Z AND THE REFINEMENT CALCULUS

by

Steve King

Technical Monograph PRG-79
ISBN 0-902928-57-9

February 1990

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Author's address:
Oxford University Computing Laboratory
Programming Research Group
8–11 Keble Road
Oxford OX1 3QD
England

Electronic mail: king@uk.ac.oxford.prg (JANET)

Z and the Refinement Calculus

Steve King

Abstract

Z has been developed as a formal specification notation, and, as such, has been used successfully for a number of years. Recently, other formal notations, the various flavours of refinement calculi, have emerged. They have been designed as wide spectrum languages to support the whole of the development cycle, from abstract specification through to executable code. We explore the differences between Z and the refinement calculus, and explain the reasons for some of those differences.

We also examine how a development might use both notations, thus giving a path to code from a Z specification. Some rules for switching between the notations are given, and their use is illustrated in a case study.

1 Introduction

Over the last few years, the formal language Z has been used with some success for the specification of large software systems. However, there has been significantly less use of Z in the later stages of the software lifecycle. In some cases, Z has been used to document designs as well as specifications, but in very few cases have all the proofs been done. Research has therefore been carried out (at Oxford and elsewhere) into *usable* notations for developing programs from formal specifications. One of the results of this work has been the Refinement Calculus. This notation differs from Z in small, but significant, ways, and it is the purpose of this paper to explore some of these differences, to give some of the rationale behind the changes in notation, and finally to show how it is possible (and beneficial) to use both Z and the refinement calculus together in a development.

The rest of the paper has the following structure: section 2 contains a brief introduction to the two notations, and section 3 highlights some of their differences, and the reasons for those differences. The next section discusses how a development might include both Z and the refinement calculus, and gives some rules and heuristic guidelines for changing notation. Section 5 contains the development of a small system, using the rules and methods of the previous section. Finally, we give some conclusions and suggest directions for future work.

2 A summary of the notations

In this section, we give very brief summaries of Z and the refinement calculus. For further details on Z, the reader is referred to [3], [9] and [10]. Information on the refinement calculus can be found in the work of Back [1], Morris [8] and Morgan [6,7]. We will use the notation of Morgan.

2.1 Z

Z is based on typed set theory, together with a structuring mechanism: the schema. This is a device for introducing a named collection of variables and giving a predicate to show how they are related. It can be used to describe both the static aspects of a system (i.e., the state space, and invariant relations on the state) and the dynamic aspects (i.e., the operations which change the state). For instance,

Class

$$y, n : \mathbb{P}\ Student$$

$$y \cap n = \{\}$$
$$\#(y \cup n) \leq max$$

describes the state of a system to record which students in a class have (y) or have not (n) completed a set of exercises. The declaration part, above the line, introduces the state variables, and the predicate part, below the line, gives the state invariant. An operation to enrol a new student into the class could be described by

Enrol_ok

$\Delta Class$
$s? : Student$

$s? \notin y \cup n$
$\#(y \cup n) < max$
$y' = y$
$n' = n \cup \{s?\}$

Now the declaration introduces the state before and after the operation (in $\Delta Class$) and the input variable $s?$. The predicate shows the relation between the variables of the state before (y and n) and the state after (y' and n') and $s?$. This single predicate (there are implicit conjunctions between the lines) contains both pre- and post-condition information. (This example is more fully developed in section 5.)

2.2 The refinement calculus

The basis of the refinement calculus is Dijkstra's language of guarded commands[2], together with an additional construct. So the notation has the usual (executable) elements:

- sequential composition
- assignment
- alternation
- iteration

The extra (non-executable) construct is the *specification statement*, which takes the form

w: [*pre* , *post*]

If it could be executed, this statement would have the following effect on a computer:

> If the initial state is described by *pre*, then, by changing only the variables listed in the frame w, establish some final state described by the postcondition *post*.

For example, if s is a set of numbers, the specification statement

$$y\colon [s \neq \{\} \ , \ y \in s]$$

chooses an element y from the non-empty set s.

An important aspect of the refinement calculus approach is to view specifications and code as different parts of a spectrum. They are both examples of the same notation—programs. Some programs happen to be executable (those without specification statements), while others are not. The development process consists of writing a specification, or perhaps an abstract program, then applying any of a large collection of laws to transform the program into executable code. For instance, we could apply the law of strengthening postconditions:

$$\text{if } post' \Rightarrow post, \text{ then } w\colon [pre \ , \ post] \sqsubseteq w\colon [pre \ , \ post']$$

where $\sqsubseteq$ is the refinement relation. Taking $post'$ to be $y = max(s)$, we can see that the specification statement above is refined by

$$y\colon [s \neq \{\} \ , \ y = max(s)]$$

3 Notational differences

3.1 Schemas and specification statements

The valid states of a system are described in very similar ways in the two notations. The refinement calculus uses the keywords **var** and **and** to introduce the state variables and invariants on them. So the refinement calculus formulation of the state schema *Class* given above would be

$$\begin{array}{l} \textbf{var } y, n : \mathbb{P}\, Student \ ; \\ \textbf{and } \ y \cap n = \{\} \wedge \\ \qquad \#(y \cup n) \leq max \end{array}$$

We can see a very obvious correspondence between the two notations.

