An Introduction to Z and Formal Specifications

J. M. Spivey
Oxford University Computing Laboratory
Programming Research Group
11, Keble Road, Oxford, OX1 3QD

June 1988

Abstract

This article is an introduction to the description of information systems using formal, mathematical specifications written in the Z notation, and to the refinement of these specifications into rigorously-checked designs.

The first part introduces the idea of a formal specification using a simple example: that of a "birthday book" in which people's birthdays can be recorded, and which is able to issue reminders on the appropriate day. The behaviour of this system for correct input is specified first, then the schema calculus is used to strengthen the specification into one requiring error reports for incorrect input.

The second part of the article introduces the idea of data refinement as the primary means of constructing designs which achieve a formal specification. Refinement is presented through the medium of two examples; the first is a direct implementation of the birthday book from part one, and the second is a simple checkpoint facility, which allows the current state of a database to be saved and later restored. A Pascal-like programming language is used to show the code for some of the operations in the examples.

1 What is a formal specification?

Formal specifications use mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved. They describe *what* the system must do without saying *how* it is to be done. This *abstraction* makes formal specifications useful in the process of developing a computer system, because they allow questions about what the system does to be answered confidently, without the need to disentangle the information from a mass of detailed program code, or to speculate about the meaning of phrases in an imprecisely-worded prose description.

A formal specification can serve as a single, reliable reference point for those who investigate the customer's needs, those who implement programs to satisfy those needs, those who test the results, and those who write instruction manuals for the system. Because it is independent of the program code, a formal specification of a system can be completed early in its development. Although it might need to be changed as the design team gains in understanding and the perceived needs of the customer evolve, it can be a valuable means of promoting a common understanding among all those concerned with the system.

Existing data modelling techniques – for example, relational databases [1] – provide some of the abstraction we need, in that they free us from describing the exact layout of data in the memory of a computer. But they are limited to fairly simple models, and they are oriented towards direct implementation: considerations of efficiency rather than clarity often dictate the structure of the model. Also, whilst data modelling techniques can help to clarify the description of a system, they provide little support for reasoning about its behaviour.

One way in which mathematical notation can help to achieve these goals is through the use of *mathematical data types* to model the data in a system. These data types are not oriented towards computer representation, but they obey a rich collection of mathematical laws which make it possible to reason effectively about the way a specified system will behave. We use the notation of *predicate logic* to describe abstractly the effect of each operation of our system, again in a way that enables us to reason about its behaviour. These two ideas also form important ingredients of other formal methods such as VDM [5, 6].

The other main ingredient in Z is a way of decomposing a specification into small pieces called *schemas*. By splitting the specification into schemas, we can present it piece by piece. Each piece can be linked with a commentary which explains informally the significance of the formal mathematics. In Z, schemas are used to describe both static and dynamic aspects of a system. The static aspects include

- the states it can occupy.
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include

- the operations that are possible.
- the relationship between their inputs and outputs.
- the changes of state that happen.

Later, we shall see how the schema language allows different facets of a system to be described separately, then related and combined. For example, the operation of a system when it receives valid input may be described first, then the description may be extended to show how errors in the input are handled. Or the evolution of a single process in a complete system may be described in isolation, then related to the evolution of the system as a whole.

We shall also see how schemas can be used to describe a transformation from one view of a system to another, and so explain why an abstract specification is correctly implemented by another containing more details of a concrete design. By constructing a sequence of specifications, each containing more details than the last, we can eventually arrive at a program with confidence that it satisfies the specification.

2 The birthday book

The best way to see how these ideas work out is to look at a small example. For a first example, it is important to choose something simple, and I have chosen a system so simple that it is usually implemented with a notebook and pencil rather than a computer. It is a system which records people's birthdays, and is able to issue a reminder when the day comes round. The first thing to describe is the *state space* of the system, and we do this with a schema:

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook} \\
\textit{known} : \mathbb{P}\, \textit{NAME} \\
\textit{birthday} : \textit{NAME} \nrightarrow \textit{DATE} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}
$$

Like most schemas, this consists of a part above the central dividing line, in which some variables are declared, and a part below the line which gives a relationship between the values of the variables. In this case we are describing the state space of a system, and the two variables represent important *observations* which we can make of the state:

- *known* is the set of names with birthdays recorded.
- *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.

The part of the schema below the line gives a relationship which is true in every state of the system and is maintained by every operation on it: in this case, it says that the set *known* is the same as the domain of the function *birthday* – the set of names to which it can be validly applied. This relationship is an *invariant* of the system.

In this example, the invariant allows the the value of the variable *known* to be derived from the value of *birthday*: *known* is a *derived* component of the state, and it would be possible to specify the system without mentioning *known* at all. However, giving names to important concepts helps to make specifications more readable; because we are describing an abstract view of the state space of the birthday book, we can do this without making a commitment to represent *known* explicitly in an implementation.

One possible state of the system is the following:

$$\textit{known} = \{\, \text{John}, \text{Mike}, \text{Susan} \,\}$$

$$
\begin{aligned}
\textit{birthday} = \{\, & \text{John} \mapsto \text{25–Mar}, \\
& \text{Mike} \mapsto \text{20–Dec}, \\
& \text{Susan} \mapsto \text{20–Dec} \,\}.
\end{aligned}
$$

Here there are three names known to the system, and the birthday function associates a date with each of them.

Notice that in this description of the state space of the system, we have not been forced to place a limit on the number of birthdays recorded in the birthday book, nor to say that the entries will be stored in a particular order. We have also avoided making a premature decision about the format of names and dates. On the other hand, we have concisely captured the information that each person can have only one birthday, because the variable *birthday* is a function, and that two people can share the same birthday as in our example.

So much for the state space; we can now start on some *operations* on the system. The first of these is to add a new birthday, and we describe it with a schema:

AddBirthday

$\Delta BirthdayBook$
$name? : NAME$
$date? : DATE$

$name? \notin known$

$birthday' = birthday \cup \{name? \mapsto date?\}$

The declaration $\Delta BirthdayBook$ alerts us to the fact that the schema is describing a *state change*: it introduces four variables *known*, *birthday*, $known'$ and $birthday'$. The first two are observations of the state before the change, and the last two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark.

