Structuring specifications in Z

by J.C.P. Woodcock

In the specification notation known as Z, *schemas* are used to structure mathematical descriptions. This paper describes the language of schemas and the conventions that are employed in their use. It also describes how proof obligations are generated during specification, and how these obligations may be discharged. The paper contains many examples, mostly taken from the specification of the user interface to a small, but realistic, software component.

1 Introduction

In the Z notation (Refs. 1–5) there are two complementary languages: the *schema language* supports the structured, systematic presentation of large-scale system specifications written in the *mathematical language*. This paper describes the former and assumes knowledge of the latter.

The schema language allows us to factor out common parts of a specification and to highlight the differences between similar constructs. This helps to reduce typographical errors; it gives us the opportunity to subordinate complexity; and it allows us to re-use existing parts of a specification and to modify them easily. Re-usability is vital to the successful application of formal methods: it allows us to remain flexible by sharing descriptions at every stage of the development process. We are quite used to sharing code — in the form of procedural abstractions in libraries — but here we are talking about specifications sharing parts, proofs sharing arguments, theories sharing abstractions, even problems sharing common aspects.

We believe that the use of schemas helps to promote a good specification style. However, just like any notation, the language of schemas requires careful and judicious application if it is not to be abused. We should not try to use schemas to overcome our shortcomings as specifiers: we should try instead to develop simple theories and to use schemas to present them nicely.

This paper is something of an informal introduction to using schemas; readers interested in a more formal definition of the schema notation are referred to Ref. 3.

1.1 *An introduction to schemas*

The mathematical language of Z (Refs. 4 and 5) is sufficiently powerful to describe many aspects of software systems. However, the application of mathematics to large-scale specification work soon results in unwieldy descriptions that are difficult to follow. It is not the mathematics that is at fault, but rather our human inability to comprehend more than just a small amount of information at a time. Therefore we must present mathematical descriptions in a sympathetic fashion, explaining small parts in the simplest possible context, and then showing how to fit the pieces together to make the whole.

The schema:
One of the most basic things that we can do to help the reader — or indeed the writer — of a specification is to identify commonly used concepts and factor them out from the mathematical description of a system. In this way, we can encapsulate an important concept and give it a name, thus increasing our vocabulary — and our mental power!

In specifications, we see a pattern occurring over and over again: a piece of mathematical text which is a *structure* describing some variables whose values are constrained in some way. We call this introduction of variables under some constraint a *schema.*

Example 1:
The following set comprehension term, lambda expression, and quantified predicate each contain the pattern of constrained variables:

$$\{m, n : \mathbb{N} \mid n = 2 \times m \bullet m \mapsto n\}$$
$$\lambda s : seq[X] \mid s \neq \langle\rangle \bullet (tl\ s) ^\frown \langle hd\ s\rangle$$
$$\forall x, y : \mathbb{N} \mid x \neq y \bullet x > y \lor y > x$$
$\square$

Example 2:
In an operating system there is a program which manages blocks of free storage which users (drawn from the

set U) may want to access.† There are n consecutively numbered blocks:

$$\begin{array}{|l} n : \mathbb{N} \\ B : \mathbb{P}\mathbb{N} \\ \hline B = 1\,..\,n \end{array}$$

The storage manager maintains a directory of which users have which blocks. We call this information structure *dir*, and we would like it to have various properties:

- No block is owned by more than one user.
- A user may own more than one block.
- Some blocks might not be owned at all.
- Some users might not own any blocks.

We can express these properties more formally by considering *dir* as a relation between B and U that satisfies the formalised properties:

- *dir* is *functional.*
- *dir* need not be an *injection.*
- *dir* may be *partial.*
- *dir* need not be a *surjection.*

So it is enough to say that *dir* is a partial function from B to U:

$$dir : B \mathrel{+\!\!\!\!\rightarrow} U$$

We also need a record of which blocks are *free*, i.e. not owned by anyone. This is just a subset of B:

$$free : \mathbb{P}B$$

Of course, it must be that the free blocks are all the ones not being used. We can specify this:

$$free = B \setminus (dom\ dir)$$

We regard this predicate as part of the ***state invariant***: it must always be true. A function $dir : B \mathrel{+\!\!\!\!\rightarrow} U$ and a set $free : \mathbb{P}B$ do not constitute a possible state of the storage manager unless they satisfy the state invariant.§

The state invariant, together with the declarations of *dir* and *free*, form a schema which we shall call SM.

□

A schema consists of two parts: a *declaration* of some variables, and a *predicate* constraining their values: schemas = declarations + predicates.

In Z we often need to distinguish between the declaration of a variable and its underlying *signature.* A declaration is syntactic: it introduces a variable, and says that its values range over some *set*; for example

$$digit : 0\,..\,9$$

† This example also appears in Ref. 6.

§ As we shall see later, the *functionality* of *dir* and the fact that *free* is wholly a subset of B are also part of the state invariant.

We can deduce the type of *digit*, because we know that the numbers $0\,..\,9$ form a subset of $\mathbb{N}$. Thus, *digit* has type $\mathbb{N}$. A signature, on the other hand, is semantic: it introduces a variable and gives its type explicitly. If we had written a constrained signature for *digit*, it would have looked like this:

$$\begin{array}{|l} digit : \mathbb{N} \\ \hline digit \in 0\,..\,9 \end{array}$$

The signature for *dir* does not include its functionality (a constraint which we added to model a requirement), nor does it mention B, since B is simply a subset of $\mathbb{N}$. In fact *dir* is just a relation between numbers and users. Since relations are sets of pairs, the signature would be written

$$dir : \mathbb{P}(\mathbb{N} \times U)$$

In Z there is a convenient short-hand for this:

$$dir : \mathbb{N} \leftrightarrow U$$

How to write a schema:

We can write a schema in one of two forms: either horizontally

$$[declaration \mid predicate]$$

or vertically

$$\begin{array}{|l} \hline declaration \\ \hline predicate \\ \hline \end{array}$$

Example 3:

We can write the storage manager schema as

$$[dir : B \mathrel{+\!\!\!\!\rightarrow} U; free : \mathbb{P}B \mid free = B \setminus (dom\ dir)]$$

or equivalently

$$\begin{array}{|l} \hline dir : B \mathrel{+\!\!\!\!\rightarrow} U; free : \mathbb{P}B \\ \hline free = B \setminus (dom\ dir) \\ \hline \end{array}$$

□

In the horizontal form the signature and predicate are separated by a vertical bar, and in the vertical form by a horizontal bar — both pronounced 'such that'. In the horizontal form, brackets delimit the schema, and in the vertical form, a broken box. If we omit the declaration part, then the schema contains no components; if we omit the predicate part, then it is a short-hand for including the predicate ***true***. In order to reduce unnecessary formal clutter, we often put declarations on separate lines and elide the semi-colon. Similarly, we often put conjuncts on separate lines and elide the conjunction symbol.

Example 4:

The domain of the directory and the *free* set ***partition*** the set of blocks B: together they account for all the blocks, and no block can belong to both sets. Thus, the storage manager schema might also be described as

$$\begin{array}{|l} dir : B \rightarrowtail U; free : \mathbb{P}B \\ \hline free \cup dom\ dir = B \wedge free \cap dom\ dir = \{\} \end{array}$$

or, eliding the semicolon and conjunction symbol, as

$$\begin{array}{|l} dir : B \rightarrowtail U \\ free : \mathbb{P}B \\ \hline free \cup dom\ dir = B \\ free \cap dom\ dir = \{\} \end{array}$$

This schema is *equivalent* to that previously presented, which we can see because the sets (*dom dir*) and ($B \backslash$(*dom dir*)) partition B.