However, the other main use of schemas, to describe operations on the state, has a significantly different counterpart in the refinement calculus. In Z, we usually describe an operation with a schema of the form

Op

ΔS
$i? : I$
$o! : O$

---

pred

This schema describes a change in the state S, with inputs $i?$ and outputs $o!$. The exact relation between the variables of the before and after states and $i?$ and $o!$ is given by *pred*.

The corresponding structure in the refinement calculus is a specification statement of the form

$$w: [pre\ ,\ post]$$

There are several differences that we can notice between these two constructs, perhaps the most obvious being that the specification statement contains two predicates, while the schema has only one. In specification work, it is convenient to use a single predicate on the before and after states, which embodies both the pre- and the postconditions. It is the fact that we have this one predicate which allows us to use the powerful specification combinators of the schema calculus, e.g. schema conjunction and disjunction. It is always possible to obtain the precondition from this single predicate by existential quantification over the outputs and the after state variables (this is exactly what the *pre* schema operator does). Indeed it is often recommended that this precondition checking should be carried out and recorded as part of the specification process (see [11]). However, when we move on to program development, it is much more convenient to have the precondition explicitly available: in order to prove that $P \sqsubseteq Q$, we need to show (with appropriate variable changes) that

$$\begin{aligned} &\text{pre } P \Rightarrow \text{pre } Q \\ &\text{pre } P \wedge Q \Rightarrow P \end{aligned}$$

Clearly it is easier to discharge these proof obligations if the preconditions of P and Q are readily to hand. Indeed, if we are carrying out a development with several levels of design, then these proofs will need to be carried out at each level. We can therefore save ourselves much effort by working with *pre*/*post* pairs, rather than a single combined predicate, with the need to calculate the new precondition at each level.

The second major difference between the schema and the specification statement $w: [pre\ ,\ post]$ is the frame w. This is a list of the state variables which can be changed by any implementation of this specification statement. Again, one of the reasons why the Z schema calculus is good for writing comprehensible specifications is that it allows us to structure our description. We can describe parts of the state of a large system and the effect of the operations on those parts of the state, before combining them, with schema conjunction and disjunction, to form descriptions of the whole state and the operations on it. However, during the refinement process, we don't have the option of using conjunction for structuring. The problem is made worse because, as we approach executable code, the program becomes inevitably more complicated and many more variables are needed. However, only a few of the variables will be changed in a particular part of the program, and so the use of the frame simplifies the postcondition by relieving us of the obligation of writing $x' = x$ for each unchanged variable.

3.2 Initial variables

In Z we distinguish between variables of the before state and the after state of an operation by adding a dash to the name of the after state variables. Thus x might refer to the value of a variable before an operation, while x' would refer to its value afterwards. In the refinement calculus, the distinction is made, in the postcondition, by adding a subscript 0 to the names of the *before* state variables. So y_0 would be the variable before an operation, and y afterwards. This has both advantages and disadvantages: the chief advantage is that we have less to write! The precondition will always refer to initial variables, so we don't need to give them any decoration to show that they are initial. The postcondition, on the other hand, will almost always refer to final variables, and only sometimes to initial variables. So we decorate the variables that appear less frequently: the initial variables. A side effect of this convention is a very simple sequential composition law:

$$w{:}\ [pre\ ,\ post] \sqsubseteq w{:}\ [pre\ ,\ mid]\ ;\ w{:}\ [mid\ ,\ post]$$

for any formula *mid* not containing any initial variables. The same law, expressed with the undashed/dashed convention, would be

$$w{:}\ [pre\ ,\ post] \sqsubseteq w{:}\ [pre\ ,\ mid]\ ;\ w{:}\ [mid[\_/\_']\ ,\ post]$$

with an inelegant renaming[1] of the variables of the *mid* predicate.

The disadvantage of this naming convention is a loss of referential transparency: when we refer to a variable x in the specification statement $w{:}\ [pre\ ,\ post]$, we need to know whether the reference is in *pre* or *post* to decide whether we are talking about the value of x before or after the operation. Newcomers to the notation can find this a bit disconcerting, but familiarity works wonders!

It is interesting to see the parallel with the history of VDM—in [4], the dashed/undashed convention is used, while, in [5], the overhook $\overline{x}$ is used for initial variables, because the refinement proof rules are much simpler when expressed with this convention.

3.3 Short variable names

One of the conventions of the refinement calculus is to use short names for state variables—preferably either one or two letters. These variables are copied around a lot, so this reduces the chances of errors in transcription. The loss of immediate clarity in the program is justified by saying that the program text is no longer what needs to be understood. Instead it is the development history which needs to be clear. The collection of the final code from the leaves of the development tree could easily be automated—the developer does not *need* to see it at all.

[1] Throughout this paper, we use the substitution notation $[new/old]$. Thus $mid[\_/\_']$ denotes the predicate *mid* with all dashed variables replaced by undashed ones.

3.4 Abbreviations

While Z uses the names of schemas as (amongst other things) abbreviations for the predicates therein, and syntactic definitions to give names to expressions, the refinement calculus uses one mechanism (very similar to syntactic definitions) to give names to both predicates and expressions. Use of this mechanism is very important as it can prevent statements from becoming too long and unwieldy.