The part of the schema below the line first of all gives a *pre-condition* for the success of the operation: the name to be added must not already be one of those known to the system. This is reasonable, since each person can only have one birthday. What happens if the pre-condition is not satisfied is not specified here: we shall see later how to extend the specification to say that an error message is to be produced. If the pre-condition is satisfied, however, the second line says that the birthday function is extended to map the new name to the given date.

We expect that the set of names known to the system will be augmented with the new name:

$$known' = known \cup \{name?\}.$$

In fact we can *prove* this from the specification of *AddBirthday*, using the invariants on the state before and after the operation:

$$\begin{aligned}
&known' \\
&\quad = \mathrm{dom}\, birthday' && \text{[invariant after]} \\
&\quad = \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]} \\
&\quad = \mathrm{dom}\, birthday \cup \mathrm{dom}\, \{name? \mapsto date?\} && \text{[fact about dom]} \\
&\quad = \mathrm{dom}\, birthday \cup \{name?\} && \text{[fact about dom]} \\
&\quad = known \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

Stating and proving properties like this one is a good way of making sure the specification is accurate; reasoning from the specification allows us to explore the behaviour of the system without going to the trouble and expense of implementing it. The two facts about dom used in this proof are examples of the laws obeyed by mathematical data types:

$$\mathrm{dom}(f \cup g) = (\mathrm{dom}\, f) \cup (\mathrm{dom}\, g)$$

$$\mathrm{dom}\{a \mapsto b\} = \{a\}.$$

The standard mathematical data types of Z obey many laws like these.

Another operation might be to find the birthday of a person known to the system. Again we describe the operation with a schema:

$$\begin{array}{|l}
\hline
\textit{FindBirthday} \\
\Xi BirthdayBook \\
name? : NAME \\
date! : DATE \\
\hline
name? \in known \\
date! = birthday(name?) \\
\hline
\end{array}$$

Two new notations are illustrated by this schema. One is the declaration $\Xi BirthdayBook$, which indicates an operation in which the state does not change: the values $known'$ and $birthday'$ of the observations after the operation are equal to their values $known$ and $birthday$ beforehand. Including $\Xi BirthdayBook$ above the line is the same as including $\Delta BirthdayBook$ above the line and the two equations:

$$known' = known$$

$$birthday' = birthday$$

below it. The other notation is the use of a name ending in an exclamation mark for an output: the *FindBirthday* operation takes a name as input and yields the corresponding birthday as output. The pre-condition for success of the operation is that $name?$ is one of the names known to the system; if this is so, the output $date!$ is the value of the birthday function at argument $name?$.

The most useful operation on the system is the one to find which people have birthdays on a given date. The operation has one input $today?$, and one output, $cards!$, which is a *set* of names: there may be zero, one, or more people with birthdays on a particular day, to whom birthday cards should be sent.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \{\, n : known \mid birthday(n) = today? \,\}$$

Again the Ξ convention is used to indicate that the state does not change. This time there is no pre-condition. The output $cards!$ is specified to be equal to the set of all values n drawn from the set $known$ such that the value of the birthday function at n is $today?$. In general, y is a member of the set $\{\, x : S \mid \ldots x \ldots \}$ exactly if y is a member of S and the condition $\ldots y \ldots$, obtained by replacing x with y, is satisfied:

$$y \in \{\, x : S \mid \ldots x \ldots \} \Leftrightarrow y \in S \wedge (\ldots y \ldots).$$

So, in our case,

$$\begin{aligned} &m \in \{\, n : \mathit{known} \mid \mathit{birthday}(n) = \mathit{today?} \,\} \\ &\qquad \Leftrightarrow m \in \mathit{known} \wedge \mathit{birthday}(m) = \mathit{today?}. \end{aligned}$$

A name m is in the output set *cards*! exactly if it is known to the system and the birthday recorded for it is *today*?.

To finish the specification, we must say what state the system is in when it is first started. This is the *initial state* of the system, and it also is specified by a schema:

$$\begin{array}{|l} \hline \mathit{InitBirthdayBook} \\ \mathit{BirthdayBook} \\ \hline \mathit{known} = \varnothing \\ \hline \end{array}$$

This schema describes a birthday book in which the set *known* is empty: in consequence, the function *birthday* is empty too.

What have we achieved in this specification? We have described in the same mathematical framework both the state space of our birthday-book system and the operations which can be performed on it. The data objects which appear in the system were described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state – information which would not be part of a program implementing the system, but which is vital to understanding it.

The effects of the operations are described in terms of the relationship which must hold between the input and the output, rather than by giving a recipe to be followed. This is particularly striking in the case of the *Remind* operation, where we simply documented the conditions under which a name should appear in the output. An implementation would probably have to examine the known names one at a time, printing the ones with today's date as it found them, but this complexity has been avoided in the specification. The implementor is free to use this technique, or any other one, as he or she chooses.

Mathematical specifications have the three virtues of being concise, precise and unambiguous. They are *concise* because mathematical notation is capable of expressing complex facts about information systems in a short space. Practical experience shows that a mathematical specification of a system is often much shorter than an equivalent informal specification. Hayes [3] reports that a formal specification for a module in the CICS system is comparable in length with the less informative English-language manual entry for the same module. Mathematical specifications are *precise* because they allow requirements to be documented accurately. The desired function of a system is described in a way that does not unduly constrain either the data structures used to represent the information in the system, or the algorithms used to compute with it. Finally, mathematical specifications are *unambiguous*: differences of interpretation can be avoided when specifications are expressed in a standardized language with a well-understood meaning.