□

Since each line of a predicate forms a conjunct

$$\begin{array}{l} a \Rightarrow b \\ c \\ d \vee e \end{array}$$

means

$$(a \Rightarrow b) \wedge c \wedge (d \vee e)$$

The exception to this is when we impose some additional, obvious structure:

$$\begin{array}{l} y \in B \\ \exists x : X \bullet \\ \quad x < y \vee \\ \quad y < x \end{array}$$

means

$$(y \in B) \wedge (\exists x : X \bullet x < y \vee y < x)$$

It does not matter in which order we write the declarations in a schema: we would be very surprised indeed if the following schema defined something different from the one that we referred to as *SM*:

$$\begin{array}{|l} free : \mathbb{P}B \\ dir : B \rightarrowtail U \\ \hline free = B \backslash (dom\ dir) \end{array}$$

How to name a schema:

Naming a schema introduces a particular kind of syntactic equivalence between name and schema. Schemas are named in the following ways: either horizontally

$$Name \mathrel{\widehat=} [declaration \mid predicate]$$

or by embedding the name in the top line of the vertical schema's box:

$$\begin{array}{|l} \underline{\ Name\ } \\ declaration \\ \hline predicate \end{array}$$

Example 5:

We can name the storage manager schema

$$SM \mathrel{\widehat=} [dir : B \rightarrowtail U; free : \mathbb{P}B \mid free = B \backslash (dom\ dir)]$$

$$\begin{array}{|l} \underline{\ SM\ } \\ dir : B \rightarrowtail U \\ free : \mathbb{P}B \\ \hline free = B \backslash (dom\ dir) \end{array}$$

□

1.2 Schemas within mathematical text

Having encapsulated a concept such as that of the storage manager, we shall probably want to use it in our mathematical descriptions of operating systems. In this section we describe exactly how schemas may appear in mathematics.

Types:

When people use set theory to specify systems, they often include some notion of *type*. For example, whenever we have introduced a new variable, we have been careful to say precisely over which set its values may range. In this sense, we say that we are working in a *typed set theory*. In Z, the notion of type is a very simple one: a type is a *maximal set*: values may belong to just one type. This is not the only possibility: in VDM (Ref. 7), for example, types may be refined by adding a *data type invariant* constraining their values. This process creates a *subtype*. Thus one can imagine a hierarchy of types and subtypes and subsubtypes etc. Type checking then requires the proof of a theorem: that a particular data type invariant has not been violated. However, if one sticks to the simple type system in Z, then type checking is decidable, and a simple compiler-like algorithm may be used.

In the mathematical language there are three kinds of types:

- given set names, for example *B*, *U*, *Password*, *Taskid*
- power set types, for example $\mathbb{P}$ *Char*, $\mathbb{P}B$, $\mathbb{P}(B \times U)$
- Cartesian product types, for example $B \times U$, *Char* × *Password*, *Taskid* × *Password*.

We allow schemas to be used as types as well.

Example 6:

We can introduce managers for different kinds of storage:

$$backingsm, mainsm : SM$$

□

In the example, the two variables each consist of a pair of values related by the predicate in *SM*. However, their type is not simply the Cartesian product

$$(\mathbb{P}(\mathbb{N} \times U)) \times (\mathbb{P}\mathbb{N})$$

since order is unimportant — we did not mind in which order we wrote components in the signature. Instead, we can regard schemas as providing a fourth sort of type: a *schema type* is a product whose elements contain named components. Note that these elements are not constrained by the predicate; i.e. the predicate does not further refine the type. If it did, then we could imagine elements belonging to two or more schema types whose declaration parts were identical, but whose predicate parts were different. This would conflict with our notion of types. However, when we introduce a variable of schema type, then this variable is constrained by the schema's predicate, in the usual way.

Example 7:
Given the declarations

$sm : SM$
$n : 10\,..\,20$

The variable *sm* satisfies the storage manager's state invariant, and n satisfies the predicate $n \in 10\,..\,20$.
□

There is no way in Z to write down the type of the variables *backingsm* and *mainsm*, other than to say they are of type *SMnorm*, where we have defined this to be as unconstrained as possible:

SMnorm
$dir : \mathbb{P}(\mathbb{N} \times U)$
$free : \mathbb{P}$

Clearly *SMnorm* contains a signature which corresponds to the declaration part of *SM*, but with all the predicate information removed. Two schemas, rewritten so that all their predicates are removed, are the same if they differ only in the order in which we wrote down their components.‡ Thus, to see whether two schema objects are of the same type, we must first reduce the schemas over which they range to their underlying signatures. If these contain exactly the same named components, agreeing on their types, then the schema objects come from the same schema type.

There is also no explicit way to write down particular values taken from the type *SMnorm*. Consider first of all an ordinary Cartesian product, say $\mathbb{N} \times \mathbb{N}$. We can form a value from this type by pairing numbers: (x, y), for instance. Here, order is vitally important: we know that x is the first number and y is the second. Both x and y must be natural numbers, and their names must be in scope already: they are not introduced by the pairing operation. In general, a value from the Cartesian product of n-types is called an n-tuple, or simply a tuple, and it can be written down using an ordered list of components, indexed by position.

The values in a schema type are also tuples, but instead of referring to them by position, we refer to them by name, since we cannot use position. The variable *backingsm* denotes just such a tuple, and in the section on lambda abstractions, we shall see a way of referring to the tuple's components. But there is also a way of constructing a schema tuple that is analogous to that of constructing an ordinary tuple. Given the schema definition

InitSM
$dir : B \leftrightarrow U$
$free : \mathbb{P}B$
$dir = \{\}$
$free = B$

the expression

$\theta InitSM$

is a *binding* with $\{\}$ as the value of *dir*, and B as the value of *free*. For this to make sense, the component names *dir* and *free* must already be in scope; they are not introduced by mentioning $\theta InitSM$. As one might expect, the schema tuple $\theta InitSM$ is an element of the schema type underlying *InitSM*, which is the same schema type underlying *SM*: namely, what we have called *SMnorm*.

There is no notation in Z that allows us to write down just what the binding *InitSM* actually is. However, we can give a rule for equality. Remember that two n-tuples a and b are equal just in case the ith element of a is equal to the ith element of b. Similarly, two tuples s and t, from the same schema type, are equal just in case, for each named component in the type, the value in s is the same as the value in t.

Schemas as predicates:

Schemas can be used in any place where a predicate is expected. The component names must already be in scope — with the right types — in the place where the schema is used as a predicate. The schema's signature is discarded, but not any constraints from the declaration.

Example 8:
In our storage allocator system, it is possible that none of the blocks have been allocated to any of the users; it is also possible that all the blocks have been allocated:

$\vdash \exists dir : B \leftrightarrow U; free : \mathbb{P}B \mid free = B \bullet SM$
$\vdash \exists dir : B \leftrightarrow U; free : \mathbb{P}B \mid free = \{\} \bullet SM$

These are equivalent to

$\exists dir : B \leftrightarrow U; free : \mathbb{P}B \mid free = B \bullet free = B \setminus (dom\ dir)$
$\exists dir : B \leftrightarrow U; free : \mathbb{P}B \mid free = \{\} \bullet free = B \setminus (dom\ dir)$
□

When we use a schema in such a way, we must be careful to use it properly. For example, consider the *economical* storage manager which maintains a director, but not a free set:

ESM
$dir : B \leftrightarrow U$

Now, is the following conjecture true, given $b_0 : B$ and distinct $u_0, u_1 : U$?

‡ Later, when we come to the convention that we use to describe sequential systems, we shall describe decorated names, and require that these decorations be removed when exposing the underlying type.

$[dir : \mathbb{P}(\mathbb{N} \times U) \mid dir = \{b_0 \mapsto u_0, b_0 \mapsto u_1\}] \ ?\vdash ESM$

The type of *dir* in *ESM* agrees with the declaration in the hypothesis, and the predicate part of *ESM* is true by definition, and therefore follows from any hypothesis that we choose to make; so the answer would seem to be 'yes'. But the hypothesis introduced a directory that was a relation, but not a partial function, and therefore not a candidate for a possible state of the economical storage manager. The answer to this apparent mystery is simple: we threw the baby out with the bath-water. Instead of disposing of the declaration of *dir*, we should have disposed of the *signature* and retained the declaration's predicate content. If we rewrite *ESM* so that all the predicate information is revealed, and the schema consists of a *signature* and a predicate over that signature

```
┌─ESM──────────────────────
│ dir : ℙ(ℕ × U)
├───────────────
│ dir ∈ B ⇸ U
└──────────────────────────
```

then it is quite clear that

$[dir : \mathbb{P}(\mathbb{N} \times U) \mid dir = \{b_0 \mapsto u_0, b_0 \mapsto u_1\}] \nvdash ESM$

The process of changing declarations into signatures is called *normalisation.*

Set comprehension:

We can use schemas to specify sets in comprehension

$\{schema \bullet term\}$

Example 9:

The set of all possible states of the storage manager is described as

$SMstates \mathrel{\widehat=} \{SM \bullet \theta SM\}$

SMstates is not the same as *SM* because the former contains only those bindings whose values satisfy the *SM*-invariant.†

□

Lambda abstraction:

We can use a schema to provide the signature and constraint for lambda expressions defining functions

$\lambda schema \bullet term$

Such a function requires as its argument an object of the schema type.

