This paper appeared in
VDM 90: VDM & Z
Formal Methods in Software Development
edited by D Bjørner, C A R Hoare and H Langmaack
Lecture Notes in Computer Science no 428

This printing contains some corrections to the published version.

Using Z as a Specification Calculus for Object-Oriented Systems

Anthony Hall
Praxis plc

Abstract

One of the useful features of the Z notation is that it offers a calculus for building large specifications from smaller components. So far most Z specifications have followed a single paradigm in which the system as a whole is treated as a state machine and parts of the specification define parts of the state and operations on these parts. A more recent paradigm for system structuring is the *object-oriented* approach, in which the system is divided into objects each of which has its own set of operations. We have built a system based on the object-oriented approach and we have used Z to specify it. This paper reports on the methods we used to develop an object-oriented Z specification, defines some extensions to the Z library which we developed and suggests some conventions and extensions which would help to make such specifications more straightforward.

1 INTRODUCTION

We have recently developed a substantial piece of software using formal specification in Z in conjunction with implementation in an object-oriented language, Objective C. In carrying out this development, we used the object-oriented approach from an early stage to guide our overall system architecture and the structure of the specification. When used in this way, the object-oriented approach leads to a way of thinking about the specification which is different from the way that Z specifications are usually written. We therefore had to develop new conventions for using Z in this context. We found that the combination of an object-oriented approach with formal specification was very effective, and this paper therefore reports on the methods we used so that others who wish to use this combination of techniques can learn from our experience.

1.1 Z

The language Z is a notation for writing specifications. It is based on typed set theory and first order logic. In addition it contains a number of constructs for structuring specifications, notably the *schema*. The use of schemas offers a *calculus* of specifications, whereby specifications of large systems can be built up from smaller parts.

The language has been developed over a number of years largely through case studies and industrial experience, and during this development a number of conventions have been developed for using the schema calculus to build up specifications. Most of these case studies have used a common paradigm: the system has been viewed as a state machine. The schema calculus has been used to build up a model of the internal state in terms of more or less separate components; operations have been defined in terms of changes to the overall state, and again the schema calculus has been used to specify these operations by merging operations on the components of the state.

The existence of this calculus is one of the main distinguishing features of Z and it is enormously important in the practical development of large scale specifications. First, it makes the job of the specifiers easier by allowing them to develop specifications incrementally. Second, it makes large Z specifications relatively easy to read, by structuring them into comprehensible units. Third, it helps with the development of proofs and possibly refinement steps in a structured way.

1.2 The Object-Oriented Approach

The object-oriented approach to system design is becoming increasingly popular, and we believe that this popularity is based on sound technical reasons. The fundamental characteristics of the object-oriented approach are:

- The structure of a system is based on the major objects which are of interest.
- The behaviour of each object is defined as a set of operations whose implementation is internal to the object.
- The behaviour of the system as a whole is defined in terms of the behaviour of its component objects.

In addition, a number of other techniques are commonly associated with the object-oriented approach, although not a necessary part of it. These techniques are primarily concerned with classifying the objects in the system. Typically, objects are grouped into *classes*, where all instances of a class have a common behaviour. Commonly these classes themselves are related to each other in an inheritance hierarchy, whereby members of a particular class have all the properties of the superclass as well as any properties special to their own class.

The object-oriented approach offers one of the most promising ways of structuring a system in a way which increases cohesion within its parts and reduces coupling between them. It is therefore important that the specification should be able to reflect this structuring.

1.3 Z and Object-Orientation

In this section we summarise the similarities and differences between the conventional way of structuring Z specifications and the way that the object-oriented approach would suggest. This leads to our objectives in bringing the techniques together.

The conventional method of building up a Z specification has a lot in common with the object-oriented approach. For example, one of the well known case studies in Z is the CAVIAR system, described in Hayes' book [1]. CAVIAR is a visitor administration system, and comprises a number of subsystems. Each subsystem is concerned with one aspect of visitor administration: there are subsystems for hotel reservations, transport reservations, conference room bookings and so on. Each subsystem is an instantiation of a generic *resource-user* system. The specification proceeds by:

1. Identifying the basic sets with which the specification is concerned. These are things like visitors, meetings, hotel rooms and so on.
2. Identifying the basic operations which the system provides: booking and cancelling hotel rooms, adding visitors to meetings and so on.
3. Dividing the system into subsystems. The criterion for identification of subsystems is not made explicit, but the subsystems which are described are essentially encapsulations of some data and a collection of operations on that data; for example, there is a hotel reservation subsystem which encapsulates the relation between visitors and hotel rooms and has operations to make and cancel hotel reservations.
4. Defining a resource-user system which is *generic* in that it can describe any type of resource and any type of user, and *general* in that it makes no restrictions on the number of resources per user or users per resource.
5. Specialising the resource user system to generate more restrictive subsystems, such as those which only allow one user per resource.
6. Instantiating the resource user subsystems so that they represent particular types of resource (for example a system in which resources are hotel rooms and users are visitors). These instantiations become the subsystems of CAVIAR.
7. Combining the instantiated, specialised subsystems into a complete system by including the subsystem state schemas in the total state schema and including the subsystem operation schemas in the total operation schemas.

These steps are very similar to those which would be taken when using an object-oriented approach.

1. The 'basic sets' of CAVIAR would be basic object classes in an object-oriented approach.
2. As far as possible each operation of the system would be treated as an operation on a single object class.
3. This would lead to the identification of objects corresponding to the subsystems of CAVIAR: an object Hotel-Reservation-Manager might be identified, for example. The CAVIAR notion of 'subsystem' as an encapsulation of data and operations is very similar in this respect to the notion of 'object' in object-oriented systems.

4. The objects would be classified. In the course of this classification, the general resource-user system would be identified as a 'superclass' of the various resource-user classes. The notion of a generic class, instantiatable to refer to particular classes of contained objects, might also be identified. There is no generally accepted object-oriented approach to providing generic classes: some object-oriented languages include this feature, while others use inheritance to achieve the same effect. Meyer [2] discusses the relationship between genericity and inheritance.

5. Inheritance would be used to define the specialised resource-user classes.

6. Instantiations of the classes would be created for each of the subsystems. The particular resources and users would be defined either by generic instantiation or by inheritance.