Exercise. We have seen how schemas can be used to describe the state space and operations of a system. Now try to write a Z specification for the following system: a teacher wants to keep a register of students in her class, and to record which of them have completed their homework. Specify:

1. The state space for a register. [Hint: use two sets of students

   $$\begin{array}{|l}
   \multicolumn{1}{l}{\textit{Register}} \\
   \hline
   \textit{enrolled} : \mathbb{P}\,\textit{STUDENT} \\
   \textit{completed} : \mathbb{P}\,\textit{STUDENT} \\
   \hline
   \ldots \\
   \hline
   \end{array}$$

   Think carefully about the invariant].

2. An operation to enroll a new student.

3. An operation to record that a student (already enrolled in the class) has finished the homework.

4. An operation to enquire whether a student (who must be enrolled) has finished the homework. (Answer in the set $\{\textit{Yes}, \textit{No}\}$).

3 Strengthening the specification

A correct implementation of our specification will faithfully record birthdays and display them, so long as there are no mistakes in the input. But the specification has a serious flaw: as soon as the user tries to add a birthday for someone already known to the system, or tries to find the birthday of someone not known, it says nothing about what happens next. The action of the system may be perfectly reasonable: it may simply ignore the incorrect input. On the other hand, the system may break down: it may start to display rubbish, or perhaps worst of all, it may appear to operate normally for several months, until one day it simply forgets the birthday of a rich and elderly relation.

Does this mean that we should scrap the specification and begin a new one? That would be a shame, because the specification we have describes the behaviour for correct input clearly and concisely, and modifying it to describe the handling of incorrect input could only make it obscure. Luckily, there is a better solution: we can describe, separately from the first specification, the errors which might be detected and the desired responses to them, then use the operations of the Z *schema calculus* to combine the two descriptions into a stronger specification.

We shall add an extra output $result!$ to each operation on the system. When an operation is successful, this output will take the value ok, but it may take other values when an error is detected. We first describe an operation $Success$ which just produces the result ok:

$$\begin{array}{|l}
\hline
\textit{Success} \\
\hline
result! : REPORT \\
\hline
result! = ok \\
\hline
\end{array}$$

The conjunction operator $\wedge$ of the schema calculus allows us to combine this description with our previous description of $AddBirthday$:

$$AddBirthday \wedge Success.$$

This describes an operation which, for correct input, both acts as described by $AddBirthday$ and produces the result ok.

For each error that might be detected in the input, we specify an operation which produces an appropriate report when the error has occurred. Here is an operation which produces the report $already\_known$ when its input $name?$ is already a member of $known$:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AlreadyKnown}} \\ \hline
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{result}! : \textit{REPORT} \\ \hline
\textit{name}? \in \textit{known} \\
\textit{result}! = \textit{already_known} \\ \hline
\end{array}$$

If the error occurs, this schema specifies that the state of the system should not change.

We can combine this description with the previous one to give a specification for a robust version of *AddBirthday*:

$$\textit{RAddBirthday} \mathrel{\widehat{=}} (\textit{AddBirthday} \land \textit{Success}) \lor \textit{AlreadyKnown}.$$

This definition written with the sign $\widehat{=}$ introduces a new schema called *RAddBirthday*, obtained by combining the three schemas shown on the right-hand side. The operation *RAddBirthday* must terminate whatever its input. If the input *name*? is already known, the state of the system does not change, and the result *already_known* is returned; otherwise, the new birthday is added to the database as described by *AddBirthday*, and the result *ok* is returned.

We have specified the various requirements for this operation separately, and then combined them into a single specification of the whole behaviour of the operation. This does not mean that each requirement must be implemented separately, and the implementations combined somehow. In fact, an implementation might search for a place to store the new birthday, and at the same time check that the name is not already known; the code for normal operation and error handling might be thoroughly mingled. This is an example of the abstraction which is possible when we use a specification language free from the constraints necessary in a programming language. The operators $\land$ and $\lor$ cannot (in general) be implemented efficiently as ways of combining programs, but this should not stop us from using them to combine specifications if that is a convenient thing to do.

The operation *RAddBirthday* could be specified directly by writing a single schema which combines the predicate parts of the three constituents *AddBirthday*, *Success* and *AlreadyKnown*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RAddBirthday}} \\
\hline
\Delta \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{date}? : \mathit{DATE} \\
\mathit{result}! : \mathit{REPORT} \\
\hline
(\mathit{name}? \notin \mathit{known} \land \\
\quad \mathit{birthday}' = \mathit{birthday} \cup \{\mathit{name}? \mapsto \mathit{date}?\} \land \\
\quad \mathit{result}! = \mathit{ok}) \lor \\
(\mathit{name}? \in \mathit{known} \land \\
\quad \mathit{birthday}' = \mathit{birthday} \land \\
\quad \mathit{result}! = \mathit{already\_known}) \\
\hline
\end{array}
$$

As you can see, the effect of the schema $\lor$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its two arguments with the logical connective $\lor$. Similarly, the effect of the schema $\land$ operator is to take the conjunction of the two predicate parts. Any common variables of the two schemas are merged: in this example, the input *name*?, the output *result*!, and the four observations of the state before and after the operation are shared by the two arguments of $\lor$. In order to write *RAddBirthday* as a single schema, it has been necessary to write out explicitly something which was implicitly part of the declaration Ξ*BirthdayBook*, namely that the state doesn't change.

A robust version of the *FindBirthday* operation must be able to report if the input name is not known:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{NotKnown}} \\
\hline
\Xi \mathit{BirthdayBook} \\
\mathit{name}? : \mathit{NAME} \\
\mathit{result}! : \mathit{REPORT} \\
\hline
\mathit{name}? \notin \mathit{known} \\
\mathit{result}! = \mathit{not\_known} \\
\hline
\end{array}
$$

The robust operation either behaves as described by *FindBirthday* and reports success, or reports that the name was not known:

$$\mathit{RFindBirthday} \mathrel{\widehat{=}} (\mathit{FindBirthday} \land \mathit{Success}) \lor \mathit{NotKnown}.$$

The *Remind* operation can be called at any time: it never results in an error, so the robust version need only add the reporting of success:

$$RRemind \mathrel{\widehat=} Remind \land Success.$$

The separation of normal operation from error-handling which we have seen here is the simplest but also the most common kind of modularization possible with the schema calculus. More complex modularizations include *promotion* or *framing* [4], where operations on a single entity – for example, a file – are made into operations on a named entity in a larger system – for example, a named file in a directory. The operations of reading and writing a file might be described by schemas. Separately, another schema might describe the way a file can be accessed in a directory under its name. Putting these two parts together would then result in a specification of operations for reading and writing named files.