4 Using Z and the refinement calculus together

In the previous section, we listed some of the differences in notation and the refinement calculus, and we gave some of the reasons why the designers of the newer notation, the refinement calculus, have felt it necessary to make these changes. The differences in notation fall into two distinct categories: some are basically insignificant—the choice between decorating variables of the before state or the after state in Z could be justified either way. Some might argue that the Z convention should change to harmonise with the refinement calculus notation, where there is a good justification for decoration of the before state variables[2]. However, other changes are far more significant—the use of *pre/post* pairs of predicates and the frame have an important effect on the way in which we can develop code from Z specifications. Ideally, we would have a wide spectrum language like the refinement calculus which also allowed us the specification combinators that we use in Z to construct large specifications from smaller ones. Unfortunately, we don't have that language. But we do have a reasonable number of people who have some expertise in the use of Z for specification. It is the purpose of this section to show how we can use Z and the refinement calculus together, in reasonable harmony, in a single development.

4.1 The basic rule for change of notation

Our plan for development is the following:

- specification in Z
- data refinement in Z
- change notation to the refinement calculus notation
- algorithm refinement using the refinement calculus

[2]We are more than happy to leave such discussions to their proper place—the standards committees.

We are assuming, of course, that the system to be developed is of sufficient size and complexity to benefit from the use of the schema notation for structuring the specification. If the system is small enough to be specified directly in the refinement calculus, then clearly that is the best method of working. There are many examples in [6] of such developments.

So our first step is to write a specification in Z, using as much of the schema calculus as is necessary to give a comprehensible description of the system. (Detailed advice may be found in any good book on Z—see [3,10,12,13].)

The second stage is to carry out the data refinement, still in the world of schemas. So we propose a concrete state and a retrieve relation which relates the abstract and concrete, and then we propose, and prove correct, the initialisation and the concrete versions of the operations. It is at this stage that our design is probably beginning to become unwieldy—the concrete state will have introduced the more complicated data structures that we are likely to implement, and our descriptions of the operations are correspondingly more complex.

It is now that we switch notations. The first, simple change is to convert from the undashed/dashed convention to the use of a subscript 0, and to shorten the names of variables, if necessary, also removing the ? and ! suffices from the names of input and output variables. If the state and a typical operation on it were described by schemas S and Op, let us denote the 'translated' versions by $\overline{S}$ and $\overline{Op}$. From $\overline{S}$, it is simple to extract the declarations and the state invariants, and record them in the **var** and **and** clauses respectively.

Now we have to consider the translation from schemas to specification statements. There is one basic law for this, which is based on an Implicit Precondition abbreviation which appeared in an early draft of [6]. Suppose we have a schema describing an operation of the form

$\overline{Op}$

$\Delta\overline{S}$

$i? : I$

$o! : O$

$pred$

Then we can convert this to the specification statement

$$w\colon [(\exists\, w : T \mid inv \bullet pred)[w/w_0]\ ,\ pred]$$

where

- the frame w consists of the variables of $\overline{S}$, together with the outputs o, and T is the type

- inv is the state invariant, obtained from $\overline{S}$
- the substitution $[w/w_0]$ ensures that the precondition is expressed in terms of undecorated variables.

There are various steps we can immediately take to 'tidy up' this specification. This process involves the systematic application of particular refinement calculus laws. For instance, we can remove from the frame any variables which the postcondition says must stay unchanged. Obviously we also remove the relevant $x = x_0$ clause from the postcondition. We can also remove any clauses from the postcondition which are merely stating the precondition—it is often recommended that a Z schema should contain its preconditions explicitly as conjuncts in the predicate. If this convention is followed, then the precondition clause will appear twice in the final specification statement.

So, for example, the operation *Enrol_ok* could give rise to the following conversion: first we change to short variable names and alter the initial variable convention.

Enrol_ok

$Class_0$
$Class$
$s : Student$

$s \notin y_0 \cup n_0$
$\#(y_0 \cup n_0) < max$
$y = y_0$
$n = n_0 \cup \{s\}$

Using the rule above, and an ahbreviation (E) for the predicate of $\overline{Enrol\_ok}$, this becomes

$$E \mathrel{\widehat=} (s \notin y_0 \cup n_0 \land \#(y_0 \cup n_0) < max \land y = y_0 \land n = n_0 \cup \{s\})$$

$$y, n: \left[\left(\exists\, y, n : \mathbb{P}\ Student \;\middle|\; \begin{array}{l} y \cap n = \{\} \\ \#(y \cup n) \leq max \end{array} \;\bullet E \right) [y, n/y_0, n_0] \;,\; E \right]$$

Simplifying the precondition, this becomes

$$y, n: \left[\left(\begin{array}{c} s \notin y_0 \cup n_0 \\ \#(y_0 \cup n_0) < max \end{array} \right) [y, n/y_0, n_0] \;,\; E \right]$$

$$= y, n: \left[\left(\begin{array}{c} s \notin y \cup n \\ \#(y \cup n) < max \end{array} \right) \;,\; \left(\begin{array}{c} s \notin y_0 \cup n_0 \\ \#(y_0 \cup n_0) < max \\ y = y_0 \\ n = n_0 \cup \{s\} \end{array} \right) \right]$$

Now we can apply the 'tidying up' procedure mentioned above: the first two conjuncts of the postcondition can be removed since they already appear in the precondition, and

we can also remove y from the frame.

$$n{:}\left[\left(\begin{array}{c} s \notin y \cup n \\ \#(y \cup n) < max \end{array}\right),\ n = n_0 \cup \{s\}\right]$$

With practice, it would be possible to write down this specification statement directly, without the need for the intermediate steps.