Example 10:

The *contiguous* free blocks in our operating system may be calculated by the following function:

```
│ cfb : SM → ℙ(ℙB)
├───────────────
│ cfb = λSM •{l, h : B | l..h ⊆ free ∧
│        (l − 1) ∉ free ∧ (h + 1) ∉ free • l..h}
```

† Other accounts of Z allow a convention that *SM* can be used as a short-hand for $\{SM \bullet \theta SM\}$, so this distinction becomes lost.

Each of the sets in the comprehension term are contiguous: $l..h$ contains every number between l and h, inclusively. Moreover, they are *maximal*, since the set $l..h$ cannot be extended in either direction, since $l - 1$ and $h + 1$ are not free. Because the smallest value that l can assume is 1, $l - 1$ is always defined.

We can apply this function to an instance of a storage management system:

$sm : SM$

```
│ spareblocks : ℙℙB
├───────────────
│ spareblocks = cfb sm
```

Since *sm* denotes a schema tuple, we can apply the function *cfb* to it.

□

We can use this technique of embedding schemas in lambda abstractions to define functions which project out components.

Example 11:

Let us introduce a variable of *SM* type

$sm : SM$

Then, if we often want to refer to the set of free blocks in *sm*, we can name it *fb*. The function $\lambda SM \bullet free$ always returns the free blocks in the storage manager supplied as an argument. Thus, our definition of *fb* could be

```
│ fb : ℙB
├───────────────
│ fb = (λSM • free) sm
```

□

The kind of function which *projects* a component from an object of schema type is useful and frequently used. The construction

$schemaobject \cdot component$

is called *component selection.* It is reminiscent of record field selection in Pascal.

Example 12:

Our storage manager variable has two projections; given $sm : SM$

$sm \cdot dir$ means $(\lambda SM \bullet dir)\ sm$
$sm \cdot free$ means $(\lambda SM \bullet free)\ sm$

□

The components of a schema tuple can be referred to only by using component selection.

Example 13:

The directory component of the initial storage manager state is denoted

$\theta InitSM \cdot dir$

which is, since *dir* must be in scope, simply *dir*.

□

Quantification:

Just as in the preceding section, we can use schemas to supply the signatures and constraints of quantified predicates:

$\exists schema \bullet predicate$
$\forall schema \bullet predicate$

Example 14:

In every storage manager, a block cannot be both free and in use:

$$\vdash \forall SM \bullet \\ \quad \neg(\exists b : B \bullet b \in dom\ dir \wedge b \in free)$$

i.e.

$$\vdash \forall dir : B \leftrightarrow U; free : \mathbb{P}B \mid \\ \quad free = B \setminus (dom\ dir) \bullet \\ \quad\quad \neg(\exists b : B \bullet b \in dom\ dir \wedge b \in free)$$

□

Hypothesis:

Schemas can provide the hypothesis for a theorem:

$schema \vdash conclusion$

Example 15:

Recall our theorem about blocks from the last section:

$$\vdash \forall dir : B \leftrightarrow U; free : \mathbb{P}B \mid \\ \quad free = B \setminus (dom\ dir) \bullet \\ \quad\quad \neg(\exists b : B \bullet b \in dom\ dir \wedge b \in free)$$

which is to say that

$$dir : B \leftrightarrow U; free : \mathbb{P}B \mid free = B \setminus (dim\ dir) \\ \vdash \\ \neg(\exists b : B \bullet b \in dom\ dir \wedge b \in free)$$

The hypothesis above the turnstile is of course simply *SM*; thus our theorem could be stated as

$SM \vdash \neg(\exists b : B \bullet b \in dom\ dir \wedge b \in free)$

□

2 Basic schema operators

In this section we introduce the simplest operations on schemas which produce new schemas.

2.1 Renaming components

From an existing schema we can produce a new schema by systematically changing the names of some of its components. The notation is

$schema[new_1/old_1, new_2/old_2 \ldots]$

Example 16:

A library, if we now interpret *B* as books, not blocks:

$Library \mathrel{\hat{=}} SM[booksonloan/dir, booksonshelves/free]$

We can expand this definition to see the effect of *systematically* changing component names:

```
┌─ Library ─────────────────────────────
│ booksonloan : B ↔ U
│ booksonshelves : ℙB
├───────────────
│ booksonshelves = B\(dom booksonloan)
└───────────────────────────────────────
```

□

Decoration:

Decoration is a special case of renaming a schema's components:

schema decoration

Each component of the schema is systematically decorated. We often use a convention when writing state-based specifications that we add a prime to the variable names to denote the state after a state transition. Decorating a schema with a prime is an easy way to construct a final state from an initial one.

Example 17:

The state of a storage manager after some operation is *SM′*, which is equivalent to

```
┌───────────────────────────────────────
│ dir′ : B ↔ U; free′ : ℙB
├───────────────
│ free′ = B\(dom dir′)
└───────────────────────────────────────
```

□

If we want to use this convention, then we must accept that *SM* and *SM′* both describe the same schema type. Therefore, in considering the type of a schema, and in deciding equality, decorations are ignored. Thus we can safely equate the following tuples

$\theta SM = \theta SM'$

Inclusion:

We can take advantage of previously defined schemas when creating new ones, by 'importing' their definitions. This technique is called *inclusion:*

```
┌───────────────────────────────────────
│ schema
│ declaration
├───────────────
│ predicate
└───────────────────────────────────────
```

The effect of including a schema in the signature of another is to form an augmented schema with a signature containing all the declarations of both schemas, and a predicate which is the conjunction of the schemas' predicates.

Example 18:

A super storage manager keeps a complicated account of the occupancy of the system; it serves little purpose except to enhance the existing design so that it can be sold for more money:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{SuperSM}}} \\
\mathit{SM} \\
\mathit{occupancy} : \mathbb{N} \\
\hline
\mathit{occupancy} = (100 \times \#(\mathit{dom}\ \mathit{dir}))\ \mathit{div}\ \#B \\
\hline
\end{array}$$

This is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{SuperSM}}} \\
\mathit{dir} : B \leftrightarrow U \\
\mathit{free} : \mathbb{P}B \\
\mathit{occupancy} : \mathbb{N} \\
\hline
\mathit{free} = B \setminus (\mathit{dom}\ \mathit{dir}) \\
\mathit{occupancy} = (100 \times \#(\mathit{dom}\ \mathit{dir}))\ \mathit{div}\ \#B \\
\hline
\end{array}$$

□

Example 19:
The new storage manager keeps track of the contiguous free blocks:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{NSM}}} \\
\mathit{SM} \\
\mathit{contiguous} : \mathbb{P}(\mathbb{P}B) \\
\hline
\mathit{contiguous} = \mathit{cfb}\ (\theta \mathit{SM}) \\
\hline
\end{array}$$

This is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{NSM}}} \\
\mathit{dir} : B \leftrightarrow U \\
\mathit{free} : \mathbb{P}B \\
\mathit{contiguous} : \mathbb{P}(\mathbb{P}B) \\
\hline
\mathit{free} = B \setminus (\mathit{dom}\ \mathit{dir}) \\
\mathit{contiguous} = \mathit{cfb}\ (\theta \mathit{SM}) \\
\hline
\end{array}$$

Note the need for θSM in the argument to *cfb*. We apply the function *cfb* to the tuple formed from the component names introduced by the inclusion of *SM*.
□

If we include a schema in the signature of another, any variables with common names are *identified*; i.e. they are regarded as referring to the *same* component. Of course, their declarations *must* agree on the type of the component.