Other modularizations are possible: for example, the specification of a system with access restrictions might separate the description of who may call an operation from the description of what the operation actually does. There are also facilities for generic definitions in Z which allow, for example, the notion of resource management to be specified in general, then applied to various aspects of a complex system [2].

Exercise. Specify a robust version of the class register system.

4 From specifications to designs

We have seen how the Z notation can be used to specify software modules, and how the schema calculus allows us to put together the specification of a module from pieces which describe various facets of its function. Now we turn our attention to the techniques used in Z to document the design of a program which implements the specification.

The central idea is to describe the concrete data structures which the program will use to represent the abstract data in the specification, and to derive descriptions of the operations in terms of the concrete data structures. We call this process *data refinement.* Often, a data refinement will allow some of the control structure of the program to be made explicit, and this is achieved by one or more steps of *operation refinement* or *algorithm development.*

For simple systems, it is possible to go from the abstract specification to the final program in one step, a method sometimes called *direct refinement*. In more complex systems, however, there are too many design decisions for them all to be recorded clearly in a single refinement step, and the technique of *deferred refinement* is appropriate. Instead of a finished program, the first refinement step results in a new specification, and this is then subjected to further steps of refinement until a program is at last reached. The result is a sequence of design documents, each describing a small collection of related design decisions. As the details of the data structures are filled in step by step, so more of the control structure can be filled in, leaving certain sub-tasks to be implemented in subsequent refinement steps. These sub-tasks can be made into subroutines in the final program, so the step-wise structure of the development leads to a modular structure in the program.

Program developments are often documented by giving an idealized account of the path from specification to program. In these accounts, the ideas all appear miraculously at the right time, one after another. There are no mistakes, no false starts, no decisions taken which are later revised. Of course, real program developments don't happen like that, and the earlier stages of a development are often revised many times as later stages cast new light on the system. In any case, specifications are seldom written without at least a rough idea of how they might be implemented, and it is very rare to find that something similar hasn't been implemented before.

This doesn't mean that the idealized accounts are worthless, however. They are often the best way of presenting the decisions which have been made and the relationships between them, and such an account can be a valuable piece of documentation, even if it is economical with the true history of the development.

The rest of this article concentrates on data refinement in Z, although the results of the operation refinement which might follow it are shown. Two examples of data refinement are presented. The first shows direct refinement: the birthday book we specified in section 2 is implemented using a pair of arrays. In the second example, deferred refinement is used to show the implementation of a simple checkpoint-restart mechanism. The implementation uses two sub-modules for which specifications in Z are derived as part of the refinement step. This demonstrates the way in which mathematics can help us to explore design decisions at a high level of abstraction.

5 Implementing the birthday book

The specification of the birthday book worked with abstract data structures chosen for their expressive clarity rather than their ability to be directly represented in a computer. In the implementation, the data structures must be chosen with an opposite set of criteria, but they can still be modelled with mathematical data types and documented with schemas.

In our implementation, we choose to represent the birthday book with two arrays, which might be declared by

$$\begin{array}{l} \mathit{names} : \mathbf{array}\ [1\mathinner{\ldotp\ldotp}]\ \mathbf{of}\ \mathit{NAME}; \\ \mathit{dates} : \mathbf{array}\ [1\mathinner{\ldotp\ldotp}]\ \mathbf{of}\ \mathit{DATE}; \end{array}$$

I have made these arrays "infinite" for the sake of simplicity. In a real system development, we would use the schema calculus to specify a limit on the number of entries, with appropriate error reports if the limit is exceeded. Finite arrays could then be used in a more realistic implementation; but for now, this would just be a distraction, so let us pretend that potentially infinite arrays are part of our programming language. We shall, in any case, only use a finite part of them at any time.

These arrays can be modelled mathematically by functions from the set $\mathbb{N}_1$ of strictly positive integers to NAME or DATE:

$$\begin{array}{l} \mathit{names} : \mathbb{N}_1 \longrightarrow \mathit{NAME} \\ \mathit{dates} : \mathbb{N}_1 \longrightarrow \mathit{DATE}. \end{array}$$

The element $\mathit{names}[i]$ of the array is simply the value $\mathit{names}(i)$ of the function, and the assignment $\mathit{names}[i] := v$ is exactly described by the specification

$$\mathit{names}' = \mathit{names} \oplus \{i \mapsto v\}.$$

The right-hand side of this equation is a function which takes the same value as names everywhere except at the argument i, where it takes the value v.

We describe the state space of the program as a schema. There is another variable hwm (for "high water mark"); it shows how much of the arrays is in use.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{BirthdayBook1}} \\
\hline
names : \mathbb{N}_1 \longrightarrow NAME \\
dates : \mathbb{N}_1 \longrightarrow DATE \\
hwm : \mathbb{N} \\
\hline
\forall i, j : 1 \mathrel{..} hwm \bullet \\
\qquad i \neq j \Rightarrow names(i) \neq names(j) \\
\hline
\end{array}
$$

The predicate part of this schema says that there are no repetitions among the elements $names(1), \ldots, names(hwm)$.