7. The complete system would be built up by creating instances of the various object classes. This is different from the way that CAVIAR is built up. In CAVIAR, subsystem definitions are simply merged. In an object-oriented approach, the system as a whole would contain an *instance* of each object class. For example, if there were a schema HR-V for the hotel reservation subsystem and a schema M-V for the meeting subsystem, then the CAVIAR style would be to merge these in a composite schema as follows:

   $CAVIAR$
   HR-V
   M-V

   A more object-oriented approach would be to have an object of class HR-V and an object of class M-V:

   $CAVIAR$
   h : ***HR-V***
   m : ***M-V***

Our objective is to develop conventions and, where necessary, new notation so that we have a calculus of specifications which parallels the calculus of implementations which an object-oriented programming language gives us. We want to be able to define classes, and build up specifications of complex systems from simpler specifications. In particular, we want to develop Z analogues of the two main structuring techniques of object-oriented systems:

- the partitioning of the specification into specifications of individual objects, and their subsequent combination;
- the definition of one class of objects in terms of other classes with which it shares behaviour.

In this paper we concentrate on the first of these questions. We develop a method of defining objects and of combining object definitions into a system specification. We will report on the definition of classes and their relationships elsewhere.

1.4 Other work

We believe that this is the first attempt to publish guidelines for the specification of object-oriented systems in a model-oriented style. There are however several related topics which have been addressed.

1.4.1 OBJ

There is a close analogy between the notion of object and the idea of an abstract data type. The specification language OBJ [3] is an example of a language based on object-oriented ideas which uses equational specifications to define the behaviour of objects.

1.4.2 Object-Oriented Process Specification

There is work in progress at the University of Surrey to extend the Z notation to make it more suitable for object-oriented specifications. This work is described in a research report [4]. This work addresses three issues:

1. A revised schema notation which is intended to make it easier to relate the specifications of operations to the objects they operate on.
2. A new semantics for operations which captures the idea of ‘everything else stays the same’, so allowing minimal specifications.
3. A treatment of concurrency.

Our work does not address these questions.

1. We use the standard schema calculus. We would like to have a way of incorporating operations within state schemas, but neither we nor the Surrey group have devised a notation for this.
2. We retain the standard interpretation of Z in which only the normal laws of logic are used in deriving the consequences of a specification.
3. We have not tried to address the important question of concurrency; we assume, as do most object-oriented implementation languages, that message passing in objects is similar to sequential procedure call.

1.4.3 Type theory

Much of the work on the formal semantics of the object-oriented approach has been couched in terms of type theories for object-oriented languages. There is an excellent summary of this work [5]. Our work raises some of the same questions, and in particular we have had to consider the different theories of inheritance. We do not address these issues in this paper.

1.4.4 Smalltalk Library Definition

London and Milsted [6] have published specifications of some of the Smalltalk library. Although we use a style close to that paper, we give more consideration to the explicitly object-oriented aspects of a system or library.

1.5 Contents of other sections

Section 2 describes how we model individual objects and their state. It introduces the notion of object identity, and the encapsulation of the state via a schema definition. It then shows how, from the definition of the state of a single object, we can systematically derive the state of the whole collection of objects. We can use the same techniques to represent relationships between objects, and the model of the whole system state allows us to express constraints on these relationships. The section also discusses possible alternative approaches and their advantages and shortcomings.

Section 3 first shows how an operation on a single object is defined in the normal Z style, as a change of the object's state. It then develops a method of calculating the effect of the operation on some set of objects (in particular the whole system state), given a definition of its effect on a single object. In order to do this, we introduce a notation for combining relational specifications analogous to functional composition. The notation is applied to a number of examples:

- the effect of an operation on the whole system state;
- initialization and object creation;
- operations on arbitrary sets of objects.

Section 4 presents a more extended example of the use of the notation in a practical example of great importance in object-oriented systems, the specification of user interfaces using the *model-view* paradigm.

The final section presents our conclusions. It summarises the methods we have developed and the remaining problems.

2 OBJECTS AND THEIR STATE

2.1 The State of an Object

Our starting point is a method of defining the behaviour of a single object. The most natural way of doing this within the Z style is to treat an object as a state machine. The possible states of an object are defined by a schema. The state contains components whose types are defined in the declaration part of the schema. The relationships between the components are constrained by the predicate part of the schema.

A specification in this form provides a model of the object. The components of the model are used in explaining the effect of operations on the object. The components are not directly accessible to the outside world; the operations give the external interface of the object. In that respect, the components are like instance variables in an object-oriented programming language. However, there are two important differences:

1. Although the state components cannot be accessed by clients of the object directly, they are visible to clients: they form part of the external definition of the object's behaviour. Instance variables, on the other hand, are purely part of the implementation of the object and are not visible in any way to clients of the object.

2. There is no requirement whatsoever that a state component identified in a specification has a corresponding instance variable in the implementation of an object. The normal refinement rules apply: as long as the operations of the object behave in the way required by the specification, the implementation of the object can use data structures completely different from those in the specification.

In this respect the components of an object schema are like the hidden constructs in a COLD-K class definition [7] and the operations on the object are like the exported constructs.

As an example of an object specification we will consider a visitor information system. If we are building a visitor information system, we will be interested in, among other things, visitors. A visitor is therefore considered to be an object, and interesting properties of a visitor are defined by a schema.

Visitor_1

$$name : NAME$$
$$address : ADDRESS$$
$$car : REGISTRATION$$

The schema *Visitor_1* can be considered to define the class of all visitors. Each visitor is an instance of this schema.

2.2 Identity

This simple scheme captures the notion that visitors have particular properties at any time, but it does not adequately represent the notion of a visitor as an object. An instance of schema *Visitor_1* is a collection of values of name, address and car, but it does not capture the notion that a visitor is an object in its own right which persists regardless of changes in these values. When we operate on a visitor we consider that in some sense the visitor after our operation is the same object as the visitor before. Anyone can change their car, move house, and even change their name; but they remain the same person. To capture this idea, we introduce the notion of an *object identity*. Every object has a property, called its identity, which once and for all distinguishes it from any other object and at the same time allows us to talk about an object being the 'same' after an operation.