Example 20:

$FancySM \mathrel{\hat=} [busy : \mathbb{P}B;\ dir : B \leftrightarrow U;\ SM \mid busy = dom\ dir]$
□

2.2 *Conventions for using schemas*

In this section we describe some of the conventions that have been found to be useful in developing structured mathematical descriptions of software systems. These conventions are the product of experience gained in many case studies — for example Ref. 1.

The Δ convention:

We often use schemas to describe the state of some part of a system; *SM* is an example of this. We also use schemas to describe the effect of state transitions — operations on the state. To do this, we form a schema which contains both the state before the operation and the state after the operation. The latter has the same component names as the former, but with the addition of a final dash.§ The predicate part of this new schema relates the values of the variables before and after the operation, as well as describing the *invariant conditions* on both states.

Example 21:
The following operation resets a storage manager; i.e. it frees all blocks:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{Reset}}} \\
\mathit{dir}, \mathit{dir}' : B \leftrightarrow U \\
\mathit{free}, \mathit{free}' : \mathbb{P}B \\
\hline
\mathit{free} = B \setminus (\mathit{dom}\ \mathit{dir}) \\
\mathit{free}' = B \setminus (\mathit{dom}\ \mathit{dir}') \\
\mathit{free}' = B \\
\mathit{dir}' = \{\} \\
\hline
\end{array}$$

We have included the names of the variables before and after the operation, the *invariants* that ensure that the two states are indeed *SM* states, and the relationships between before and after variables.
□

We can make the description of operations more concise by using schema inclusion and decoration.

Example 22:
The operation of resetting a storage manager can also be written as

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\mathit{Reset}}} \\
\mathit{SM} \\
\mathit{SM}' \\
\hline
\mathit{free}' = B \\
\mathit{dir}' = \{\} \\
\hline
\end{array}$$

□
Since the inclusion of a schema describing the state before and after an operation occurs so very frequently, we often include both in another schema, and conventionally spell its name with an initial delta (Δ), to conjure up the idea of change.

Example 23:
Let the following schema describe a state change for a simple storage manager operation:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\Delta \mathit{SMOp}}} \\
\mathit{SM} \\
\mathit{SM}' \\
\hline
\end{array}$$

§ This is only a *convention*, and other conventions are also possible. For example, in VDM (Ref. 7), a convention is adopted whereby before variables are decorated with an overhook ($\overleftarrow{x}$, for example) to distingush them from after variables, which go undecorated.

Using this, our reset operation may be written as

$$\begin{array}{|l}
\textit{Reset} \\
\hline
\Delta SMOp \\
\hline
free' = B \\
dir' = \{\} \\
\hline
\end{array}$$

□

The use of the Δ-convention leads to shorter specifications, which we hope are easier to read, because we have encapsulated just what it means to operate on a state — in this case we must preserve the state invariant, which is the essence of what it is to be a storage manager.

Sometimes we may wish to add a further predicate that must be true for all operations on a state of a particular kind: an *operation invariant.* We can include this predicate as part of the Δ-schema, so long as this is not likely to cause confusion to the reader. If there are several kinds of operations, each with different predicates characterising them, then some naming convention should be adopted which distinguishes between them.

The ? and ! conventions:

Not all state transitions can be described in the manner discussed above; some require input or output. Conventionally we decorate the names of inputs with a final query (?), and those of outputs with a final shriek (!).

Example 24:

The interface to the storage manager consists of two complementary operations: one to request a block of storage, and one to release a block of storage. Both of these operations will involve an output that indicates what has happened; the value of this output will be drawn from the datatype Report, which is defined as

$\textit{Report}::\textit{okay} \mid \textit{nospace} \mid \textit{blockfree} \mid \textit{notowner}$

We shall describe the declaration of this output once and for all in an augmented version of $\Delta SMOp$:

$$\begin{array}{|l}
\Delta SM \\
\hline
\Delta SMOp \\
r!: \textit{Report} \\
\hline
\end{array}$$

□

Example 25:

The following operation corresponds to user $u?$ requesting and being allocated a free block $b!$:

$$\begin{array}{|l}
\textit{Request}_0 \\
\hline
\Delta SM \\
u?: U \\
b!: B \\
\hline
free \neq \{\} \\
b! \in free \\
free' = free \setminus \{b!\} \\
dir' = dir \cup \{b! \mapsto u?\} \\
r! = okay \\
\hline
\end{array}$$

That is, there must be a free block to allocate; the block being allocated to $u?$ must be free; and after the operation it is no longer free, but is allocated to $u?$. The state invariant holds before and after the operation.

□

The Ξ convention:

Some operations are nothing more than an interrogation of the state. In such an operation, inputs and outputs may occur, but no state component changes. A schema which records the fact that no change in the state takes place is usually written with an initial xi (Ξ), to conjure up the idea of equality.

Example 26:

We can describe an operation which interrogates the storage manager's state, returning the number of free blocks; it has no side effects:

$$\begin{array}{|l}
\textit{ReadFreeSize} \\
\hline
\Xi SM \\
s!: \mathbb{N} \\
\hline
s! = \#free \\
\hline
\end{array}$$

where

$$\begin{array}{|l}
\Xi SM \\
\hline
\Delta SM \\
\hline
\theta SM = \theta SM' \\
\hline
\end{array}$$

By including ΞSM, we say that *ReadFreeSize* is an operation on the state that leaves the variables of *SM* equal to the variables of *SM'*.

□

3 Logical schema operators

The use of the basic schema operations and associated conventions allows us some convenience in writing specifications: they are useful, but not very powerful. In this section we describe some very simple combinators which allow us to structure specifications in a *very* powerful way. The idea is to explain aspects of a system in the simplest possible context, and then explain more about the system by combining the simpler parts. If schemas are the pieces, then logical schema operations are the glue that allows us to stick them together.

3.1 Schema conjunction

Conjunction is a very powerful specification combinator: we can say what a system must do by listing a number of requirements, and then saying that we must satisfy them all — their *conjunction.* This is a natural way to specify a system.

In the schema notation we have a combinator that corresponds to conjunction of requirements: it is called simply *schema conjunction.* If $schema_1$ and $schema_2$ are both schemas, then so is

$schema_1 \wedge schema_2$

We form the conjunction of two schemas by merging their signatures, identifying any common variables, whose types must agree, and then conjoining their predicates. Thus $schema_1 \wedge schema_2$ has all the components of $schema_1$ and of $schema_2$, and they satisfy both $schema_1$'s predicate, *and* $schema_2$'s predicate.

Example 27:
Suppose that we have a more restricted storage manager than before: not all the blocks are available for use, some are reserved for operating system use, and only a certain number of users are privileged enough to be allowed to own blocks. We can describe the components in our new state as

$$\begin{array}{|l}
\textit{RSM} \\ \hline
blocks : \mathbb{P}B \\
users : \mathbb{P}U \\
alloc : B \leftrightarrow U \\ \hline
\end{array}$$

Of course,, we need to restrict the components to reflect our requirements, and we can do this by considering each requirement separately. First, only certain blocks are allocated to users:

$$\begin{array}{|l}
\textit{RegisteredBlocks} \\ \hline
RSM \\ \hline
\mathrm{dom}\ alloc \subseteq blocks \\ \hline
\end{array}$$

Secondly, only certain users are allowed to own the registered blocks:

$$\begin{array}{|l}
\textit{RegisteredUsers} \\ \hline
RSM \\ \hline
\mathrm{ran}\ alloc \subseteq users \\ \hline
\end{array}$$

Now, the state of the restricted storage manager can be described as the conjunction of the requirements that we have presented:

$$RSMState \mathrel{\hat{=}} RegisteredBlocks \wedge RegisteredUsers$$

This example is so small that it hardly warrants the structure that we have imposed upon it; however it is nice to name requirements, and this becomes indispensable once they become numerous. This technique is known in software engineering as *separation of concerns*. Expanding our definition we get

$$\begin{array}{|l}
\textit{RSMState} \\ \hline
blocks : \mathbb{P}B \\
users : \mathbb{P}U \\
alloc : B \leftrightarrow U \\ \hline
\mathrm{dom}\ alloc \subseteq blocks \\
\mathrm{ran}\ alloc \subseteq users \\ \hline
\end{array}$$

□

Note that in the example we could have described *RSMState* as

$$\begin{array}{|l}
\textit{RSMState} \\ \hline
RegisteredBlocks \\
RegisteredUsers \\ \hline
\end{array}$$

So, inclusion is just a special case of schema conjunction.