The idea of this representation is that each name is linked with the date in the corresponding element of the array *dates*. We can document this by defining another schema *Abs* that defines the *abstraction relation* between the abstract state space *BirthdayBook* and the concrete state space *BirthdayBook*1:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Abs}} \\
\hline
BirthdayBook \\
BirthdayBook1 \\
\hline
known = \{\, i : 1 \mathrel{..} hwm \bullet names(i) \,\} \\
\forall i : 1 \mathrel{..} hwm \bullet \\
\qquad birthday(names(i)) = dates(i) \\
\hline
\end{array}
$$

This schema relates two points of view on the state of the system. The observations involved are both those of the abstract state – *known* and *birthday* – and those of the concrete state – *names*, *dates* and *hwm*. The first predicate says that the set *known* consists of just those names which occur somewhere among $names(1), \ldots, names(hwm)$. The set $\{\, y : S \bullet \ldots y \ldots \}$ contains those values taken by the expression $\ldots y \ldots$ as y takes values in the set S, so *known* contains a name n exactly if $n = names(i)$ for some value of i such that $1 \leq i \leq hwm$. We can write this in symbols with an existential quantifier:

$$n \in known \Leftrightarrow (\exists\, i : 1 \mathrel{..} hwm \bullet n = names(i)).$$

The second predicate says that the birthday for $names(i)$ is the corresponding element $dates(i)$ of the array *dates*.

Several concrete states may represent the same abstract state: in the example, the order of the names and dates in the arrays does not matter, so long as names and dates correspond properly. The order is not used in determining which abstract state is represented by a concrete state, so two states which have the same names and dates in different orders will represent the same abstract state. This is quite usual in data refinement, because efficient representations of data often cannot avoid including superfluous information.

On the other hand, each concrete state represents only one abstract state. This too is the usual situation, because we don't expect to find superfluous information in the abstract state. It does sometimes happen that one concrete state represents several abstract states, but this is often a sign of a badly-written specification that has a bias towards a particular implementation.

Having explained what the concrete state space is, and how concrete states are related to abstract states, we can begin to implement the operations of the specification. To add a new name, we increase *hwm* by one, and fill in the name and date in the arrays:

$$
\begin{array}{|l}
\hline
\textit{AddBirthday}1 \\
\Delta \textit{BirthdayBook}1 \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\hline
\forall i : 1 \mathrel{..} \textit{hwm} \bullet \textit{name}? \neq \textit{names}(i) \\
\\
\textit{hwm}' = \textit{hwm} + 1 \\
\textit{names}' = \textit{names} \oplus \{\textit{hwm}' \mapsto \textit{name}?\} \\
\textit{dates}' = \textit{dates} \oplus \{\textit{hwm}' \mapsto \textit{date}?\} \\
\hline
\end{array}
$$

This schema describes an operation which has the same inputs and outputs as *AddBirthday*, but operates on the concrete instead of the abstract state. It is a correct implementation of *AddBirthday*, because of the following two facts:

1. Whenever *AddBirthday* is legal in some abstract state, the implementation *AddBirthday*1 is legal in any corresponding concrete state.

2. The final state which results from *AddBirthday*1 represents an abstract state which *AddBirthday* could produce.

Let us look at the reasons why these two facts are true. The operation *AddBirthday* is legal exactly if its pre-condition $name? \notin known$ is satisfied. If this is so, the predicate

$$known = \{\, i : 1 \mathrel{..} hwm \bullet names(i) \,\}$$

from *Abs* tells us that *name*? is not one of the elements $names(i)$:

$$\forall\, i : 1 \mathrel{..} hwm \bullet name? \neq names(i).$$

This is the pre-condition of *AddBirthday*1.

To prove the second fact, we need to think about the concrete states before and after an execution of *AddBirthday*1, and the abstract states they represent according to *Abs*. The two concrete states are related by *AddBirthday*1, and we must show that the two abstract states are related as prescribed by *AddBirthday*:

$$birthday' = birthday \cup \{name? \mapsto date?\}.$$

The domains of these two functions are the same, because

$$\begin{aligned}
&\mathrm{dom}\, birthday' \\
&\quad = known' && \text{[invariant after]} \\
&\quad = \{\, i : 1 \mathrel{..} hwm' \bullet names'(i) \,\} && \text{[from } Abs'\text{]} \\
&\quad = \{\, i : 1 \mathrel{..} hwm \bullet names'(i) \,\} \cup \{names'(hwm')\} \\
&\qquad && \text{[since } hwm' = hwm + 1\text{]} \\
&\quad = \{\, i : 1 \mathrel{..} hwm \bullet names(i) \,\} \cup \{name?\} \\
&\qquad && \text{[since } names' = names \oplus \{hwm' \mapsto name?\}\text{]} \\
&\quad = known \cup \{name?\} && \text{[from } Abs\text{]} \\
&\quad = \mathrm{dom}\, birthday \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

There is no change in the part of the arrays which was in use before the operation, so for all i in the range $1 \mathrel{..} hwm$,

$$names'(i) = names(i) \wedge dates'(i) = dates(i).$$

For any i in this range,

$$birthday'(names'(i))$$

$$
\begin{aligned}
&= dates'(i) && \text{[from } Abs'\text{]} \\
&= dates(i) && [dates \text{ unchanged}] \\
&= birthday(names(i)). && \text{[from } Abs\text{]}
\end{aligned}
$$

For the new name, stored at index $hwm' = hwm + 1$,

$$
\begin{aligned}
&birthday'(names'(hwm')) \\
&\quad = dates'(hwm') && \text{[from } Abs'\text{]} \\
&\quad = date?. && \text{[spec. of } AddBirthday1\text{]}
\end{aligned}
$$

So the two functions $birthday'$ and $birthday \cup \{name? \mapsto date?\}$ are equal, and the abstract states before and after the operation are guaranteed to be related as described by $AddBirthday$.

The description of the concrete operation uses only notation which has a direct counterpart in our programming language, so we can translate it directly into a subroutine to perform the operation:

```
procedure AddBirthday(name : NAME; date : DATE);
begin
    hwm := hwm + 1;
    names[hwm] := name;
    dates[hwm] := date
end;
```

The second operation, $FindBirthday$, is implemented by the following operation, again described in terms of the concrete state:

$$
\begin{array}{|l}
\hline
\textit{FindBirthday1} \\
\hline
\Xi BirthdayBook1 \\
name? : NAME \\
date! : DATE \\
\hline
\exists\, i : 1 \,..\, hwm \bullet \\
\qquad name? = names(i) \land date! = dates(i) \\
\hline
\end{array}
$$

The predicate says that there is an index i at which the $names$ array contains the input $name?$, and the output $date!$ is the corresponding element of the array *dates*. For this to be possible, *name*? must in fact appear somewhere in the array *names*: this is the pre-condition of the operation.