In our approach, we introduce a set to represent the identities of all possible visitors, and associate each visitor with their identity. As a convention, we use upper case letters for the names of sets of identities. For example, the given set VISITOR will be used for visitor identities. An obvious way of associating visitors with identities is to include the identity of each visitor as one of its properties. Following object-oriented conventions, we can choose to call this property *self*:

$$
\begin{array}{|l}
\hline
Visitor \\
\hline
self : VISITOR \\
name : NAME \\
address : ADDRESS \\
car : REGISTRATION \\
\hline
\end{array}
$$

We capture the idea that the identities never change by defining an operation schema which leaves *self* unchanged. We reserve the delta convention to have its default meaning of *any* change in state, and define a schema called VisitorOp which we will incorporate into every operation on visitors:

$$
\begin{array}{|l}
\hline
VisitorOp \\
\hline
\Delta Visitor \\
\hline
self' = self \\
\hline
\end{array}
$$

This use of identities distinguishes an object based approach from a value based approach.

There are many benefits from having identities. Codd, in his extended relational model [8] discusses how they can avoid many difficulties in conventional database systems. There is a discussion of identities in an object-oriented context in the proceedings of OOPSLA 86 [9].

We see below that identities are essential for referring to objects from within other objects and in operation definitions.

2.3 The system state

If we model each object as a state machine, we can model the state of the whole system as a collection of objects. For example a system containing just visitors is simply a set of visitor objects.

$$
\begin{array}{|l}
\hline
VisitorSystem\_1 \\
visitors : \mathbb{P}\ Visitor \\
\hline
\end{array}
$$

There is a constraint that all the visitors in the system have different identities. We could write this directly, but we find it convenient later to express it in another way. We introduce a derived variable into the state: a function from identities to objects. This function is defined as the set of (*identity, object*) pairs where the *identity* is the identity in the object's state. The fact that this is a function guarantees that no two objects in the state have the same identity.

$$
\begin{array}{|l}
\hline
VisitorSystem \\
visitors : \mathbb{P}\ Visitor \\
idVisitor : VISITOR \pfun Visitor \\
\hline
idVisitor = \{v : visitors \bullet v.self \mapsto v\} \\
\hline
\end{array}
$$

This definition of the system can be derived uniformly for each kind of object. It is used later in defining relationships between objects and in interpreting operation definitions. Because it is a systematic transformation of the schema defining the object, we could envisage introducing a convention or a schema calculus operator to define it. For example we could adopt the convention that for any object schema *Thing*, the schema $*Thing* denoted the corresponding system state, so that in our example $*Visitor* would denote the schema we have called *VisitorSystem*.

2.4 Relationships Between Objects

2.4.1 Representing Relationships

Up to now we have assumed that the components of objects have simple types and are not themselves objects. However, most of the interesting properties of systems arise from relationships between objects. We therefore need to have a systematic way of representing such relationships. There are two reasonable ways which are appropriate in different circumstances.

1. We can have a property of one object whose value is the related object or set of objects.

2. We can have a new kind of object which itself represents the relationship.

Consider two examples: the association between meetings and the visitors who are attending them, and the association between visitors and the organisations to which they belong. Visitors, meetings and organisations are all represented by schemas similar to *Visitor*.

As an example of the first method, we might represent the organisation to which a visitor belongs as a property, say *affiliation*, of a visitor [1].

$Visitor$
$self : VISITOR$
$name : NAME$
$address : ADDRESS$
$car : REGISTRATION$
$affiliation : ORGANISATION$

We recommend this method where, from the point of view of the system being specified, only one of the object types is of major interest. In this case it is unlikely that the visitor information system will do anything with the affiliation of a visitor other than record it and possibly use it for some sort of selective retrieval. The relationship itself is not of primary concern to the system.

The second method would be more suitable for a relationship which is of primary concern to the system. Clearly one of the major functions of a visitor system is to manage the relationship between visitors and meetings. We would therefore introduce an object to represent this: a meeting - visitor subsystem. There are likely to be important operations on this object.

$MeetingVisitor$
$self : MV$
$mv : MEETING \leftrightarrow VISITOR$

(Note that for the sake of this example we are simplifying by ignoring time and any constraints on the number of meetings a visitor may attend.)

2.4.2 Use of identities

In both these methods we have to represent *references* from one object to another: in the first case from visitors to their organisations, and in the second from *MeetingVisitor* to meetings and visitors. These references use object identities. They must not use object values - it would be wrong, for example, to write

$MeetingVisitor$
$self : MV$
$mv : Meeting \leftrightarrow Visitor$

[1] From now on we will use this extended definition of Visitor.

where *Visitor* is the schema describing the state of a visitor. For consider what would happen if we changed some detail about a visitor: the effect would have to propagate to any object, like *MeetingVisitor*, which referred to it.

This issue reflects an important difference between objects and instances of abstract data types. If *MeetingVisitor* were an abstract data type then it would completely hide the component visitors and meetings and there is no sense in which one could "change" a visitor in *MeetingVisitor* unless *MeetingVisitor* itself provided an interface to do so. In that case, it would be quite satisfactory to model the internal structure of *MeetingVisitor* by the *values* of its components. However, such an approach would be contrary to the separation of concerns which the object-oriented method supports. It would mean that *MeetingVisitor* would have to know *everything* about the properties of visitors, so as to offer an interface to them. In the object-oriented method, the opposite is true: *MeetingVisitor* knows *nothing* about visitors except that they attend meetings.

2.4.3 Existence of objects in the system

There is a constraint that must be satisfied by the references from one object to another, whichever style is adopted. Clearly, the referred-to object must exist in the system. To express this constraint we need the state of the whole system. For example there is an implied constraint in our definition of Visitor, as follows:

$System$

$VisitorSystem$
$OrganisationSystem$
$\ldots$

$\forall\, v : visitors \bullet v.affiliation \in \mathrm{dom}\, idOrganisation$

Some object-oriented systems have a converse property: that the *only* objects which exist in the system are those which are referred to by other objects or directly accessible by the user. The effect of deleting the last reference to an object is that the object itself disappears.

For example, we might make the recording of an organisation dependent on there being some visitor affiliated to that organisation in the system.

$DependentSystem$

$System$

$\mathrm{dom}\, idOrganisation = \{v : visitors \bullet v.affiliation\}$

2.5 Alternative approaches

Although intuitively appealing, modelling object identities by a value in the object's state is not the only method that could be chosen. One could use a more conventional

Z style and model properties of objects as functions from identities to property values. For example we could have specified *VisitorSystem* like this:

$$
\begin{array}{|l}
\hline
VisitorSystem \\
visitors : \mathbb{P}\ VISITOR \\
name : VISITOR \pfun NAME \\
address : VISITOR \pfun ADDRESS \\
car : VISITOR \pfun REGISTRATION \\
\hline
\end{array}
$$

This method is similar to that used in COLD-K [7] to define classes. The disadvantage of this method is that it is more cluttered than our formulation and that it does not so clearly encapsulate the properties of visitors.