4.2 More sophisticated rules for change of notation

It would be possible to carry out all of our notation changes using the rule above, and then develop the final program in the refinement calculus notation. However, there are various patterns that often occur in the schema versions of the concrete operations, and we can use these patterns as we carry out the notational change: instead of obtaining a simple specification statement, which is then developed into a program, we go straight to an abstract program which already has some structure. We are encoding the application of several refinement calculus laws into our translation rule, thus saving the developer the need to apply these laws in the cases where she can recognise a pattern in the schema formulation of the concrete operations. Examples of the use of these rules are given in the case study in the next section. Notice that we do not give rules for *every* construct of the guarded command language. For instance, there is no rule for translating a schema description of an operation directly into an iteration: it is very rare that we can see the invariant and the bound function obviously in the schema description. There is also no rule given below for assignment, but it is easy to work out the form of an operation described in the schema notation which would correspond to an assignment in the refinement calculus. However, since the rule is not needed in the case study below, it is omitted for brevity.

Alternation 1

One of the simplest rules is used when we have an operation which is defined as the disjunction of two cases (perhaps the successful case and an error case):

$$Op \triangleq Op1 \vee Op2$$

Suppose also that the preconditions of $Op1$ and $Op2$ are simply expressible in our target programming language. Then we can translate Op to an alternation:

if pre $Op1 \rightarrow Op1^*$
[] pre $Op2 \rightarrow Op2^*$
fi

where $Op1^*$ and $Op2^*$ are the specification statements which result from the application of the basic translation rule above.

Alternation 2

The transformation above stipulated that the preconditions of $Op1$ and $Op2$ should be simply expressible in the target programming language. What happens when the preconditions are more complex? Perhaps they involve a quantifier, or a complicated set expression. In this case, we can introduce a local variable, and use it to store the result of evaluating one of the preconditions before the alternation. Suppose we have

$$Op \mathrel{\widehat{=}} Op1 \lor Op2$$

and the precondition of $Op1$ is a complex expression. Then we can translate this to

$$\begin{array}{l} [\![\ \textbf{var}\ b : \mathit{Boolean} \bullet \\ \quad b{:}\ [\mathit{true}\ ,\ b \Leftrightarrow \text{pre}\ Op1]\,; \\ \quad \textbf{if}\ b \rightarrow Op1^{\bullet} \\ \quad [\!]\ \text{pre}\ Op2 \rightarrow Op2^{\bullet} \\ \quad \textbf{fi} \\ ]\!] \end{array}$$

where b is some fresh variable with scope delimited by $[\![$ and $]\!]$, and $Op1^{\bullet}$ and $Op2^{\bullet}$ are as above. Clearly, if pre $Op2 = \neg$pre $Op1$, then we can simplify the second guard to $\neg b$.

Alternation 3

The most general version of the alternation rule allows us to evaluate any expression before the alternation, and to store the result in a fresh variable of any type. With Op as above, the refinement calculus version becomes

$$\begin{array}{l} [\![\ \textbf{var}\ r : T \bullet \\ \quad r{:}\ [\mathit{true}\ ,\ \phi]\,; \\ \quad \textbf{if}\ \psi_1 \rightarrow w{:}\ \left[\phi \land \psi_1\ ,\ \overline{Op1}\right] \\ \quad [\!]\ \psi_2 \rightarrow w{:}\ \left[\phi \land \psi_2\ ,\ \overline{Op2}\right] \\ \quad \textbf{fi} \\ ]\!] \end{array}$$

where ϕ, ψ_1 and ψ_2 are any predicates, which satisfy the side conditions

1. $\phi \land (\overline{\text{pre}\ Op1} \lor \overline{\text{pre}\ Op2}) \Rightarrow (\psi_1 \lor \psi_2)$
2. $\phi \land (\overline{\text{pre}\ Op1} \lor \overline{\text{pre}\ Op2}) \Rightarrow (\psi_i \Rightarrow \overline{\text{pre}\ Opi})$ for $i = 1, 2$

Notice that if pre $Op1 = \neg$pre $Op2$, the antecedents above simplify to ϕ, leaving

1'. $\phi \Rightarrow (\psi_1 \vee \psi_2)$

2'. $\phi \Rightarrow (\psi_i \Rightarrow \overline{\text{pre } Op_i})$ for $i = 1, 2$

Sequential composition

The final way in which we can use the structure of the Z specification to help us with the structure of the refinement calculus program is when we notice that the operation is described as the conjunction of two other operations which act on disjoint parts of the state. Suppose we have

$$Op \mathrel{\widehat=} Op1 \wedge Op2$$

where $Op1$ and $Op2$ take the forms

$$Op1 \mathrel{\widehat=} [\Delta S \mid s_1' = s_1 \wedge P1(s_2, s_2')]$$
$$Op2 \mathrel{\widehat=} [\Delta S \mid s_2' = s_2 \wedge P2(s_1, s_1')]$$

where s_1 and s_2 are disjoint (vectors of) state variables, and $P1$ and $P2$ are predicates showing how part of the state is altered. Then we have two possibilities: we can either update first s_1 and then s_2, or vice versa. So Op becomes either

$$s_1 : \left[\overline{\text{pre } Op2}\ ,\ P2\right];$$
$$s_2 : \left[\overline{\text{pre } Op1}\ ,\ P1\right]$$

or

$$s_2 : \left[\overline{\text{pre } Op1}\ ,\ P1\right];$$
$$s_1 : \left[\overline{\text{pre } Op2}\ ,\ P2\right]$$

In this section, we have shown various rules and heuristics for translating from the schemas of our data refined operations into the refinement calculus notation of abstract programs. First we showed the basic rule which simply extracts the precondition from the combined predicate of the Z schema. Then we gave some rules which take advantage of the structure which has been built up within the Z design, and use that structure in the refinement calculus program. We have given four such rules—there are probably several other patterns of design that could be exploited in a similar way.