Since neither the abstract nor the concrete operation changes the state, there is no need to check that the final concrete state is acceptable, but we need to check that the pre-condition of *FindBirthday*1 is sufficiently liberal, and that the output *date*! is correct. The pre-conditions of the abstract and concrete operations are in fact the same: that the input *name*? is known. The output is correct because for some i, $name? = names(i)$ and $date! = dates(i)$, so

$$
\begin{aligned}
&date! \\
&\quad = dates(i) && \text{[spec. of } FindBirthday1\text{]} \\
&\quad = birthday(names(i)) && \text{[from } Abs\text{]} \\
&\quad = birthday(name?). && \text{[spec. of } FindBirthday1\text{]}
\end{aligned}
$$

The existential quantifier in the description of *FindBirthday*1 leads to a loop in the program code, searching for a suitable value of i:

```
procedure FindBirthday(name : NAME; var date : DATE);
    var i : INTEGER;
begin
    i := 1;
    while names[i] ≠ name do i := i + 1;
    date := dates[i]
end;
```

The operation *Remind* poses a new problem, because its output *cards* is a *set* of names, and cannot be directly represented in the programming language. We can deal with it by introducing a new abstraction relation, showing how it can be represented by an array and an integer:

AbsCards

$$
\begin{array}{|l}
cards : \mathbb{P}\, NAME \\
cardlist : \mathbb{N}_1 \longrightarrow NAME \\
ncards : \mathbb{N} \\
\hline
cards = \{\, i : 1 \mathrel{..} ncards \bullet cardlist(i) \,\}
\end{array}
$$

The concrete operation can now be described: it produces as outputs *cardlist* and *ncards*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Remind1\ }} \\
\Xi BirthdayBook1 \\
today? : DATE \\
cardlist! : \mathbb{N}_1 \longrightarrow NAME \\
ncards! : \mathbb{N} \\
\hline
\{\, i : 1 \mathrel{..} ncards! \bullet cardlist!(i) \,\} \\
\qquad = \{\, j : 1 \mathrel{..} hwm \mid dates(j) = today? \bullet names(j) \,\} \\
\hline
\end{array}
$$

The set on the right-hand side of the equation contains all the names in the *names* array for which the corresponding entry in the *dates* array is *today*?. The program code for *Remind* uses a loop to examine the entries one by one:

```
procedure Remind(today : DATE;
                 var cardlist : array [1 ..] of NAME;
                 var ncards : INTEGER);
    var j : INTEGER;
begin
    ncards := 0; j := 0;
    while j < hwm do begin
        j := j + 1;
        if dates[j] = today then begin
            ncards := ncards + 1;
            cardlist[ncards] := names[j]
        end
    end
end;
```

The initial state of the program has $hwm = 0$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ InitBirthdayBook1\ }} \\
BirthdayBook1 \\
\hline
hwm = 0 \\
\hline
\end{array}
$$

Nothing is said about the initial values of the arrays *names* and *dates*, because they do not matter. If the initial concrete state satisfies this description, and it is related to the initial abstract state by the abstraction schema *Abs*, then

$$\begin{aligned} &known \\ &\qquad = \{\, i : 1 \mathinner{..} hwm \bullet names(i) \,\} && \text{[from } Abs\text{]} \\ &\qquad = \{\, i : 1 \mathinner{..} 0 \bullet names(i) \,\} && \text{[from } InitBirthdayBook1\text{]} \\ &\qquad = \varnothing. && \text{[since } 1 \mathinner{..} 0 = \varnothing\text{]} \end{aligned}$$

so the initial abstract state is as described by *InitBirthdayBook*. This description of the initial concrete state can be used to write a subroutine to initialize our program module:

```
procedure InitBirthdayBook;
begin
    hwm := 0
end;
```

In this direct refinement, we have taken the birthday book specification and in a single step produced a program module which implements it. The relationship between the state of the book as described in the specification and the values of the program variables which represent that state was documented with an abstraction schema, and this allowed descriptions of the operations in terms of the program variables to be be derived. These operations were simple enough to implement immediately, but in a more complex example, rules of operation refinement could be used to check the code against the concrete operation descriptions.

Exercise. Implement the class register you specified earlier. Use two arrays

```
names : array [1 . . ] of NAME;
finished : array [1 . . ] of (Yes, No);
```

Document:

1. The concrete state space.
2. The abstraction relation.
3. The concrete operations.

6 A simple checkpointing scheme

This example shows how refinement techniques can be used at a high level in the design of systems, as well as in detailed programming. What we shall call a *database* is simply a function from addresses (modelled by the set $ADDR$) to pages of data ($PAGE$):

$$DATABASE == ADDR \longrightarrow PAGE.$$

This definition written with the sign $==$ introduces $DATABASE$ as an abbreviation for the set of functions from $ADDR$ to $PAGE$. We shall be looking at a system which – from the user's point of view – contains two versions of a database

$$\begin{array}{|l}
\hline
\textit{CheckSys} \\
working, backup : DATABASE \\
\hline
\end{array}$$

This schema has no predicate part: it specifies that the two observations *working* and *backup* may be any databases at all, and need not be related.

Most operations affect only the working database. For example, it is possible to access the page at a specified address:

$$\begin{array}{|l}
\hline
\textit{Access} \\
\Xi CheckSys \\
a? : ADDR \\
p! : PAGE \\
\hline
p! = working(a?) \\
\hline
\end{array}$$

This operation takes an address $a?$ as input, and produces as its output $p!$ the page stored in the working database at that address. Neither version of the database changes in the operation.

It is also possible to update the working database with a new page:

$$
\begin{array}{|l}
\hline
\textit{Update} \\
\Delta \textit{CheckSys} \\
a? : \textit{ADDR} \\
p? : \textit{PAGE} \\
\hline
\textit{working}' = \textit{working} \oplus \{a? \mapsto p?\} \\
\textit{backup}' = \textit{backup} \\
\hline
\end{array}
$$

In this operation, both an address $a?$ and a page $p?$ are supplied as input, and the working database is updated so that the page $p?$ is now stored at address $a?$. The old contents of the address are lost.