The method does have some advantages. In particular, it avoids a problem we have with optional properties. In our formulation we have a difficulty with visitors who do not, for example, have cars. The most obvious way out is to define the type REGISTRATION to include a *nil* value, for example like this:

$$REGISTRATION ::= nil \mid r\langle\langle NUMBER \rangle\rangle$$

This works and is what object-oriented practitioners would expect, but introduces extra clutter when we come to use values of type NUMBER. In the whole-state approach, however, the problem disappears since we can simply omit visitors who do not have cars from the *car* function. Optional properties have domains which are subsets of the objects in the system, while mandatory properties have domains which are equal to the objects in the system. For example we could expand *VisitorSystem* as follows:

$$
\begin{array}{|l}
\hline
VisitorSystem \\
visitors : \mathbb{P}\ VISITOR \\
name : VISITOR \pfun NAME \\
address : VISITOR \pfun ADDRESS \\
car : VISITOR \pfun REGISTRATION \\
\hline
\mathrm{dom}\ name = visitors \\
\mathrm{dom}\ address = visitors \\
\mathrm{dom}\ car \subseteq visitors \\
\hline
\end{array}
$$

This small advantage is not enough to outweigh the intuitive appeal of encapsulating each object's state, and its closer parallel with object-oriented thinking.

2.6 Conventions

In what follows we will follow the convention in Section 2.2 above and assume that there is, for each kind of object, a schema defining the set of all objects of that kind in the system. The system as a whole is the conjunction of these schemas. Our example will have the state schema:

$System$

$VisitorSystem$
$MeetingSystem$
.....

At this stage we do not specify all of the different kinds of objects. Each of the contained schemas is similar to VisitorSystem defined above.

3 OPERATIONS

3.1 Operations on Single Objects

The conventional way of defining operations in Z is to define a change of state in terms of the values of components before and after the operation, and we can use this method directly. We have already mentioned that we need to ensure that operations on an object do not change its identity, and this can simply be expressed by convention.

The simplest way to define an operation is to define its effect on a single object. For example, there may be an operation to change the address of a visitor.

$ChangeAddress$

$VisitorOp$
$address? : ADDRESS$

$address' = address?$

$name' = name$
$car' = car$
$affiliation' = affiliation$

Note that this operation is deterministic: the state after the operation is completely defined by the state before and the input parameter. It is of course also common in Z to give non-deterministic specifications, where the system state after is not totally determined. For example we might want to specify a non-deterministic form of *ChangeAddress*, call it *NewAddress*, where the new values of name, car and affiliation were not specified; such a specification could then be combined with other partial specifications, for example *NewAffiliation*, to give a composite operation.

$NewAddress$

$VisitorOp$
$address? : ADDRESS$

$address' = address?$

3.2 Encapsulation

One of the characteristics of the object-oriented approach is that operations are considered part of the definition of the class. Z has no notation to collect together operations like this, so we simply assume by convention that the operations described are all those which are provided.

The operations on an object, including its enquiry operations, are the *only* way that other objects can discover anything about it. There is no need, therefore, for the components of an object to relate directly to its observable properties. They are used solely to explain what the effects of the operation are. This means that our semantics for an object are given by allowable traces over its operations, in a similar way to Schuman, Pitt and Byers [4]. This also means that the implementation of objects does not have to provide the components described in their schemas, provided that there is observational equivalence between the implemented operations and the abstract operation definitions. The normal refinement rules for Z [10] therefore apply to our specifications.

3.3 Calculating the Effect on the System

Although we define *ChangeAddress* as an operation on a single visitor, there is an implicit effect on the system. We can derive the effect from the definition on a single object. All other objects in the system are left unchanged; the mapping of identifiers to object states is simply changed to reflect the new state of the one object affected.

This is an example of a much more general problem: given the specification of some operation on a single object, we wish to use this specification to define the effect on some set of objects. We have developed a method of doing this calculation systematically. The method consists of three steps:

1. Determine which objects are affected by the operation.
2. Calculate the possible sets of final states each affected object might have.
3. Calculate the set of possible system states which arise from each object having one of its possible final states.

Variants of the method can be used depending on how each step is carried out.

This section uses the example of ChangeAddress and NewAddress to develop the method. In doing this development we will point out problems with various alternative formulations which we tried, and which work in limited circumstances. Subsequent sections show how the final method can be applied in different contexts.

If only one object is affected, the affected object can be defined by including its state change schema explicitly. For example one way of describing the effect of *ChangeAddress* on the system would be to include the operation schema in a larger state change schema as follows:

$$
\begin{array}{|l}
\hline
ChangeAddressSystem \\
\hline
ChangeAddress \\
\Delta VisitorSystem \\
\hline
idVisitor' = idVisitor \oplus \{self \mapsto \theta Visitor'\} \\
\hline
\end{array}
$$

This is very similar to the normal Z method of promoting an operation schema to work in some larger state; however it differs in that the operation is not on a part of the system, but on an object of which the system contains an instance.

This formulation is correct whether or not ChangeAddress is deterministic. Unfortunately it does not generalise easily to the case of operations on several objects, because of the inclusion of the schema ChangeAddress. We cannot have more than one instance of the operation if we use schema inclusion like this. We have therefore developed an alternative way of expressing the effect which is more generally applicable. To explain the more general form, we start by considering the simple case where the operation definition is deterministic. Instead of including the ChangeAddress Schema in the signature, we can use it as part of the predicate. We define which object is changed by giving its identity, and extract the before state from the identity to state mapping. There is then a determinate final state given by the operation schema, which we use to give a new value for the mapping.

$$
\begin{array}{|l}
\hline
ChangeAddressSystem \\
\hline
\Delta VisitorSystem \\
v? : VISITOR \\
visitorAddress? : ADDRESS \\
\hline
idVisitor' = idVisitor \oplus \{v? \mapsto \\
\quad (\mu\, ChangeAddress \mid \\
\qquad \theta Visitor = idVisitor\, v? \land \\
\qquad address? = visitorAddress? \\
\qquad \bullet\, \theta Visitor')\} \\
\hline
\end{array}
$$