5 Case study

The example we have chosen is a fairly simple one, which will probably be familiar to both VDM and Z users. The following statement of requirements comes from [4].

A computerised class manager's assistant is required to keep track of students enrolled on a class, and to record which of them have done the midweek exercises. When a student applies for a class, he will be enrolled on it, unless it is full. Such a student will be presumed not to have done the exercises. When a student completes the exercises, the fact is to be recorded. A student may leave a class even if he has not done the exercises, but only the students who have done the exercises are entitled to a completion certificate.

Specification[3]

We need just one given set, to identify individual students:

$[Student]$

The maximum size of a class is a global constant.

$$max : \mathbb{N}$$
$$max > 0$$

Our abstract state consists of a pair of disjoint sets: the set y represents the students who have done the exercises, and n is those who have enrolled, but not yet done the exercises.

Class
$$y, n : \mathbb{P}\ Student$$
$$y \cap n = \{\}$$
$$\#(y \cup n) \leq max$$

Both sets are empty in the initial state.

Init
$$Class'$$
$$y' = \{\}$$
$$n' = \{\}$$

This clearly satisfies the constraint that y' and n' shall be disjoint, and that the size of their union shall be less than or equal to max.

We will describe only two of the required operations. The third, describing what happens when a student leaves the class, is left as an exercise for the reader. The first operation

[3]This Z specification is based on one given by John Wordsworth in [13].

Operation	Inputs	Precondition
Enrol_ok	$s?$	$s? \notin y \cup n$
		$\#(y \cup n) < max$
Compl_ok	$s?$	$s? \in n$
Found	$s?$	$s? \in y \cup n$
Full		$\#(y \cup n) = max$
Missing	$s?$	$s? \notin n$

Figure 1: Preconditions of the abstract operations

describes the effect of enrolling a student in the class. This student will not have done the exercises, so he will be put in the set n.

Enrol_ok

$\Delta Class$
$s? : Student$

$s? \notin y \cup n$
$\#(y \cup n) < max$
$y' = y$
$n' = n \cup \{s?\}$

The second operation specifies the effect of a student completing the exercises. The student will have been enrolled, but will not have done the exercises, so he will be transferred from n to y.

Compl_ok

$\Delta Class$
$s? : Student$

$s? \in n$
$y' = y \cup \{s?\}$
$n' = n \setminus \{s?\}$

The preconditions for the partial operations so far defined, together with those for the error schemas below, are summarised in the table in Figure 1.

We introduce a new type for the error reports.

$$Response ::= ok \mid found \mid full \mid missing$$

The error situations for *Enrol_ok* are described by the following schemas.

Found

$\Xi Class$
$s? : Student$
$resp! : Response$

$s? \in y \cup n$
$resp! = found$

Full

$\Xi Class$
$resp! : Response$

$\#(y \cup n) = max$
$resp! = full$

Similarly the operation *Compl_ok* has an error situation dealt with by

Missing

$\Xi Class$
$s? : Student$
$resp! : Response$

$s? \notin n$
$resp! = missing$

We also need to give a report in the successful cases:

Success

$resp! : Response$

$resp! = ok$

The complete description of the operations can now be given:

$Enrol \triangleq (Enrol\_ok \wedge Success) \vee Full \vee Found$

$Complete \triangleq (Compl\_ok \wedge Success) \vee Missing$

5.1 Data refinement

Our concrete representation for the sets given in the specification above will consist of two arrays, one for students, and one for boolean values, and a counter to say how much

of the arrays is in use. It is intended that the values in the second array will be *true* for those who have done the exercises, and *false* for those who have not. We model the arrays by total functions whose domain is the index set $(1 \,..\, max)$.

Class_1

$$cl : 1 \,..\, max \rightarrow Student$$
$$ex : 1 \,..\, max \rightarrow Boolean$$
$$num : 0 \,..\, max$$

$$((1 \,..\, num) \lhd cl) \in (\mathbb{N} \rightarrowtail Student)$$

The concrete state invariant says that there will be no duplicates in the first *num* elements of the array of students[4].

The retrieve relation, relating the concrete and abstract states, is as follows:

Retr

$$Class$$
$$Class\_1$$

$$y = \{i : 1 \,..\, num \mid (ex\ i) = true \bullet (cl\ i)\}$$
$$n = \{i : 1 \,..\, num \mid (ex\ i) = false \bullet (cl\ i)\}$$

Initially, no part of either of the arrays is in use:

Init_1

$$Class\_1'$$

$$num' = 0$$

Our initialisation proof obligation is to show that

$$Init\_1 \vdash (\exists\ Class' \bullet (Init \wedge Retr'))$$

This is easily shown to be true, since the value of num', determined by *Init_1* to be zero, gives the empty set for y' and n' when substituted in the right hand sides of the expressions in $Retr'$. These values for y' and n' satisfy *Init*.