Schema disjunction:
Another powerful specification combinator is disjunction; it allows us to offer alternatives in the behaviour of a system. This may be to include some non-determinism in the description, or to present partial descriptions of a system, and then make the description total by combining the parts in disjunction. If $schema_1$ and $schema_2$ are both schemas, then so is

$$schema_1 \vee schema_2$$

We form the disjunction of two schemas by merging their signatures — identifying common variables, whose types must agree — and then disjoining their predicates. Thus $schema_1 \vee schema_2$ has all the components of $schema_1$ and of $schema_2$, and either they satisfy $schema_1$'s predicate, or $schema_2$'s *predicate, or* both. *Schema disjunction* is similar to *schema* conjunction: the signatures of the schemas are merged, common variables are identified, but the predicate parts are disjoined.

Example 28:
The operation of requesting a block from the storage manager is a partial operation:

$$\begin{array}{|l}
\textit{Request}_0 \\ \hline
\Delta SM \\
u? : U \\
b! : B \\ \hline
free \neq \{\} \\
b! \in free \\
free' = free \setminus \{b!\} \\
dir' = dir \cup \{b! \mapsto u?\} \\
r! = okay \\ \hline
\end{array}$$

It is obvious that *Request* cannot proceed successfully if $free = \{\}$: if we try to apply the operation when no free block can be allocated, then we are in considerable trouble! To prevent this unsatisfactory state of affairs, we must specify what should happen in the *unsuccessful* case, when $free = \{\}$: the error response '*nospace*' is returned, and there are no side-effects, the state being left unchanged:

$$\begin{array}{|l}
\textit{RequestErr} \\ \hline
\Xi SM \\ \hline
free = \{\} \\
r! = nospace \\ \hline
\end{array}$$

Now our complete request operation — which is total — is defined as

$$RequestBlock \mathrel{\hat{=}} Request_0 \vee RequestErr$$

We can undo all our good work and partially expand the definition of *RequestBlock*:

RequestBlock

$$\begin{array}{l} \Delta SM \\ u? : U \\ b! : \mathbb{N} \\ \hline (b! \in B \\ free \neq \{\} \\ b! \in free \\ free' = free \setminus \{b!\} \\ dir' = dir \cup \{b! \mapsto u?\} \\ r! = okay) \\ \vee \\ (free = \{\} \\ r! = nospace \\ \theta SM = \theta SM') \end{array}$$

Note that since we were merging the *signatures* of the two schemas, the requirement that

$$b! \in B$$

occurs only in the first disjunct, since $b!$ was not mentioned at all in the schema *RequestErr*. If '*nospace*' is returned, then nothing can be assumed about the value of $b!$; it need not even be a block name. The user of the schema notation should be careful about the need to normalise schemas before combining them with schema disjunction. We return to this point in the next section.
□

Example 29:
When a user who has been allocated a particular block has finished with it, it can be released so that it may be allocated to another user. The block being released, $b?$, really must be owned by the user releasing it, $u?$:

$$b? \in dom\ dir \wedge dir\ b? = u?$$

or more concisely, $(b? \mapsto u?) \in dir$. The block is removed from the directory and put into the *free* set. The successful operation is described by the following definition of $Release_0$:

$Release_0$

$$\begin{array}{l} \Delta SM \\ u? : U \\ b? : B \\ \hline (b? \mapsto u?) \in dir \\ free' = free \cup \{b?\} \\ dir' = \{b?\} \lhd dir \\ r! = okay \end{array}$$

There are two situations in which it would be wrong to release a block $b?$. The first of these is when $b?$ is actually already free:

RelFreeErr

$$\begin{array}{l} \Xi SM \\ b? : B \\ \hline b? \in free \\ r! = blockfree \end{array}$$

The second case occurs when the block is owned by some user, but not by $u?$:

RelOwnerErr

$$\begin{array}{l} \Xi SM \\ u? : U \\ b? : B \\ \hline b? \in dom\ dir \\ dir\ b? \neq u? \\ r! = notowner \end{array}$$

Now the total user interface for the release operation can be described:

$$Release \triangleq Release_0 \vee RelFreeErr \vee RelOwnerErr$$

As an informal argument for the totality of *Release*, first note that, by appeal to the state invariant, there are two cases: either $b?$ is free, or it is in the domain of the directory. In the former case *RelFreeErr* is applicable. In the latter case, either $b?$ is allocated to $u?$, and thus $Release_0$ is applicable, or it is allocated to someone else, and thus *RelOwnerErr* is applicable. This exhausts all possibilities. This is the kind of informal argument that is often carried out in a design review. Later, we shall see how to be more rigorous with the argument.
□

Schema negation:
We can form the *negation* of a schema:

$$\neg schema$$

The effect is to leave the signature part unchanged and to negate the predicate part. The warnings about *normalising* the schema first had better be heeded if silly results are not to follow.

Example 30:
The initial storage manager's state is defined as

SMInit

$$\begin{array}{l} SM' \\ \hline dir' = \{\} \\ free' = B \end{array}$$

We can imagine some later point in the system, where the storage manager has a state different from that which it had initially:

$$NotSMInit \triangleq \neg SMInit$$

Now, if we expand the definition, not forgetting to normalise the declaration, and not forgetting de Morgan's law, we obtain

NotSMInit

$$\begin{array}{l} dir' : \mathbb{P}(\mathbb{N} \times U) \\ free' : \mathbb{P}\mathbb{N} \\ \hline dir' \notin B \leftrightarrow U \vee \\ free' \notin \mathbb{P}B \vee \\ dir' \neq \{\} \vee \\ free' \neq B \end{array}$$

So clearly *NotSMInit* includes all those states in the schema type *SMnorm* except those described by *SMInit*, but including those states which *fail* to satisfy the state invariant for *SM*. If we meant to describe those states of a storage manager that satisfy the state invariant, other than the initial one, then we should have written

$$SM \wedge \neg SMInit$$

□

Existential and universal schema quantifications:

Schema quantification is an operation on a schema; i.e. it takes a schema and in turn produces a schema. The effect is to quantify certain named variables within the schema. They must indeed occur within the schema already, and, as usual, the types must be in agreement:

$$\begin{array}{l} \exists\ decs \bullet schema \\ \forall\ decs \bullet schema \end{array}$$

Example 31:

Suppose that we have our storage management system more economically described as merely a directory

$$ESM \mathrel{\widehat{=}} [dir : B \mathrel{+\!\!\!\rightarrow} U]$$

The free blocks are all those not being used; it is not too difficult to prove the following theorem:

$$ESM \vdash \exists free : \mathbb{P}B \bullet SM$$

Expanding this we get

$$ESM \vdash [dir : B \mathrel{+\!\!\!\rightarrow} U \mid \exists free : \mathbb{P}B \bullet free = B \setminus (\mathrm{dom}\ dir)]$$

or rather more concisely

$$ESM \vdash \exists free : \mathbb{P}B \bullet free = B \setminus (\mathrm{dom}\ dir)$$

which is so, since '\' and '*dom*' are both total.

□

Schema hiding:

Of more frequent use than schema quantification is schema hiding. It is one of the schema language's mechanisms for abstraction — the most important weapon in the war against complexity. We abstract by making certain components part of the internal mechanism of some specification of no concern at the level at which we wish to work. We can hide component names from view as follows:

$$schema \setminus (name_1, name_2, \ldots)$$

The act of hiding a name corresponds to existentially quantifying the name in the schema.