Note that there is a naming problem: we cannot use the same name for the address parameter in the outer operation, since this name is hidden by the redefinition in ChangeAddress within the set comprehension. A more systematic way of dealing with this problem would be to decorate the included operation schema:

$$
\begin{array}{|l}
\hline
ChangeAddressSystem \\
\Delta VisitorSystem \\
v? : VISITOR \\
address? : ADDRESS \\
\hline
idVisitor' = idVisitor \oplus \{ v? \mapsto \\
\quad (\mu\ ChangeAddress_1 \mid \\
\qquad \theta Visitor_1 = idVisitor\ v? \land \\
\qquad address?_1 = address? \\
\qquad \bullet\ \theta Visitor_1')\} \\
\hline
\end{array}
$$

This formulation relies on ChangeAddress being deterministic. We are regarding it as a function from before to after states. To make this more obvious, we can define the function that an operation schema represents explicitly. In general, a deterministic operation schema defines a function from input parameters to a function from before states to after states:

$$
\begin{array}{|l}
changeAddressF : ADDRESS \longrightarrow Visitor \nrightarrow Visitor \\
\hline
changeAddressF = \\
\quad \{ a : ADDRESS \bullet a \mapsto \\
\qquad \{ ChangeAddress \mid address? = a \\
\qquad\quad \bullet\ \theta Visitor \mapsto \theta Visitor' \\
\qquad \} \\
\quad \}
\end{array}
$$

What we are doing is composing this function with selected mappings from idVisitor (in this case just one) and then overwriting idVisitor with this new function. Using this we can give an equivalent formulation of ChangeAddressSystem:

$$
\begin{array}{|l}
\hline
ChangeAddressSystem \\
\Delta VisitorSystem \\
v? : VISITOR \\
address? : ADDRESS \\
\hline
idVisitor' = idVisitor \oplus \\
\quad ((\{ v? \} \lhd idVisitor) \fatsemi (changeAddressF\ address?)) \\
\hline
\end{array}
$$

If the operation is not deterministic, we cannot use function overriding in this simple way , since the operation represents a relation rather than a function. Suppose that we use the non-deterministic version of *ChangeAddress*, that is *NewAddress*. Like any operation schema, this generates a function from input parameters to a relation between before and after states.

$$
\begin{array}{|l}
newAddressR : ADDRESS \longrightarrow Visitor \leftrightarrow Visitor \\
\hline
newAddressR = \\
\quad \{ a : ADDRESS \bullet a \mapsto \\
\quad\quad \{ NewAddress \mid address? = a \\
\quad\quad\quad \bullet \theta Visitor \mapsto \theta Visitor' \\
\quad\quad \} \\
\quad \}
\end{array}
$$

We will often find it useful to represent operations in this form. The transformation from the schema representation is purely mechanical, although because the types involved are variable it cannot be defined directly in Z. We would therefore find it useful to have a convention or a schema calculus operation to denote this transformation. For example for any operation schema Op which had inputs of type INPUT and transformed a state of type State, we could define the expression $\mathbb{R}Op$, to mean the corresponding function of signature

$INPUT \longrightarrow State \leftrightarrow State$

In our example, $\mathbb{R}$*NewAddress* would denote the function that we have called *newAddressR*.

If we try to use *newAddressR* to define the effect of NewAddress on the system state, we do not immediately have such a convenient notation. When we compose $\{ v? \} \lhd idVisitor$ with $(newAddressR\ address?)$ we end up with a relation from visitor identity to visitor state. We know that the final system state contains just one pair from that relation - but we cannot say which one. More generally, we have the same problem when we operate on *sets* of objects with a non-deterministic operation. We then have a relation between identities and visitor states, and we know that the final system state contains exactly one pair for each value of the identity. In other words the final system state is formed by overwriting with a function from identities to visitor states which:

a has the same domain as the relation

b is a subset of the relation

For each relation there is a set of such functions. The best we can do is to define this set of functions, and state in our specification that the final system state is derived from a member of that set.

We define a generic function called *functions* which produces this set of functions from any relation.

$$
\begin{array}{|l}
[I, X] \\
\hline
functions : (I \leftrightarrow X) \longrightarrow \mathbb{P}(I \pfun X) \\
\hline
functions = (\lambda\, r : I \leftrightarrow X \bullet \\
\quad \{ f : I \pfun X \mid \mathrm{dom} f = \mathrm{dom}\, r \land f \subseteq r \})
\end{array}
$$

This operator *functions* can be used to define a new operation $\oplus_{rel}$ which is the relational analogue of functional overriding. The effect of $\oplus_{rel}$ is best explained by

analogy with normal functional overriding. The operator $\oplus$ is a function from two functions to a third. We typically use it in expressions like

$f = f_1 \oplus f_2$

In contrast, $\oplus_{rel}$ is a function from a function and a relation to a set of functions, and is used in expressions like

$f \in f_1 \oplus_{rel} r$

It thus 'transmits' the non-determinism of the relational specification into the final operation. The formal definition of $\oplus_{rel}$ is as follows:

$$[I, X]$$

$$\_ \oplus_{rel} \_ : (I \pfun X) \times (I \rel X) \fun \power (I \pfun X)$$

$$\forall f : I \pfun X; r : I \rel X \bullet$$
$$f \oplus_{rel} r = \{f' : functions\ r \bullet f \oplus f'\}$$

Thus $f \oplus_{rel} r$ returns the set of functions each of which could be obtained by overriding f with one of the functions corresponding to the relation r.

We can now use $\oplus_{rel}$ to give a completely general formulation of the effect on the system state of a non-deterministic operation on an object. For example

NewAddressSystem

$$\Delta VisitorSystem$$
$$v? : VISITOR$$
$$address? : ADDRESS$$

$$idVisitor' \in idVisitor \oplus_{rel}$$
$$((\{v?\} \domres idVisitor) \fatsemi (newAddressR\ address?))$$

As an alternative to naming and defining the relation *newAddressR*, we could include an equivalent expression in the specification of *NewAddressSystem*.

NewAddressSystem

$$\Delta VisitorSystem$$
$$v? : VISITOR$$
$$newAddress? : ADDRESS$$

$$idVisitor' \in idVisitor \oplus_{rel}$$
$$\{NewAddress \mid \theta Visitor = idVisitor\ v? \land$$
$$address? = newAddress?$$
$$\bullet\ v? \mapsto \theta Visitor'\}$$

Again, if we use this method we have to be careful of name clashes between the parameters of the operation being defined and those of the operation being used in the definition.