The successful part of the *Enrol* operation is represented by the following concrete operation.

[4] $x \,..\, y$ is the set $\{i : \mathbb{N} \bullet x \leq i \leq y\}$

Enrol_ok_1

$\Delta Class\_1$

$s? : Student$

$s? \notin \{i : 1 .. num \bullet cl\ i\}$

$num < max$

$cl' = cl \oplus \{num' \mapsto s?\}$

$ex' = ex \oplus \{num' \mapsto false\}$

$num' = num+1$

Other schemas are required to make the operation correspond to the abstract version:

Full_1

$\Xi Class\_1$

$resp! : Response$

$num = max$

$resp! = full$

Found_1

$\Xi Class\_1$

$s? : Student$

$resp! : Response$

$\exists i : 1 .. num \bullet cl\ i = s?$

$resp! = found$

Putting these together, we can give the robust concrete version of *Enrol*:

$$Enrol\_1 \triangleq (Enrol\_ok\_1 \wedge Success) \vee Full\_1 \vee Found\_1$$

Similarly, we can describe the concrete versions of the different parts of the *Complete* operation. The successful part is represented by

Compl_ok_1

$\Delta Class\_1$

$s? : Student$

$\exists i : 1 .. num \bullet (cl\ i = s? \wedge ex\ i = false \wedge$
$cl' = cl \wedge$
$cx' = cx \oplus \{i \mapsto true\}$
$num' = num)$

Operation	Inputs	Precondition
$Enrol\_ok\_1$	$s?$	$s? \notin \{i : 1 .. num \bullet cl\ i\}$
		$num < max$
$Compl\_ok\_1$	$s?$	$(\exists i : 1 .. num \bullet cl\ i = s? \land ex\ i = false)$
$Found\_1$	$s?$	$(\exists i : 1 .. num \bullet cl\ i = s?)$
$Full\_1$		$num = max$
$Missing\_1$	$s?$	$(\forall i : 1 .. num \bullet cl\ i \neq s? \lor ex\ i = true)$

Figure 2: Preconditions of the concrete operations

The error part of *Complete* is given by

$$
\begin{array}{|l}
\hline
Missing\_1 \\
\hline
\Xi Class\_1 \\
s? : Student \\
resp! : Response \\
\hline
\forall i : 1 .. num \bullet cl\ i \neq s? \lor ex\ i = true \\
resp! = missing \\
\hline
\end{array}
$$

So the total interface for *Complete* is

$$Complete\_1 \mathrel{\widehat=} (Compl\_ok\_1 \land Success) \lor Missing\_1$$

A summary of the preconditions for the concrete operations is given in Figure 2. We can easily verify that the operations we have given are indeed total.

We now need to verify that the concrete versions of *Enrol* and *Complete* that we have given are genuine refinements of the abstract operations. The theorems we have to prove are

$$(\text{pre } Enrol) \land Retr \vdash \text{pre } Enrol\_1$$
$$(\text{pre } Enrol) \land Retr \land Enrol\_1 \vdash (\exists Class\_1 \bullet Enrol \land Retr')$$
$$(\text{pre } Complete) \land Retr \vdash \text{pre } Complete\_1$$
$$(\text{pre } Complete) \land Retr \land Complete\_1 \vdash (\exists Class\_1 \bullet Complete \land Retr')$$

These proofs are omitted for the sake of brevity, but they are not complex, since we can verify immediately that both concrete and abstract versions are total, and the *Retrieve* relation is functional.

5.2 Notational change and algorithm refinement

The first part of the notational change is to obtain the declarations and state invariant from *Class_1*.

$$\begin{array}{l}\textbf{var } cl : 1\,..\,max \rightarrow Student;\\ \quad ex : 1\,..\,max \rightarrow Boolean;\\ \quad num : 0\,..\,max;\\ \textbf{and } \ ((1\,..\,num) \lhd cl) \in \mathbb{N} \rightarrowtail Student\end{array}$$

We can also easily get the initialisation from *Init_1*

$$\textbf{initially } num = 0$$

We will use the Alternation 3 translation rule for both operations. In each case, the specification statement that we insert before the alternation will check to see whether s occurs in the active part of the cl array. If it is there, then its position will be recorded in the new variable w; if not, w will be set to $num{+}1$. So the specification statement will establish the following postcondition

$$(w \in 1\,..\,num \wedge cl[w] = s) \vee (w = num{+}1 \wedge s \notin cl[1\,..\,num])$$

where $cl[1\,..\,num]$ denotes the set containing the first num elements of the array cl. Let us call this predicate H.

The three branches of the alternation correspond to the three disjoined schemas in the definition of *Enrol_1*. We have also removed cl, ex and num from the frame in the 'error' branches. So we have

Enrol (**value** s : *Student*; **result** r : *Response*)

$$\begin{array}{ll}
\sqsubseteq & \textbf{var } w : 0\,..\,max{+}1 \bullet \\
& w{:}\ [\mathit{true}\ ,\ H]; & (1)\\
& \textbf{if } w = num{+}1 \wedge num < max \rightarrow \\
& \qquad r, cl, ex, num{:}\ \left[\begin{pmatrix} H \\ w = num{+}1 \\ num < max \end{pmatrix}\ ,\ \begin{pmatrix} cl = cl_0 \oplus \{num \mapsto s\} \\ ex = ex_0 \oplus \{num \mapsto false\} \\ num = num_0{+}1 \\ r = ok \end{pmatrix} \right] & (2)\\
& [\!] \ num = max \rightarrow \\
& \qquad r{:}\ [H \wedge num = max\ ,\ r = full] & (3)\\
& [\!] \ w \in 1\,..\,num \rightarrow \\
& \qquad r{:}\ [H \wedge w \in 1\,..\,num\ ,\ r = found] & (4)\\
& \textbf{fi}
\end{array}$$

We need first to check the provisos of translation rule Alternation 3. We note that the concrete operation *Enrol_1* is total, so we have to prove 1′ and 2′. In this case, 1′ becomes

$$H \Rightarrow (w = num{+}1 \wedge num < max) \vee num = max \vee w \in 1 \,..\, num$$

which is obviously true.

