Specifying and Interpreting Class Hierarchies in Z

Anthony Hall
Praxis
Bath UK

Abstract

In a previous article I described how Z could be used to specify certain aspects of object-oriented systems. This paper continues that approach by describing how Z can be used to describe relationships between classes. A Z model of a class hierarchy can be created. Two different but related semantics are given for this model. In the first, extensional, semantics a class is related to the set of object identifiers of objects belonging to the class and inheritance is interpreted as set inclusion. In the second, intensional, semantics a class is related to a definition of the intended behaviour of the class, and inheritance is interpreted as a particular kind of satisfaction relation between behaviour specifications. I give sound (but not complete) conditions for this relation to hold.

1 Introduction

The use of formal specification with the object-oriented approach to system construction is a subject of increasing interest. A recent book [6] outlines several approaches to using the formal notation Z [5] and related notations for object-oriented specification. One of the approaches cited in that work is one which we used successfully on a substantial project, and an earlier article [2] described some of that work. The article defined two aspects of the object-oriented approach:

- 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 some behaviour.

The earlier article only addressed the first of these aspects; in this paper I describe an approach to the second.

This paper is organised as follows. The next subsection summarises the approach which was introduced in the earlier article. Section 2 gives an informal introduction to the idea of class and to the relationship of inheritance which can exist between classes. It gives a notation for describing classes and their relationships in Z. Section 3 gives a semantics for this description by relating the idea of class to the set of objects which are instances of the class. Section 4 gives a semantics by relating the idea of class to the allowed behaviour of objects belonging to that class. Section 5 summarises the ideas of the paper and gives a brief comparison with other work.

1.1 Summary of Notation

This section summarises the approach and notation introduced in my previous paper, insofar as it is necessary to an understanding of what follows. (It contains some small extensions to that approach which we have found useful subsequently.)

An object is represented by a schema describing its state and a collection of schemas defining the operations on the object. The state schema of any object contains a variable *self* whose type is the set of identities of that kind of object, and whose value is the identity of that particular instance. For example we may wish to model a riding school, and to do so we would introduce objects to represent riders. If we were interested in the name, weight and skill level of a rider, then the corresponding object schema might be

$$
\begin{array}{|l}
\hline
Rider \\
\hline
self : RIDER \\
name : NAME \\
weight : WEIGHT \\
skill : SKILL \\
\hline
\end{array}
$$

I use the notation $\mathbb{S}Rider$ to mean the following schema representing a collection of riders, uniquely labelled with their identities:

$$
\begin{array}{|l}
\hline
\mathbb{S}Rider \\
\hline
riders : \mathbb{P}\ Rider \\
idRider : RIDER \pfun Rider \\
riderIds : \mathbb{P}\ RIDER \\
\hline
idRider = \{r : riders \bullet r.self \mapsto r\} \\
riderIds = \mathrm{dom}\ idRider \\
\hline
\end{array}
$$

Here *riders* is the set of all known objects of class *Rider*—that is, all riders known to the system. The other two variables are both derived from *Riders* and do not themselves add any extra state. The first, *idRider*, is a function which, given the identity of a rider, yields the object with that identity. The fact that the relation *idRider* is a function (it is in fact a partial injection) does add information: it guarantees that no two riders have the same identity. The variable *riderIds* is simply the set of identities of all riders known to the system and is introduced only for convenience in writing later specifications.

Operations on objects are represented by a normal Z state-change schema, with the extra constraint that object identities are invariant. Thus any operation on a rider includes the following schema:

$$
\begin{array}{|l}
\hline
RiderOp \\
\hline
\Delta Rider \\
\hline
self' = self \\
\hline
\end{array}
$$

I use the notation $\mathbb{R}Op$ to mean an operation schema converted into a function from input parameters to a relation between start and end state. For example, if *ChangeSkillRider* is an operation defined as follows

$$
\begin{array}{|l}
\hline
ChangeSkillRider \\
\hline
RiderOp \\
skill? : SKILL \\
\hline
\ldots \\
\hline
\end{array}
$$

then I assume the following definition

$$
\begin{array}{|l}
\hline
\mathbb{R}\, ChangeSkillRider : SKILL \longrightarrow Rider \longleftrightarrow Rider \\
\hline
\mathbb{R}\, ChangeSkillRider = \\
\quad \{ s : SKILL \bullet s \mapsto \\
\qquad \{ ChangeSkillRider \mid skill? = s \\
\qquad\quad \bullet \theta Rider \mapsto \theta Rider' \\
\qquad \} \\
\quad \} \\
\hline
\end{array}
$$

2 Classes and Class Relationships

2.1 What is a class?

The notion of *class* has two related but distinct meanings: we can think of it as defining some set of similar objects, or we can think of it as defining the properties that those objects have in common. The first meaning is called the *extension* of the class; the second is called the *intension*. For example the class *Rider* can mean:

- The set of all riders
  - the extension
- The definition of the properties of riders
  - the intension

A feature of object-oriented systems is that classes are not independent of each other: they can be arranged in a hierarchy. One class can be said to be a subclass of another. For example, we might introduce the idea of *teachers* into our riding school model: a teacher would be a particular kind of rider, and the class *Teacher* would be a subclass of the class *Rider*.