The use of $\oplus_{rel}$ allows us to calculate, in a systematic way, system specifications from object specifications. The non-determinism of the specifications is preserved.

The precondition of the single object operation plays an important role in this method. It selects the objects which are affected by the operation by defining the domain of the relation. We therefore have a proof obligation: to show that, for all the objects we intend to affect, the precondition is in fact satisfied. If not, then our operation on the state will leave the object unaffected, whereas to be consistent with normal Z the operation ought to be undefined in these circumstances. The importance of the precondition is apparent in the laws which follow.

This technique is generally applicable wherever we need to define the effect of an operation on some set of objects. Although we have introduced it by discussing the effect on the total system of single object operations, there are other applications. These are illustrated in the following subsections.

The composition of specifications in this way has some desirable properties. The operator *functions*, and hence $\oplus_{rel}$, obeys restricted forms of the distribution law over intersection, union and difference, and hence we can deduce some properties of operations defined by conjunction, disjunction and negation.

The most useful law is as follows:

$$a, b : I \leftrightarrow X \mid \mathrm{dom}\, a = \mathrm{dom}\, b = \mathrm{dom}(a \cup b)$$
$$\vdash$$
$$functions(a \cup b) = functions\ a \cup functions\ b$$

This tells you how to calculate the effect of applying an operation defined by schema conjunction (or inclusion). Suppose there is an object operation C defined as

$C \mathrel{\widehat{=}} A \wedge B$

The effect on the system of applying C is the conjunction of the effects of A and B, provided that the same objects are affected and that for all affected objects, the preconditions of both A and B are met.

The other two laws are not as strong: the equality is replaced by a subset relationship. If C is defined by disjunction

$C \mathrel{\widehat{=}} A \vee B$

then the result of C may be that of A or of B, but it could also be some combination of them - for example applying A to one object and B to another. Only if there is exactly one object affected does the equality hold. Formally:

$$a, b : I \leftrightarrow X \mid \mathrm{dom}\, a = \mathrm{dom}\, b$$
$$\vdash$$
$$functions\ a \cup functions\ b \subseteq functions(a \cup b)$$

Finally, if an object operation is defined by negation

$B \mathrel{\widehat{=}} \neg A$

then the effect on the system is one of the possible states which could be obtained from an arbitrary operation on the affected objects, excluding those which could be obtained by applying A to the affected objects. Again, in general not all such states are

possible since they include states where A is applied to *some* of the relevant objects; in the special case where only one object is affected, the subset becomes an equality.

$$a, b : I \leftrightarrow X \mid \mathrm{dom}\, a = \mathrm{dom}\, b = \mathrm{dom}(a \setminus b)$$
$$\vdash$$
$$functions(a \setminus b) \subseteq functions\ a \setminus functions\ b$$

3.4 Example 1: Initialization

In Z the initial state of a system is defined by a predicate constraining the state variables. In object-oriented terms we are defining the effect of a special kind of operation, to create a *new* object. Again we can consider the state of the created object, but we should also consider the constraints imposed by the system as a whole. Suppose that, to initialize a Visitor, we give values of the name and affiliation and leave the car and address unset (in this case by using the special value *nil*).

InitVisitor

$$Visitor$$
$$name? : NAME$$
$$affiliation? : ORGANISATION$$

$$name = name?$$
$$affiliation = affiliation?$$
$$address = nil$$
$$car = nil$$

This fails to capture two important facts, however:

1. After object creation, the object exists in the system.
2. The identity chosen for the new object is distinct from all other identities in the system.

These are examples of properties that are not expressible without an explicit model of the state of the whole system. Creation is in essence an operation on the system as a whole. So, of course, is deletion. Again, in the simple case we can express the effect as follows:

CreateVisitor_1

$$InitVisitor$$
$$\Delta VisitorSystem$$

$$self \notin \mathrm{dom}\, idVisitor$$
$$idVisitor' = idVisitor \cup \{self \mapsto \theta Visitor\}$$

Once again, since this pattern is the same for all object creation, it may not be necessary to write out the specification in detail; it may be enough to define InitVisitor. However, if visitors are created as part of other operations we will need to express the effect on the whole system. We can calculate the effect of InitVisitor using a variant of the general method. The selection of what visitors are created is presumably some function of the larger operation; for definiteness, let us suppose it determines a particular set of names for a single organisation and that exactly one visitor is created for each name. Then there must be just one visitor with each given name and affiliation created, but we do not know what identity each will have. We arbitrarily assign an identity for each name. The state of each new object is determined by the initialization operation with the given name and organization as parameters. The new state of the whole system has the new objects added to *idVisitor*.

$$
\begin{array}{l}
\textit{CreateVisitor} \\
\hline
\Delta \textit{VisitorSystem} \\
\textit{names}? : \mathbb{P}\, \textit{NAME} \\
\textit{organisation}? : \textit{ORGANISATION} \\
\hline
\exists f : \textit{VISITOR} \rightarrowtail\!\!\!\!\!\!+\; \textit{names}? \mid \operatorname{dom} f \cap \operatorname{dom} \textit{idVisitor} = \varnothing \bullet \\
\textit{idVisitor}' \in \textit{idVisitor} \oplus_{rel} \\
\quad \{\textit{InitVisitor} \mid \\
\quad\quad \textit{self} \in \operatorname{dom} f \land \textit{name}? = f(\textit{self}) \land \\
\quad\quad \textit{affiliation}? = \textit{organisation}? \bullet \\
\quad\quad\quad \textit{self} \mapsto \theta\, \textit{Visitor}\}
\end{array}
$$

3.5 Example 2: Operations on sets of objects

The technique we have described above extends quite naturally to defining the effect of operations which change sets of objects. For example, suppose that we wish to change the addresses of all visitors who belong to a particular organisation. The steps are:

1. Select the required objects by choosing visitors of the right affiliation.
2. Calculate the allowed final states of these objects using the relation corresponding to *NewAddress* in its equivalent form *newAddressR* applied to the given address.
3. Use the $\oplus_{rel}$ operator to generate the corresponding functions and overwrite the *idVisitor* component with the new functions.

$$
\begin{array}{|l}
\hline
ChangeOrganisationAddress \\
\Delta VisitorSystem \\
organisation? : ORGANISATION \\
address? : ADDRESS \\
\hline
idVisitor' \in idVisitor \oplus_{rel} \\
\quad ((idVisitor \rhd \{ Visitor \mid affiliation = organisation? \}) \fatsemi \\
\qquad (newAddressR\ address?) \\
\quad) \\
\hline
\end{array}
$$

4 MODELS AND VIEWS

4.1 Overview

As a larger example of an application of these ideas, we show how to define a system of *models* and *views*. One of the techniques used in object-oriented development is to isolate the behaviour of objects from their representation to the user. To do this, each real-world object is represented by a pair of implementation-level objects. One, called the *model* object, represents the intrinsic structure and behaviour of the real-world object. The other, called the *view*, represents the appearance of the object to the computer user. In fact, a single model object may have several views of the same type, and may even have several types of view.