For the first branch, 2′ is

$$H \Rightarrow ((w = num{+}1 \wedge num < max) \Rightarrow \overline{\text{pre } Enrol\_ok\_1})$$

Since we have $w = num{+}1$, the second disjunct of H must hold, and so $s \notin cl[1 \,..\, num]$. But

$$\overline{\text{pre } Enrol\_ok\_1} = (s \notin \{i : 1 \,..\, num \bullet cl\ i\} \wedge num < max)$$

and so we are finished.

The second branch is trivial, since the new guard $num = max$ is exactly the same as $\overline{\text{pre } Full\_1}$.

For the third branch, we need to show

$$H \Rightarrow (w \in 1 \,..\, num \Rightarrow \exists i : 1 \,..\, num \bullet cl[i] = s)$$

From $w \in 1 \,..\, num$, we can conclude that the first disjunct of H holds. Therefore $cl[w] = s$. So we take $i = w$ in the RHS.

So the provisos for Alternation 3 are satisfied.

Each of the three branches of the alternation can be implemented with an assignment. The justification, involving substituting the new value for the old in the postconditions, is omitted here[5]. In the case of (2), the development should include a check that the state invariant is maintained—this is where the precondition of (2) is used.

(2) $\sqsubseteq\ r, cl[num{+}1], ex[num{+}1], num := ok, s, false, num{+}1$

$\sqsubseteq\ r := ok;$
$num := num{+}1;$
$cl[num] := s;$
$ex[num] := false$

(3) $\sqsubseteq\ r := full$

(4) $\sqsubseteq\ r := found$

[5]A summary of a few of the laws of the refinement calculus is given as an appendix.

We will refine (1) to an iteration which sets the value of w. To do this, we re-phrase H to introduce the invariant:

$(1) = \; w\colon [\mathit{true}\ ,\ H]$

$= \; I \mathrel{\widehat{=}} w \leq num{+}1 \wedge s \notin \mathrm{ran}(1\,..\,w{-}1) \lhd cl \bullet$
$\quad w\colon [\mathit{true}\ ,\ I \wedge (w = num{+}1 \vee cl[w] = s)]$

$\sqsubseteq$ "sequential composition"
$\quad w\colon [\mathit{true}\ ,\ I]\,;$ (5)
$\quad w\colon [I\ ,\ I \wedge (w = num{+}1 \vee cl[w] = s)]$ (6)

The initialisation is trivial:

$(5) \sqsubseteq v := 1$

Now we can introduce the iteration itself

$(6) \sqsubseteq$ "**invariant** I, **variant** $num{+}1{-}w$ "
do $w \neq num{+}1 \wedge cl[w] \neq s \rightarrow$

$$w\colon \left[\begin{pmatrix} I \\ w \neq num{+}1 \\ cl[w] \neq s \end{pmatrix} \ ,\ \begin{pmatrix} I \\ 0 \leq num{+}1{-}w < num{+}1{-}w_0 \end{pmatrix} \right] \qquad (7)$$

od

The body of the loop can be implemented with a simple assignment. (The justification is left as an exercise for the reader!)

$(7) \sqsubseteq w := w{+}1$

The development of the code for the *Complete* operation is very similar to that for the *Enrol* operation above. Once again, we use rule Alternation 3, with exactly the same specification statement before the alternation.

Complete (**value** s : *Student*; **result** r : *Response*)

$\sqsubseteq$ **var** $w : 0\,..\,max{+}1$ •

w: $[true\ ,\ H]$; (8)

if $w \in 1\,..\,num \wedge ex[w] = false \rightarrow$

$$r, ex: \left[\begin{pmatrix} H \\ w \in 1\,..\,num \\ ex[w] = false \end{pmatrix}\ ,\ \begin{pmatrix} r = ok \\ ex = ex_0 \oplus \{w \mapsto true\} \end{pmatrix}\right] \quad (9)$$

[] $w = num{+}1 \vee ex[w] = true \rightarrow$

r: $[H \wedge (w = num{+}1 \vee ex[w] = true)\ ,\ r = missing]$ (10)

fi

The provisos for using rule Alternation 3 are discharged as above, and the branches of the alternation become assignments:

(9) $\sqsubseteq$ $r, ex[w] := ok, true$

(10) $\sqsubseteq$ $r := missing$

The loop development is identical to the one above:

(8) $\sqsubseteq$ $w := 1$;
do $w \neq num{+}1 \wedge cl[w] \neq s \rightarrow$
$w := w{+}1$
od

Our final act in the development is to collect up all the code for the two operations. This is given in Figure 3.