There are two operations involving the back-up database. We can take a copy of the working database: this is the *CheckPoint* operation:

$$
\begin{array}{|l}
\hline
\textit{CheckPoint} \\
\Delta \textit{CheckSys} \\
\hline
\textit{working}' = \textit{working} \\
\textit{backup}' = \textit{working} \\
\hline
\end{array}
$$

We can also restore the working database to the state it had at the last checkpoint:

$$
\begin{array}{|l}
\hline
\textit{Restart} \\
\Delta \textit{CheckSys} \\
\hline
\textit{working}' = \textit{backup} \\
\textit{backup}' = \textit{backup} \\
\hline
\end{array}
$$

This completes the specification of our system, and we can begin to think of how we might implement it. A first idea might be really to keep two copies of the database, so implementing the specification directly. But experience tells us that copying the entire database is an expensive operation, and that if checkpoints are taken frequently, then the computer will spend much more time copying than it does accessing and updating the working database.

The mathematics cannot make observations like this one automatically for us, but by allowing the specification to be expressed precisely and abstractly, mathematical techniques can help the designer to carry out this kind of analysis, perhaps by calling to mind other, similar systems.

A better idea for an implementation might be to keep only one complete copy of the database, together with a record of the changes made since creation of this master copy. The master copy consists of a single database:

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Master}} \\ \hline master : DATABASE \\ \hline \end{array}$$

The record of changes made since the last checkpoint is a *partial function* from addresses to pages: it is partial because we expect that not every page will have been updated since the last checkpoint.

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Changes}} \\ \hline changes : ADDR \nrightarrow PAGE \\ \hline \end{array}$$

The concrete state space is described by putting these two parts together:

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{CheckSys1}} \\ \hline Master \\ Changes \\ \hline \end{array}$$

How does this concrete state space mirror our original abstract view? The master database is what we described as the back-up, and the working database is $master \oplus changes$, the result of updating the master copy with the recorded changes. We can record this relationship with an abstraction schema:

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{AbsDB}} \\ \hline CheckSys \\ CheckSys1 \\ \hline backup = master \\ working = master \oplus changes \\ \hline \end{array}$$

The notation $master \oplus changes$ denotes a function which agrees with *master* everywhere except in the domain of *changes*, where it agrees with *changes*.

How can we implement the four operations? Accessing a page at address $a?$ should return $working(a?) = (master \oplus changes)(a?)$, so a valid specification of $Access1$ is as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Access1 \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Xi CheckSys1 \\
a? : ADDR \\
p! : PAGE \\
\hline
p! = (master \oplus changes)(a?) \\
\hline
\end{array}
$$

But we can do a little better than this: if $a? \in \mathrm{dom}\, changes$, then

$$(master \oplus changes)(a?) = changes(a?),$$

and if $a? \notin \mathrm{dom}\, changes$, then

$$(master \oplus changes)(a?) = master(a?).$$

So we can use operation refinement to develop the operation further; it is implemented by

```
procedure Access(a : ADDR; var p : PAGE);
    var r : RESULT;
begin
    GetChange(a, p, r);
    if r ≠ found then
        ReadMaster(a, p)
end;
```

What are the operations *GetChange* and *ReadMaster*? We need give only their specifications here, and can leave their implementation to a later stage in the development. *GetChange* operates only on the *changes* part of the state; it checks whether a given page is present, returning a report and, if possible, the page itself:

$$
\begin{array}{|l}
\hline
\textit{GetChange} \\
\Xi \textit{Changes} \\
a? : \textit{ADDR} \\
p! : \textit{PAGE} \\
r! : \textit{RESULT} \\
\hline
(a? \in \operatorname{dom} \textit{changes} \land \\
\qquad p! = \textit{changes}(a?) \land \\
\qquad r! = \textit{found}) \lor \\
(a? \notin \operatorname{dom} \textit{changes} \land \\
\qquad r! = \textit{not\_present}) \\
\hline
\end{array}
$$

As you will see, this is a specification which could be structured nicely with the schema $\lor$ operator. The *ReadMaster* operation simply returns a page from the *master* database:

$$
\begin{array}{|l}
\hline
\textit{ReadMaster} \\
\Xi \textit{Master} \\
a? : \textit{ADDR} \\
p! : \textit{PAGE} \\
\hline
p! = \textit{master}(a?) \\
\hline
\end{array}
$$

For the *Update* operation, we want $\textit{backup}' = \textit{backup}$, so

$$\textit{master}' = \textit{backup}' = \textit{backup} = \textit{master}.$$

Also $\textit{working}' = \textit{working} \oplus \{a? \mapsto p?\}$, so we want

$$\textit{master}' \oplus \textit{changes}' = (\textit{master} \oplus \textit{changes}) \oplus \{a? \mapsto p?\}.$$

Luckily, the overriding operator $\oplus$ is associative: it satisfies the law

$$(a \oplus b) \oplus c = a \oplus (b \oplus c).$$

If we let $\textit{changes}' = \textit{changes} \oplus \{a? \mapsto p?\}$, then

$$
\begin{aligned}
&\textit{working}' \\
&\qquad = \textit{working} \oplus \{a? \mapsto p?\} && \text{[spec. of } \textit{Update}\text{]}
\end{aligned}
$$

$$
\begin{aligned}
&= (\mathit{master} \oplus \mathit{changes}) \oplus \{a? \mapsto p?\} && [\text{from } \mathit{AbsDB}] \\
&= \mathit{master} \oplus (\mathit{changes} \oplus \{a? \mapsto p?\}) && [\text{associativity of } \oplus] \\
&= \mathit{master}' \oplus \mathit{changes}'. && [\text{spec. of } \mathit{Update}1]
\end{aligned}
$$

and the abstraction relation is maintained. So the specification for *Update*1 is

$$
\begin{array}{|l}
\hline
\textit{Update}1 \\
\Delta \mathit{CheckSys}1 \\
a? : \mathit{ADDR} \\
p? : \mathit{PAGE} \\
\hline
\mathit{master}' = \mathit{master} \\
\mathit{changes}' = \mathit{changes} \oplus \{a? \mapsto p?\} \\
\hline
\end{array}
$$

This is implemented by an operation *MakeChange* which has the same effect as described here, but operates only on the *Changes* part of the state.