This method of isolating the essential behaviour in the model object, and connecting the model to one or more views, is used in many systems and is described in some detail by Brad Cox [11].

It is usually best to specify the model object first, since this is the conceptual centre of the specification. Having specified the model, we then have to specify one or more views for the model and also specify how the states of the views are related to the states of the model. In doing this it is important to recognise that the view and the model are each encapsulated objects with their own states and operations. They are not part of a single object and they do not, for example, share instance variables. The view and the model only communicate with each other via messages, like any other objects. This means that we should not write invariants which relate the internal state of the views to the internal state of the model. Such invariants would not, in fact, be maintained at all times. Typically, the protocol is as follows:

1. A user carries out several manipulations on a view.

2. Only when the user commits the changes are the changes communicated to the model object. This is done by sending the model messages to update its state.

3. The model adjusts its state according to these messages. It then transmits the fact that it has changed to any other views that it may have.

4.2 The State of the System

To model this state of affairs, we must record what views are associated with what models in the system. Then for each view operation, we must define what the corresponding model operation is. Finally, we must define the converse: for each change in the model, what is the corresponding change in the associated views. We do not, however, record any direct relationships between the *states* of the models and views.

The state of the overall system, therefore, contains the sets of models, the sets of views, and the relationship between them. We can simplify this if we assume that the identities of models are all of one type [MODEL], and those of views are of one type [VIEW]. Each view is a view of only one model; there is therefore a function from VIEW to MODEL. Its inverse is not necessarily functional, however, since a model may have several views.

$$
\begin{array}{|l}
\hline
ModelView \\
models : \mathbb{P}\ MODEL \\
views : \mathbb{P}\ VIEW \\
viewModel : VIEW \pfun MODEL \\
\hline
\mathrm{dom}\ viewModel = views \\
\mathrm{ran}\ viewModel \subseteq models \\
\hline
\end{array}
$$

For example, consider a view of a visitor. A visitor view might have the state:

$$
\begin{array}{|l}
\hline
VisitorNameAddressView \\
selfV : VIEW \\
nameV : TEXT \\
addressV : TEXT \\
\hline
\end{array}
$$

For simplicity we consider a restricted view which only displays (and allows editing of) the name and address. In the view the name and address are in the concrete form of text; we assume that there are total functions which convert names and addresses to text, and partial functions which convert text to names and addresses. The partial functions implement any checking which may be needed to validate names and addresses, in the sense that their domains define what are valid input texts.

$$
\begin{array}{|l}
nameText : NAME \fun TEXT \\
addressText : ADDRESS \fun TEXT \\
textName : TEXT \pfun NAME \\
textAddress : TEXT \pfun ADDRESS
\end{array}
$$

The portion of the state we are interested in will be ModelView, plus the mappings from identities to states for visitor models and visitor views.

$$
\begin{array}{|l}
\hline
VisitorModelView \\
\hline
ModelView \\
idVisitor : VISITOR \pfun Visitor \\
idVisitorView : VISITORVIEW \pfun VisitorNameAddressView \\
\hline
\end{array}
$$

Note that we assume for simplicity that $VISITOR \subseteq MODEL$ and $VISITORVIEW \subseteq VIEW$

4.3 Initialization

The initial state of VisitorModelView is probably empty. More significant is the initial state of a view when it is first created. Since a view cannot exist without a corresponding model, the view is created with a particular model in mind and either common parameters are used in initializing the model and its first view, or if the model already exists then its state is used in initializing the view. The definition of view initialization, therefore, must define the way that the state (current or intended) of the model is communicated to the view.

Typically we would initialize a visitor view from an existing model of a visitor. To do this, we would need to assume that the visitor model had operations to return its name and address. Suppose these are GetName and GetAddress, with the obvious behaviour:

$$
\begin{array}{|l}
\hline
GetName \\
\hline
\Xi Visitor \\
name! : NAME \\
\hline
name! = name \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
GetAddress \\
\hline
\Xi Visitor \\
address! : ADDRESS \\
\hline
address! = address \\
\hline
\end{array}
$$

Now we can define the operation of initializing a view.

$$
\begin{array}{|l}
\hline
InitVisitorNameAddressView \\
\hline
VisitorNameAddressView \\
model? : Visitor \\
\hline
nameV \in nameText(\!|\{ GetName \mid \theta Visitor = model? \bullet name! \}|\!) \\
addressV \in \\
\quad addressText(\!|\{ GetAddress \mid \theta Visitor = model? \bullet address! \}|\!) \\
\hline
\end{array}
$$

Here we use the operations defined on the model to determine what parameters are passed to the view. The formulation given is general: it does not require the operations to be functional, although in this case they happen to be so.

The effect on the state is to create a new view, initialize it as above, and make it a view of the cited model.

$$
\begin{array}{|l}
\hline
\mathit{CreateVisitorNameAddressView} \\
\Delta \mathit{VisitorModelView} \\
m? : \mathit{MODEL} \\
\hline
m? \in \mathit{models} \\
\exists\, v : \mathit{VISITORVIEW} \bullet \\
\quad v \notin \mathit{views} \land \\
\quad \mathit{idVisitorView}' \in \mathit{idVisitorView} \oplus_{rel} \\
\qquad \{\mathit{InitVisitorNameAddressView} \mid \\
\qquad\quad \mathit{selfV} = v \land \\
\qquad\quad \mathit{model}? = \mathit{idVisitor}\, m? \\
\qquad\quad \bullet\ \mathit{selfV} \mapsto \theta\, \mathit{VisitorNameAddressView}\} \\
\quad \land \\
\quad \mathit{viewModel}' = \mathit{viewModel} \oplus \{v \mapsto m?\} \\
\hline
\end{array}
$$

This is analogous to other operations in that the effect on the system follows systematically from the operation on the object concerned. The new view is given an identity which is not already used; the function from view identities to views has the new view added and the new view is connected to the model of which it is intended to be a view.