Example 32:

The release operation described earlier required the user to input the name of the block being released, as well as the name of the user. We can imagine a less secure, anonymous release operation where only the name of the block is required:

$$ARelease_0 \mathrel{\widehat{=}} Release_0 \setminus (u?)$$

Expanding the definition we can discover the full definition:

$$\begin{array}{|l} \multicolumn{1}{l}{ARelease_0} \\ \hline \Delta SM \\ b? : B \\ \hline \exists u? : U \bullet \\ \quad (b? \mapsto u?) \in dir \\ \quad free' = free \cup \{b?\} \\ \quad dir' = \{b?\} ⩤ dir \\ \quad r! = okay \\ \hline \end{array}$$

In other words, an anonymous release is just like the ordinary release, because we can find just what the input *u*? should have been. We can go through one further step and eliminate the quantifier, since the value of *u*? that we are looking for is obviously simply *dir b*?, and (*dir b*?) is defined, since *b*? is in (*dom dir*):

$$\begin{array}{lr} \exists u? : U \bullet (b? \mapsto u?) \in dir & \\ \quad \Leftrightarrow \exists u? : U \bullet b? \in \mathrm{dom}\ dir \wedge dir\ b? = u? & \\ & \text{(property of functions)} \\ \quad \Leftrightarrow b? \in \mathrm{dom}\ dir & \text{(existential elimination)} \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{ARelease_0} \\ \hline \Delta SM \\ b? : B \\ \hline b \in \mathrm{dom}\ dir \\ free' = free \cup \{b?\} \\ dir' = \{b?\} ⩤ dir \\ r! = okay \\ \hline \end{array}$$

We could have described our anonymous operation as

$$ARelease_0 \mathrel{\widehat{=}} \exists u? : U \bullet Release_0$$

□

3.2 Schema composition

We can view transitions as relations between states; this prompts us to consider the *composition* of these relations. The expression

$$schema_1 \mathbin{;} schema_2$$

denotes the (forward) *relational composition* of the two schemas. They must follow the convention about dashing variables in the state after an operation. The effect is that the after state of *schema*$_1$ is identified with the before state of *schema*$_2$ and then hidden; the signatures are then merged, identifying common variables; and finally the predicate parts are conjoined. It is as though *schema*$_2$ follows *schema*$_1$.

Example 33:

Let *S* and *T* be two very simple schemas describing operations on a state containing the component *v*:

$$\begin{array}{|l} \multicolumn{1}{l}{S} \\ \hline v, v' : V \\ \hline P(v, v') \\ \hline \end{array}$$

$$\begin{array}{|l}
\hline T \\
v, v' : V \\
\hline
Q(v, v') \\
\hline
\end{array}$$

Then the composition of S with T is defined as follows:

$$S \mathbin{⨾} T \triangleq (S[v_0/v'] \wedge T[v_0/v]) \setminus (v_0)$$

The *transient state* produced by S and consumed by T has been hidden from view, and it occurs existentially quantified in the expanded form of the schema:

$$\begin{array}{|l}
\hline
v, v' : V \\
\hline
\exists v_0 : V \bullet P(v, v_0) \wedge Q(v_0, v') \\
\hline
\end{array}$$

□

Example 34:

A desirable property of the storage management system is that if a block is successfully allocated to a user it may then be successfully released by that user, and the system will be unchanged. We can formalise this requirement as follows:

$$(Request_0[b/b!] \mathbin{⨾} Release_0[b/b?]) \setminus (b) \vdash \Xi SM$$

Because of the identification of common variables, the user $u?$ and the report $r!$ are the same in both operations. The proof of this theorem in our system is straightforward. First, define

$$Op \triangleq (Request_0[b/b!] \mathbin{⨾} Release_0[b/b?]) \setminus (b)$$

If we expand the definition of Op we obtain

$$\begin{array}{|l}
\hline Op \\
\Delta SM \\
u? : U \\
\hline
\exists b : B; free_0 : \mathbb{P}B; dir_0 : B \rightarrowtail U \bullet \\
\quad free_0 = B \setminus (dom\ dir_0) \\
\quad free \neq \{\} \\
\quad b \in free \\
\quad free_0 = free \setminus \{b\} \\
\quad dir_0 = dir \cup \{b \mapsto u?\} \\
\quad (b \mapsto u?) \in dir_0 \\
\quad free' = free_0 \cup \{b\} \\
\quad dir' = \{b\} \lhd dir_0 \\
\quad r! = okay \\
\hline
\end{array}$$

Since we know exactly what values are assumed by dir_0 and $free_0$, we can use existential elimination to remove them:

$$\begin{array}{|l}
\hline Op \\
\Delta SM \\
u? : U \\
\hline
\exists b : B \bullet \\
\quad free \setminus \{b\} = B \setminus (dom\ (dir \cup \{b \mapsto u?\})) \\
\quad free \neq \{\} \\
\quad b \in free \\
\quad (b \mapsto u?) \in dir \cup \{b \mapsto u?\} \\
\quad free' = (free \setminus \{b\}) \cup \{b\} \\
\quad dir' = \{b\} \lhd (dir \cup \{b \mapsto u?\}) \\
\quad r! = okay \\
\hline
\end{array}$$

Consider the predicate that corresponds to the invariant on the intermediate state: it follows from the invariant on the before state, and some simple properties of the domain and set difference functions:

$$\begin{array}{ll}
B \setminus (dom\ (dir \cup \{b \mapsto u?\})) & \\
\quad = B \setminus ((dom\ dir) \cup dom\ \{b \mapsto u?\} & \\
 & \text{(distributivity of } dom) \\
\quad = B \setminus ((dom\ dir) \cup \{b\}) & \text{(property of } dom) \\
\quad = (B \setminus (dom\ dir)) \setminus \{b\} & \text{(property of } \setminus) \\
\quad = free \setminus \{b\} & \text{(invariant on } SM)
\end{array}$$

In the intermediate state, b is temporarily allocated to $u?$. This is described by the predicate

$$(b \mapsto u?) \in dir \cup \{b \mapsto u?\}$$

which is actually just a simple property of set membership. If b is in *free*, then removing it and then putting it back again leaves *free* unchanged. Thus

$$\begin{array}{l}
b \in free \wedge free' = (free \setminus \{b\}) \cup \{b\} \\
\quad \Leftrightarrow (free \setminus \{b\}) \cup \{b\} = free \wedge free' \\
\qquad\qquad = (free \setminus \{b\}) \cup \{b\} \\
\quad \Leftrightarrow free' = free
\end{array}$$

If b is in *free*, then it is not in the domain of the directory; thus allocating b to $u?$ and then removing it from the directory leaves the directory unchanged. Note that

$$\begin{array}{ll}
b \in free & \\
\quad \Leftrightarrow b \in B \setminus (dom\ dir) & \text{(invariant on } SM) \\
\quad \Leftrightarrow b \in B \wedge b \notin (dom\ dir) & \text{(property of } \setminus)
\end{array}$$

Thus

$$\{b\} \lhd dir = dir$$

Remembering that b is free, consider the final value of the directory:

$$\begin{array}{ll}
dir' = \{b\} \lhd (dir \cup \{b \mapsto u?\}) & \\
\quad \Leftrightarrow dir' = (\{b\} \lhd dir) \cup (\{b\} \lhd \{b \mapsto u?\}) & \\
 & \text{(distributivity of } \lhd) \\
\quad \Leftrightarrow dir' = (\{b\} \lhd dir) \cup \{\} & \text{(property of } \lhd) \\
\quad \Leftrightarrow dir' = (\{b\} \lhd dir) & \text{(property of } \cup) \\
\quad \Leftrightarrow dir' = dir & (b \text{ is free})
\end{array}$$

Using these simplifications, we can reduce Op to

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Op}} \\ \hline
\Delta SM \\
u? : U \\ \hline
\exists b : B \bullet \\
\quad free \neq \{\} \\
\quad b \in free \\
\quad free' = free \\
\quad dir' = dir \\
\quad r! = okay \\ \hline
\end{array}$$

Of course we can always find an element in a non-empty set, so we can use existential elimination once more:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Op}} \\ \hline
\Delta SM \\
u? : U \\ \hline
free \neq \{\} \\
b \in free \\
free' = free \\
dir' = dir \\
r! = okay \\ \hline
\end{array}$$

Remembering the Ξ convention, we can rewrite this as

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Op}} \\ \hline
\Xi SM \\
u? : U \\ \hline
free \neq \{\} \\
b \in free \\
r! = okay \\ \hline
\end{array}$$

Whence our result.