For the *CheckPoint* operation, we want $\mathit{backup}' = \mathit{working}$, so we immediately see that

$$\mathit{master}' = \mathit{backup}' = \mathit{working} = \mathit{master} \oplus \mathit{changes}.$$

We also want $\mathit{working}' = \mathit{working}$, so

$$\mathit{master}' \oplus \mathit{changes}' = \mathit{master} \oplus \mathit{changes} = \mathit{master}'.$$

This equation is solved by setting $\mathit{changes}' = \varnothing$, since the empty function $\varnothing$ is a right identity for $\oplus$, as expressed by the law

$$a \oplus \varnothing = a.$$

So a specification for *Checkpoint*1 is

$$
\begin{array}{|l}
\hline
\textit{CheckPoint}1 \\
\Delta \mathit{CheckSys}1 \\
\hline
\mathit{master}' = \mathit{master} \oplus \mathit{changes} \\
\mathit{changes}' = \varnothing \\
\hline
\end{array}
$$

This can be refined to the code

$$\mathit{MultiWrite}(\mathit{changes});\ \mathit{ResetChanges}$$

where *MultiWrite* carries out the updating of the *master* database, and *ResetChanges* sets *changes* to $\varnothing$.

Finally, for the operation $\mathit{Restart}1$, we have $\mathit{backup}' = \mathit{backup}$, so we need $\mathit{master}' = \mathit{master}$, as for *Update*. Again, we want

$$\mathit{master}' \oplus \mathit{changes}' = \mathit{master}',$$

this time because $\mathit{working}' = \mathit{backup}$, so we choose $\mathit{changes}' = \varnothing$ as before:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Restart}1} \\
\hline
\Delta \mathit{CheckSys}1 \\
\hline
\mathit{master}' = \mathit{master} \\
\mathit{changes}' = \varnothing \\
\hline
\end{array}$$

This can be refined to a simple call to *ResetChanges*.

Now we have found implementations for all the operations of our original specification. In these implementations, we have used two new sets of operations, which we have specified with schemas but not yet implemented. One set, *ReadMaster* and *MultiWrite* operates on the *master* part of the concrete state, and the other, containing *MakeChange*, *GetChange*, and *ResetChanges*, operates only on the *changes* part of the state. The result is two new specifications for what are in effect modules of the system, and in later stages they can be developed independently. Perhaps the *master* function would be represented by an array of pages stored on a disk, and *changes* by a hash table held in main store.

The mathematical method can describe data structures with equal ease, whether they are held in primary or secondary storage. It describes operations in terms of their function, and is indifferent to whether the execution takes microseconds or hours to finish. Of course, the designer must be very closely concerned with the capabilities of the equipment to be used, and it is vital to distinguish primary storage, which though fast has limited capacity, from the slower but larger secondary storage. But we regard it as a strength and not a weakness of the mathematical method that it does not reflect this distinction. By modelling only the functional characteristics of a software module, a mathematical specification technique encourages a healthy *separation of concerns*: it helps the designer to focus his or her attention on functional aspects, and to compare different designs, even if they differ widely in performance.

*   *   *

There has been space in this article for only a sketch of the refinement process which leads from a Z specification to a working program. There has been no space to state explicitly the facts which must be proved to show that a data refinement is valid, and we have only touched on the idea of operation refinement, in which a specification written in the notation of predicate logic is turned into a program written in a programming language. But I hope that an important message has come across.

This message, which might be taken as the creed of a mathematical approach to programming, is that the development of a program and the proof of its correctness are not tasks which should be undertaken separately. It is notoriously difficult to give even an informal proof of correctness for a program which has been developed without such a proof in mind, so rigorously-checked programs and their proofs must be developed together. More importantly, however, the constraints on the development of the program imposed by the need to prove its correctness can help to guide the process of development. In the checkpoint example, we saw how the rules of refinement allowed us to find the descriptions of the concrete operations almost by calculation. This is the real power of mathematical methods: their ability to systematize our understanding of programming and make us more articulate in explaining specifications and programs to others. This is the reason why mathematical methods are becoming more important in every-day programming, even where risks to life and property do not make a formal, mathematical proof of correctness into a moral necessity.

Acknowledgements

The author is grateful to M. A. McMorran, S. Powell and J. B. Wordsworth of IBM United Kingdom Laboratories, and to T. Clement of the University of

Manchester for helpful comments on the paper, and to Oriel College, Oxford and Rank Xerox (UK) Ltd. for financial support.

References

[1] DATE, C. J.: 'An Introduction to Database Systems', third edition (Addison-Wesley, 1981).

[2] FLINN, L. W., and SØRENSEN, I. H.: 'CAVIAR: A Case Study in Specification'. In 'Specification Case Studies', ed. I. J. Hayes (Prentice-Hall International, 1987).

[3] HAYES, I. J.: 'Applying Formal Specification to Software Development in Industry'. In 'Specification Case Studies', ed. I. J. Hayes (Prentice-Hall International, 1987).

[4] MORGAN, C. C., and SUFRIN, B. A.: 'Specification of the UNIX Filing System'. In 'Specification Case Studies', ed. I. J. Hayes (Prentice-Hall International, 1987).

[5] JONES, C. B.: 'Software Development: A Rigorous Approach' (Prentice-Hall International, 1980).

[6] JONES, C. B.: 'Systematic Software Development using VDM' (Prentice-Hall International, 1986).

[7] SPIVEY, J. M.: 'Understanding Z: A Specification Language and its Formal Semantics' (Cambridge University Press, 1987).