4.4 Updating via views

Now consider the effect of editing the address in one of the views of a visitor. Two things must happen: the address in the model must change to reflect the new value; then this change must be propagated to any other views of the model. The editing operation itself can be represented as the input of a new text; we consider only the case where it is a valid address. The check for this might be delegated to the model itself; we do not consider that case here.

$$
\begin{array}{|l}
\hline
\mathit{EditViewAddress} \\
\mathit{VisitorNameAddressViewOp} \\
\mathit{newAddressText}? : \mathit{TEXT} \\
\hline
\mathit{newAddressText}? \in \mathrm{dom}\ \mathit{textAddress} \\
\mathit{addressV}' = \mathit{newAddressText}? \\
\mathit{nameV}' = \mathit{nameV} \\
\hline
\end{array}
$$

Once again, the effect on the system can be derived in an obvious way. First consider the effect on the model, which has the SetAddress operation defined earlier.

This operation is invoked by the view:

$$
\begin{array}{|l}
\hline
EditAddress\_1 \\
\Delta VisitorModelView \\
v? : VISITORVIEW \\
a? : TEXT \\
\hline
idVisitorView' \in idVisitorView \oplus_{rel} \\
\quad \{EditViewAddress \mid \theta VisitorNameAddressView = idVisitorView(v?) \\
\quad \bullet\ selfV \mapsto \theta VisitorNameAddressView'\} \\
idVisitor' \in idVisitor \oplus_{rel} \\
\quad \{SetAddress \mid \theta Visitor = idVisitor(viewModel(v?)) \wedge \\
\qquad address? = textAddress(a?) \\
\quad \bullet\ self \mapsto \theta Visitor'\} \\
\hline
\end{array}
$$

Any other views connected to this model are updated to reflect the new state of the model. The simplest way of specifying this is to define that the views are reinitialized to reflect the new model state. (In simple cases, this might also be the way to implement the operation.)

$$
\begin{array}{|l}
\hline
EditAddress\_2 \\
\Delta VisitorModelView \\
v? : VISITORVIEW \\
\hline
idVisitorView' \in idVisitorView \oplus_{rel} \\
\quad \{InitVisitorNameAddressView \mid \\
\qquad viewModel(selfV) = viewModel(v?) \wedge \\
\qquad model? = idVisitor(viewModel(v?)) \wedge \\
\qquad selfV \in \mathrm{dom}\ idVisitorView \\
\qquad \bullet\ selfV \mapsto \theta VisitorNameAddressView\} \\
\hline
\end{array}
$$

(Note that this operation actually updates the original view as well as any others; this is usually the correct thing to do but it could be avoided if required.) The complete operation is just the composition of these two:

$EditAddress \mathrel{\widehat{=}} EditAddress\_1 \fatsemi EditAddress\_2$

5 CONCLUSIONS

5.1 Summary

We have made some progress towards being able to specify object-oriented systems in Z, and having a calculus of specifications which matches the object-oriented style. The method consists of:

1. conventions for modelling object states;
2. the use of object identities to refer to objects and express their individuality;
3. a convention for expressing the state of a system in terms of the objects it contains;
4. use of object identities to model relationships between objects;
5. a method of defining operations in terms of single objects and calculating their effect on the whole system, or on defined sets of objects;
6. a convention for modelling classes and their relationships;
7. a convention for representing metaclass information;
8. some limited guidance on the meaning of inheritance and the description of subclass states and operations.

In this paper we have covered the first five components of the method.

5.2 Outstanding Problems

We now have a reasonable method for defining object states and specifying systems in terms of these definitions. The notation is still a little clumsy. Furthermore, although we can define what we want in any particular case, we do not have *general* notations for describing what is wanted. This is for two reasons: some of the notions we want are new pieces of schema calculus, and unlike the mathematical language the schema calculus of Z is not extensible; also, some of the operations we want require quantification over types, which is not possible in a first order strongly typed language like Z.

However, we can propose some particular extensions and we would benefit from some extra notation:

1. Given any object class *Thing*, we can assume that the system contains a component *ThingSystem* with components *idThing* and *things*. We could imagine having a notation, say $\$Thing$, to denote this system.
2. Given any operation schema on Thing *Op* with input of type *INPUT*, we would like a notation, say $\mathbb{R}Op$, to denote the corresponding function of signature

   $$INPUT \longrightarrow Thing \leftrightarrow Thing$$

   In some cases, of course, there are several input parameters so the type *INPUT* is a product type.

   Such a notation would simplify the expressions used to calculate the effects of operations on the whole system or on arbitrary sets of objects.

5.3 Conclusion

The combination of a formal specification method with an object-oriented approach has proved very fruitful. Most ideas from Z have proved directly applicable. In some cases we have been able to extend the notation to express useful ideas in an object-oriented framework, but there are some areas where there is still a mismatch between the two approaches. We hope that the work reported here will be one step in bringing together two of the most powerful ideas in software development.

6 REFERENCES

References

[1] I. Hayes. *Specification Case Studies*. Prentice Hall, 1987. ISBN 0-13-826579-8.

[2] B. Meyer. Genericity versus Inheritance. *SIGPLAN Notices*, 21(11):391 –405, November 1986.

[3] J. A. Goguen and J. T. Tardo. An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications, 1979.

[4] S. A. Schuman, D. H. Pitt, and P. J. Byers. Object-Oriented Process Specification.

[5] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29 – 72, March 1988.

[6] R. L. London and K. R. Milsted. Specifying Reusable Components Using Z: Realistic Sets and Dictionaries. *ACM SIGSOFT Software Engineering Notes*, 14(3):120 –127, May 1989.

[7] H. B. M. Jonkers. An Introduction to COLD-K. July 1988.

[8] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4), December 1979.

[9] S. N. Khoshafian and G. P. Copeland. Object Identity. *SIGPLAN Notices*, 21(11):406 – 416, November 1986.

[10] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Second edition, 1992.

[11] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison Wesley, 1987.