□

We should pose and prove such desirable properties as theorems of our specifications. The presence of these theorems and their proofs distinguishes better specifications from poorer ones.

4 Precondition investigation

A *precondition* for an operation describes all the states from which it is guaranteed that a final state can be reached.

The style of specifying state transitions that we have touched upon in this paper involves just a single predicate relating the state before and the state after an operation. After all, a state transition is just a relation. Those familiar with the VDM style (Ref. 7) may have expected — but failed to find — a pair of predicates: a *precondition* and a *postcondition*. The answer to this riddle is that in the schema language the single predicate serves both purposes. In fact, we can *calculate* the precondition of an operation defined in a schema:

pre *schema*

is defined to be *schema* with all the after variables and outputs hidden in the manner described above. A precondition in Z is something that *guarantees* that a final state can in fact be reached. There is some advantage in *calculating* the precise precondition, rather than asserting it. In fact, any asserted precondition would have to be at least as strong as the precondition which may be calculated in the manner suggested. The VDM specification (*true*, *false*) has no equivalent in Z; it simply cannot be written. It denotes the operation which can always be applied, but which can achieve no result.

Example 35:

Recall the definition of the successful operation of requesting and being allocated a block of storage from the storage manager:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Request}_0} \\ \hline
\Delta SM \\
u? : U \\
b! : B \\ \hline
free \neq \{\} \\
b! \in free \\
free' = free \setminus \{b!\} \\
dir' = dir \cup \{b! \mapsto u?\} \\
r! = okay \\ \hline
\end{array}$$

Although we have specified that the state invariant is preserved by including ΔSM, and therefore SM', how can we tell that this operation really does just that? We must calculate the precondition.

Theorem 1:

The precondition for $Request_0$ is just $free \neq \{\}$. Let

$PreRequest_0 \mathrel{\widehat{=}} \text{pre } Request_0$

i.e. $PreRequest_0$ is the schema

$\exists SM'; b! : B; r! : Report \bullet Request_0$

Then we must show that $PreRequest_0$ is simply

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PreRequest}_0} \\ \hline
SM \\
u? : U \\ \hline
free \neq \{\} \\ \hline
\end{array}$$

Proof:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{PreRequest}_0} \\ \hline
SM \\
u? : U \\ \hline
\exists dir' : B \rightarrowtail U; free' : \mathbb{P}B; b! : B; r! : Report \bullet \\
\quad free \neq \{\} \\
\quad b! \in free \\
\quad free' = free \setminus \{b!\} \\
\quad dir' = dir \cup \{b! \mapsto u?\} \\
\quad r! = okay \\
\quad free' = B \setminus (dom\ dir') \\ \hline
\end{array}$$

First, since we have three equations giving us alternative expressions for dir', $free'$ and $r!$, then we can use existential elimination to remove them from the quantified predicate.

Note that we still retain two proof obligations:

- *dir′* is not just a relation between B and U, but a *partial function*.
- *free′* is not just a subset of $\mathbb{N}$, but a subset of B.

$$\begin{array}{|l}
\multicolumn{1}{l}{PreRequest_0} \\
SM \\
u? : U \\
\hline
\exists b! : B \bullet \\
\quad dir \cup \{b! \mapsto u?\} \in B \pfun U \\
\quad free \setminus \{b!\} \subseteq B \\
\quad free \neq \{\} \\
\quad b! \in free \\
\quad free \setminus \{b!\} = B \setminus (dom\ (dir \cup \{b! \mapsto u?\})) \\
\hline
\end{array}$$

It is fairly easy to dispose of the first of the quantified conjuncts which requires that the enlarged directory remains functional, since it happens to be equivalent to the directory already being functional (a fact known from *SM*) and $b!$ being free:

$$\begin{aligned}
& dir \cup \{b! \mapsto u?\} \in B \pfun U \\
& \quad \Leftrightarrow dir \in B \pfun U \wedge b! \notin dom\ dir \wedge b! \in B \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(property of functions)} \\
& \quad \Leftrightarrow dir \in b \pfun U \wedge b! \in B \setminus (dom\ dir) \qquad \text{(property of } \setminus \text{)} \\
& \quad \Leftrightarrow dir \in b \pfun U \wedge b! \in free \qquad \text{(invariant on } SM\text{)}
\end{aligned}$$

If $b!$ is in *free*, then removing $b!$ from *free* leaves it still a subset of B:

$$\begin{aligned}
& free \setminus \{b!\} \subseteq B \wedge b! \in free \\
& \quad \Rightarrow (free \setminus \{b!\}) \cup \{b!\} \subseteq B \cup \{b!\} \wedge b! \in free \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(monotonicity of } \cup \text{)} \\
& \quad \Rightarrow (free \setminus \{b!\}) \cup \{b!\} \subseteq B \wedge b! \in free \\
& \qquad\qquad\qquad\qquad\qquad\qquad (b! \text{ quantified over } B) \\
& \quad \Rightarrow free \subseteq B \wedge b! \in free \qquad \text{(property of } \subseteq \text{)}
\end{aligned}$$

In fact, we can strengthen this to an equivalence:

$$\begin{aligned}
& free \subseteq B \\
& \quad \Rightarrow free \setminus \{b!\} \subseteq B \qquad \text{(anti-monotonicity of } \setminus \text{)}
\end{aligned}$$

Operation	Input	Output	Precondition
InitSM			*true*
Request$_0$	$u? : U$	$b! : B$ $r! : Report$	$free \neq \{\}$
RequestErr		$r! : Report$	$free = \{\}$
Release$_0$	$u? : U$ $b? : B$	$r! : Report$	$(b? \mapsto u?) \in dir$
RelFreeErr	$b? : B$	$r! : Report$	$b? \in free$
RelOwnerErr	$u? : U$ $b? : B$	$r! : Report$	$b? \in dom\ dir$ $dir\ b? \neq u?$

Fig. 1 Preconditions for the storage allocation operations

We saw earlier that

$$free \setminus \{b!\} = B \setminus (dom\ (dir \cup \{b! \mapsto u?\}))$$

follows from the invariant on SM and some properties of the operators. Using these results, we can rewrite our schema as

$$\begin{array}{|l}
\multicolumn{1}{l}{PreRequest_0} \\
SM \\
u? : U \\
\hline
\exists b! : B \bullet \\
\quad free \neq \{\} \\
\quad dir \in B \pfun U \\
\quad b! \in free \\
\quad free \subseteq B \\
\hline
\end{array}$$

Of course, the functionality of *dir* follows from the invariant on *SM*, as does the fact that $free : \mathbb{P} B$, and therefore $free \subseteq B$. Making these simplifications, we have

$$\begin{array}{|l}
\multicolumn{1}{l}{PreRequest_0} \\
SM \\
u? : U \\
\hline
\exists b! : B \bullet \\
\quad free \neq \{\} \\
\quad b! \in free \\
\hline
\end{array}$$

But of course, if *free* is not empty, then we can always find an element in it. So finally, we can use existential elimination to remove reference to $b!$, giving us

$$\begin{array}{|l}
\multicolumn{1}{l}{PreRequest_0} \\
SM \\
u? : U \\
\hline
free \neq \{\} \\
\hline
\end{array}$$

QED.

So, the operation of requesting a block from the storage manager is a partial one: $Request_0$ cannot proceed successfully if $free = \{\}$. If we try to apply the operation when no free block can be allocated, then we are in considerable trouble!

□

Whenever we describe a system, we should investigate the preconditions of the operations. It often helps to present the results of these investigations in an easily readable form.‡ To summarise the operations, we present their preconditions in Fig. 1.

5 A concise history of the schema

In this paper I have tried to give an account of the role played by schemas in structuring specifications written in Z. Although the illustrations are novel, I have merely chronicled the work of many other people. In this section I would like briefly to set out how the notion of schemas developed. Although I have gathered details from those

‡ This method of presentation was suggested by John Wordsworth, a colleague at IBM UK Laboratories, Hursley Park.

close to hand when this section was written,† I am to blame for any inaccuracies in this history.

