com/cas/tree/store/scico/sub/1999/34/3/563.pdf` (last access: 09/05/2000).
8. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming.* Prentice Hall, 1998.
9. Mark B. Josephs. Specifying reactive systems in Z. Technical Report PRG-19-91, Programming Research Group, Oxford University Computing Laboratory, 1991.
10. K. Lano, J. Bicarregui, J. Fiadeiro, and A. Lopes. Specification of Required Nondeterminism. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, Lecture Notes in Computer Science 1313, pages 298–317. Springer-Verlag, September 1997.

11. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd., 2nd edition, 1992. Online: `http://spivey.oriel.ox.ac.uk/~mike/zrm/index.html`(last access 26/07/1998).
12. M.W.A. Steen, H. Bowman, J. Derrick, and E.A. Boiten. Disjunction of LOTOS specifications. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Description Techniques and Protocol Specification, Testing and Verification: FORTE X / PSTV XVII '97*, pages 177–192, Osaka, Japan, November 1997. Chapman & Hall. Online: `http://www.cs.ukc.ac.uk/pubs/1997/350` (last access: 20/01/2000).
13. Ben Strulo. How Firing Conditions Help Inheritance. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM'95: The Formal Specification Notation*, Lecture Notes in Computer Science 967, pages 264–275. Springer Verlag, 1995.
14. Ian Toyn. *Z Notation: Final Committee Draft, CD 13568.2*, August24 1999. Online: `http://www.cs.york.ac.uk/~ian/zstan/` (last access 09/05/2000).
15. S. H. Valentine. Inconsistency and Undefinedness in Z – A Practical Guide. In Bowen et al. [2], pages 233–249.

Appendix A: Relational View of Operations

In this appendix we give a formal definition of the relational view of an operation schema, as a binary relation between the appropriate sets of bindings. Binding types are not first class citizens in "traditional" Z, but using notations and conventions from the Draft Z Standard [14] we can provide a sensible typing to the operations defined here.

Define the signature of a schema by changing its predicate to true:

$$\Sigma Op = Op \vee \neg Op$$

Using the precondition operator, we can define "before" and "after" signatures of a schema by:

$$\Sigma_{bef}\, Op = \Sigma(\mathrm{pre}\, Op)$$

$$\Sigma_{aft}\, Op = \exists\, \Sigma_{bef}\, Op \bullet \Sigma\, Op$$

By the conventional interpretation of the precondition operator, $\Sigma_{bef}\, Op$ will contain Op's before state and any inputs; $\Sigma_{aft}\, Op$ contains its after state and any outputs.

In order to provide a type for the relational view of an operation, we have to define the types of "before"-bindings and "after"-bindings of an operation. This could be done explicitly using quantification and filtering over sets of names as in the Draft Standard for pre, but also using just its $[\sigma]$ notation for binding types.

Every (well-defined) schema Op has a unique type of the form $\mathbb{P}[\sigma]$. Let us denote this σ by b^{Op}; define $b^{Op}_{bef} = b^{\Sigma_{bef} Op}$ and analogously b^{Op}_{aft}. Then the relational view of an operation is defined by

$$\text{rel } Op = \{x : [b^{Op}_{bef}];\ y : [b^{Op}_{aft}] \mid \exists\, \Sigma Op \bullet x = \Theta \Sigma_{bef} Op \wedge y = \Theta \Sigma_{aft} Op \wedge Op \bullet (x, y)\}$$

The definition of val Op as given in Section 4 actually requires a slight modification when ΣOp and Σgd_Op are different. Let the extension ext of Op_1 to the signature of Op_2 be defined by:

$$Op_1 \text{ ext } Op_2 = [\Sigma Op_2 \mid Op_1]$$

Then

$$\begin{aligned}\text{val } Op = {} & \{x : [b^{Op}_{bef}];\ y : [b^{Op}_{aft}] \mid (x, y) \in \text{rel } Op \bullet (x, y) \mapsto \mathbf{t}\} \\ & \cup \{x : [b^{Op}_{bef}];\ y : [b^{Op}_{aft}] \mid (x, y) \notin \text{rel}(gd\_Op \text{ ext } do\_Op) \bullet (x, y) \mapsto \mathbf{f}\} \\ & \cup \{x : [b^{Op}_{bef}];\ y : [b^{Op}_{aft}] \mid (x, y) \in \text{rel } (gd\_Op \wedge \neg do\_Op) \bullet (x, y) \mapsto \bot\}\end{aligned}$$