6 Conclusions and directions for future work

We have shown in this paper how two different notations can be used in reasonable harmony within a single development. Our aim has been to use each of the notations for the part of the development cycle for which it is best suited: we used Z in the specification and design stages, where we can use the schema calculus to structure work, to introduce the complexity in manageable pieces and to put those pieces together to give the whole picture. Then we used some of that structure to help us with our first program in the refinement calculus notation. Finally we used the laws of the refinement calculus to develop our abstract programs into executable code.

```
Enrol (value s : Student; result r : Response) ≙
|[ var w : 0 .. max+1 •
   w := 1;
   do   w ≠ num+1 ∧ cl[w] ≠ s →
            w := w+1
   od   ;
   if w = num+1 ∧ num < max →
            r := ok;
            num := num+1;
            cl[num] := s;
            ex[num] := false
   [] num = max →
            r := full
   [] w ∈ 1 .. num →
            r := found
   fi
]|

Complete (value s : Student; result r : Response) ≙
|[ var w : 0 .. max+1 •
   w := 1;
   do   w ≠ num+1 ∧ cl[w] ≠ s →
            w := w+1
   od   ;
   if w ∈ 1 .. num ∧ ex[w] = false →
            r := ok;
            ex[w] := true
   [] w = num+1 ∨ ex[w] = true →
            r := missing
   fi
]|
```

Figure 3: Code for the two operations

As was mentioned above, one direction for future work might be to develop further rules and heuristics for the notational change—it would be interesting to see, for instance, what sort of programs correspond to Z specifications which use the technique of promotion.

When more case studies have been completed, it may be possible, as was hinted above, to recommend that the Z notation should be changed in various ways to make the development path smoother. These would probably be small syntactic changes. Further case studies may also give some ideas for enhancements to the refinement calculus notation, particularly in structuring and the use of modules.

A final interesting point of research would be to investigate the point in the lifecycle at which the notational change should take place. We have advocated changing after the data refinement, but another possibility would be to change immediately after the specification, and to use the auxiliary variable techniques of [6] for data refinement, which would allow us to 'mix up' algorithm refinement and data refinement.

Acknowledgments

Thanks to Carroll Morgan and Trevor Vickers for helpful comments at very short notice! Also thanks to the VDM90 referees for their input.

A Selected laws of the refinement calculus

We give here a selection of the laws of the refinement calculus, which are used in the case study in section 5 of the paper.

1. *Weakening the precondition:* If $pre \Rightarrow pre'$ then

   $$w: [pre\ ,\ post] \sqsubseteq w: [pre'\ ,\ post]$$

2. *Strengthening the postcondition:* If $post' \Rightarrow post$ then

   $$w: [pre\ ,\ post] \sqsubseteq w: [pre\ ,\ post']$$

3. *Introducing local variables:* If x does not appear free in pre or $post$, then

   $$w: [pre\ ,\ post] = |[\ \mathbf{var}\ x : T \bullet w, x: [pre\ ,\ post]\ ]|$$

4. *Introducing assignment:* If E is an expression, then

$$w{:}[post[E/w]\ ,\ post] \sqsubseteq w := E$$

5. *Introducing sequential composition:*

$$w{:}[pre\ ,\ post] \sqsubseteq w{:}[pre\ ,\ mid]\ ;\ w{:}[mid\ ,\ post]$$

6. *Introducing alternation:*

$$\begin{aligned} & w{:}[pre \wedge (\bigvee i \bullet G_i)\ ,\ post] \\ = \ & \textbf{if}([]\ i \bullet G_i \rightarrow w{:}[pre \wedge G_i\ ,\ post])\ \textbf{fi} \end{aligned}$$

7. *Introducing iteration:*

$$\begin{aligned} & w{:}[inv\ ,\ inv \wedge \neg(\bigvee i \bullet G_i)] \\ \sqsubseteq \ & \textbf{do} \\ & \quad ([]\ i \bullet G_i \rightarrow w{:}[inv \wedge G_i\ ,\ inv \wedge (0 \leq V < V_0)]) \\ & \textbf{od} \end{aligned}$$

The predicate inv is the invariant and the expression V is the integer-valued variant.

References

[1] R.-J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593-624, 1988.

[2] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[3] I.J. Hayes, editor. *Specification case studies.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1987.

[4] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1980.

[5] C.B. Jones. *Systematic software development using VDM.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1986.

[6] C.C. Morgan. *Programming from Specifications.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1990.

[7] C.C. Morgan, K.A. Robinson, and P.H.B. Gardiner. On the refinement calculus. Technical Report PRG-70, Programming Research Group, 1988.

[8] J.M. Morris. Programs from specifications. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs.* Addison-Wesley, 1989.

[9] J.M. Spivey. *Understanding Z: a specification language and its formal semantics.* Number 3 in Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, 1988.

[10] J.M. Spivey. *The Z notation: a reference manual.* Prentice-Hall International series in computer science / C.A.R. Hoare, series editor. Prentice-Hall International, Englewood Cliffs, N.J. ; London, 1989.

[11] J.C.P. Woodcock. Calculating properties of z specifications. *ACM SigSoft Software Engineering Notes*, 15(4):43–64, 1989.

[12] J.C.P. Woodcock. Using Z—Specification, Refinement and Proof. Oxford University Computing Laboratory, 1990.

[13] J.B. Wordsworth. A Z Development Method. IBM UK Laboratories Ltd, Hursley Park, 1989.