By 1982, researchers at Oxford and elsewhere, including Jean-Raymond Abrial, Tim Clement, Steve Schuman, Ib Holm Sørensen and Bernard Sufrin, had started to use schemas. They used them more or less as though they were textual macros, though there was already some debate about this. They appeared in contexts which would be familiar if you have read this paper: for example, in lambda abstractions, set comprehensions and quantifications in logical formulas. Like macros, schemas could always be expanded by replacing the name of the schema by its body; thus the underlying mathematics could always be recovered.

Specifications were given structure by using *classes* and *chapters*, containing definitions of state and operations in terms of variables, types, relations and functions. Nondeterminism was thought to be captured by functions that returned arbitrary elements of sets — using μ-terms.

The specification of operations by schemas as well as merely states was introduced by Carroll Morgan following a suggestion by Tony Hoare, after discussions with Bernard Sufrin. The first schema operators appeared soon after — semi-colon and disjunction — both first used by Carroll Morgan. It was this last application that first separated the description of errors from that of the successful part of the operation. This presentation style — now a widely used Z style — using primes, queries, shrieks, Δs and Ξs was further developed by the Distributed Computing Software project and then taken over by the CICS formalisation project (see Ref. 1). The Δ was suggested by Steve Schuman. Promotion — or framing — was worked out by Bernard Sufrin and Carroll Morgan (see Ref. 8).

As these schema operators were shown to be useful in practice, so the ideas were more fully worked out and other logical operators appeared, such as conjunction and piping — the latter invented by Ian Hayes. The name *schema calculus* was coined by Bernard Sufrin to describe the various logical operators between schemas. The meaning of the calculus, however, was still inspired by macros.

In defiance of popular advice at the time, Mike Spivey gave a denotational description of the meaning of schemas (Ref. 3). History has shown that this has been a most successful course.

Further developments in the language of schemas are still taking place: in particular, Mike Spivey has recently suggested the idea of nesting schemas, one inside another, to give a hierarchical structure, as opposed to the rather flat results of schema inclusion. This would allow the packaging of abstract data types, for example. Another application, being worked upon by the author, is in the description of process structures in Concurrent Z.

6 Acknowledgments

This paper follows an earlier introduction to the schema language (Ref. 9), although the illustrations are entirely new and the notation has been revised according to Ref. 2. The ideas were developed over a number of years at Oxford by members of the Programming Research Group (Ref. 1); I am particularly grateful to Ian Hayes, Carroll Morgan, Ib Holm Sørensen and Bernard Sufrin for showing me how to use them.

This paper was written as part of the teaching notes for a course in mathematics for software engineering, and as part of the reference material for the Z notation. Its production has been encouraged and supported by IBM UK Laboratories — in particular by John Nicholls and John Wordsworth — to whom I am most grateful.

Ian Hayes at the University of Queensland, Steve King at Oxford University and Sam Valentine and Rod Bark, lately of Systems Designers, spent a lot of time reading and commenting on earlier drafts of this paper. Students on the M.Sc. in Computation at Oxford and on several industrial courses removed obscurities and mistakes. Their diligence is much appreciated. Jock McDoowi read and corrected all the drafts.

† Particular thanks are due to Carroll Morgan.

7 References

1 HAYES, I. (Ed.): 'Specification case studies' (Prentice-Hall International, 1987)

2 KING, S., SØRENSEN, I.H., and WOODCOCK, J.C.P.: 'Z: grammar and concrete and abstract syntaxes'. Monograph PRG-68, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, 1988

3 SPIVEY, J.M.: 'Understanding Z: a specification language and its formal semantics'. Cambridge Tracts in Theoretical Science 3 (Cambridge University Press, 1988)

4 SPIVEY, J.M.: 'The Z notation: a reference manual' (Prentice-Hall International, 1989)

5 SUFRIN, B. (Ed.): 'The Z handbook'. Programming Research Group, Oxford University Computing Laboratory, Oxford, England, 1986

6 WOODCOCK, J., and LOOMES, M.: 'Software engineering mathematics' (Pitman, 1988)

7 JONES, C.B.: 'Systematic software development using VDM' (Prentice-Hall International, 1986)

8 MORGAN, C.C., and SUFRIN, B.: 'Specification of the UNIX filing system', *IEEE Transactions on Software Engineering*, 1984, **SE-10,** (2), pp. 128–142

9 MORGAN, C.C.: 'The Schema language'. Course Notes, M.Sc. in Computation, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, July 1984

Dr. J.C.P. Woodcock is Joint Rutherford-Pembroke Atlas Research Fellow in Computation with the Programming Research Group, Oxford University Computing Laboratory, 8–11 Keble Road, Oxford OX1 3QD, England.

8 Appendix — laws used in proofs

In this section, we describe only those laws which we used to justify our proofs. Most of them come from the Z reference manual (Ref. 4); those that do not are easily proved from others which are.

8.1 *Properties of functions*

The following equivalence is often used to rewrite a property in a specification, because the right-hand side is a more concise expression. There is usually a trade-off in these matters: the left-hand side is often more useful in a proof:

$$f : X \rightarrowtail\!\!\!\!\!\!+\; Y; x : X; y : Y \vdash$$
$$x \in dom\, f \land f x = y \Leftrightarrow (x \mapsto y) \in f$$

When we unite two functions, the result is a function just in case the functions' domains are disjoint:

$$f, g : X \rightarrowtail\!\!\!\!\!+ Y \vdash f \cup g \in X \rightarrowtail\!\!\!\!\!+ Y \Leftrightarrow (dom\, f) \cap (dom\, g) = \{\}$$

8.2 *Properties of the domain function*

The *dom* function is ***distributive***:

$$f, g : X \leftrightarrow Y \vdash dom\,(f \cup g) = (dom\, f) \cup (dom\, g)$$

Its effect on the singleton function is simple:

$$x : X; y : Y \vdash dom\,\{x \mapsto y\} = x$$

8.3 *Properties of set difference*

The set difference operator may be defined by its membership property:

$$x : X; A, B : \mathbb{P}X \vdash x \in A \setminus B \Leftrightarrow x \in A \wedge x \notin B$$

If we subtract the union of two sets it is the same as if we subtracted each set individually:

$$A, B, C : \mathbb{P}X \vdash A \setminus (B \cup C) = (A \setminus B) \setminus C$$

In a special case, set difference and union can act as inverses:

$$A, B : \mathbb{P}X \vdash B \subseteq A \Leftrightarrow (A \setminus B) \cup B = A$$

Set difference is *anti-monotonic* in its second argument; i.e. it *reverses* the inclusion ordering:

$$A, B, C : \mathbb{P}X \vdash B \subseteq C \Rightarrow A \setminus C \subseteq A \setminus B$$

A useful special case of anti-monotonicity gives us

$$A, B : \mathbb{P}X \vdash A \setminus B \subseteq A$$

This property relies on the inclusion ordering having a least element.

8.4 *Properties of set union*

The set union operator has the empty set as an identity:

$$A : \mathbb{P}X \vdash A \cup \{\} = A$$

It is *monotonic*, in the sense that it respects the inclusion ordering:

$$A, B, C : \mathbb{P}X \vdash A \subseteq B \Rightarrow A \cup C \subseteq B \cup C$$

8.5 *Properties of set inclusion*

The inclusion ordering on sets has a least element:

$$A : \mathbb{P}X \vdash \{\} \subseteq A$$

It is a transitive relation

$$A, B, C : \mathbb{P}X \vdash A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C$$

Transitivity together with the special case of anti-monotonicity of set difference gives us the property

$$A, B, C : \mathbb{P}X \vdash A \subseteq B \Rightarrow A \setminus C \subseteq B$$

8.6 *Properties of domain subtraction*

Domain subtraction§ is distributive:

$$f, g : X \leftrightarrow Y; A : \mathbb{P}X \vdash A \mathbin{\lhd\!\!\!-} (f \cup g) = (A \mathbin{\lhd\!\!\!-} f) \cup (A \mathbin{\lhd\!\!\!-} g)$$

We find the following trivial property helpful:

$$x : X; y : Y \vdash \{x\} \mathbin{\lhd\!\!\!-} \{x \mapsto y\} = \{\}$$

§ Also called 'domain anti-restriction' in Ref. 4.