The idea is that objects belonging to a subclass *class*2 are 'like' objects of its superclass *class*1, and that only the differences need to be mentioned when defining or coding the subclass *class*2. This idea is called *inheritance*. Although inheritance is an immensely useful idea, it does not have a single clear interpretation. There is no agreement on what 'like' means. Some possible meanings are:

1. An instance of *class*2 can at all times be treated as if it were an instance of *class*1.

   This idea is called substitutability. If an object of *class*2 is to be substitutable for another of *class*1, then *class*2 may have more operations,

but it must have at least the operations of *class*1 and their effect must be 'the same'. It is now common to call this relationship between classes *subtyping*.

2. *Class*2 has at least a set of operations with the same names and parameters as *class*1.

   Here there is no guarantee that the operations of *class*2 have anything in common except their names and parameters. This is what inheritance means in C++ and in Smalltalk.

3. *Class*2 normally has the same operations as *class*1, but not necessarily.

   This is the model in Eiffel.

4. The code of *class*1 may be useful when implementing *class*2.

In this paper I am interested in subtyping. I would like to develop a model satisfying the principle of substitutability. There are three reasons for this choice:

1. When writing a specification we are interested in ensuring certain behaviour of the specificands. A model of inheritance which did not ensure some sort of behavioural compatibility would not be very interesting from a specifier's point of view.

2. Most models of inheritance have *some* notion of "similarity" between the methods. Subtyping is the strictest such notion; if we can model subtyping then we may have made progress towards formalising the less strict forms of inheritance.

3. The fourth, very weak, model of inheritance, whereby the specification of a superclass would be re-used in the specification of its subclass, is already available in Z and poses no additional problems.

We can interpret the substitutability requirement in extensional or intensional terms. Extensionally, it means that the set of all instances of *class*2 is a subset of the instances of *class*1. Intensionally, it means that an instance of *class*2 must behave as if it were an instance of *class*1.

2.2 A Model of Classes

In Z it is, of course, perfectly straightforward to set up a model to represent classes. We can simply introduce the notion of all classes, say

$$[CLASS]$$

and then define particular members of this set, such as

$$RiderClass : CLASS$$

We can represent a class hierarchy by a relation between classes.

$$subSuper : CLASS \leftrightarrow CLASS$$

$$subSuper^+ \cap \mathrm{id}\ Class = \varnothing$$

Here the *subSuper* relation gives the immediate superclasses of a class. In general the relation is acyclic. This is expressed by saying that the transitive closure does not contain any elements from the identity mapping—in other words, we cannot reach a class by following its superclass, then the superclass of its superclass, and so on.

(Note that "id *Class*" is the identity relation on classes, not to be confused with the variables we have been defining like "*idRider*".)

For example, we can introduce the class of teachers and say that it is a subclass of the class of riders:

$$TeacherClass : CLASS$$
$$\hline$$
$$(TeacherClass, RiderClass) \in subSuper$$

This model already allows us to characterise the class structure that is allowed. For example, in many systems, only single inheritance is allowed: a class may only have one immediate superclass. In that case *subSuper* is a function.

$$subSuper \in CLASS \pfun CLASS$$

This model is adequate to represent the classes in a system where the class relationships are fixed as part of the specification. On the other hand, it may be desirable to define a system in which new classes can be defined and new class relationships established. In that case, the set of known classes and the relationships between them would be part of the state, and there would be a collection of operations on the class system. A suitable state definition would be

ClassSystem
$$classes : \mathbb{P}\, CLASS$$
$$subSuper : CLASS \rel CLASS$$
$$\hline$$
$$subSuper^+ \cap \mathrm{id}\, Class = \emptyset$$
$$\mathrm{dom}\, subSuper \subseteq classes$$
$$\mathrm{ran}\, subSuper \subseteq classes$$

There is an obvious collection of operations to construct and maintain the class system.

In the rest of this paper I will assume a model in which the class system is fixed for a given specification. The extension to a dynamic class system is straightforward, although the following points should be borne in mind:

- The *ClassSystem* schema must be incorporated into any schemas which refer to the class structure.
- Care must be taken with operations which delete or modify existing class relationships if there are instances of the classes in existence.

3 Extensional Semantics

In this section I give a simple interpretation of a class in terms of the identities of the objects which are instances of the class. In this interpretation, the subclass relation on classes corresponds to a subset relation on instances.

In this model, I assume that all object identities are of the same type, say

$$[OBJECT]$$

For each class, such as riders and teachers, there is a set of identities of instances of that class, for example

$$RIDER, TEACHER : \mathbb{P}\ OBJECT$$

There is a relation between each class and the set of identities of members of that class

$$extension : CLASS \leftrightarrow OBJECT$$

For each class we can now define the members of that class:

$$extension(\!|\{RiderClass\}|\!) = RIDER$$
$$extension(\!|\{TeacherClass\}|\!) = TEACHER$$

All riders are members of *RiderClass*; they are the only members.

3.1 Extensional meaning of Subclass

To give a meaning to subclassing we have to relate the subclass relation to the extension relation.

$$\forall\, c_{SUB}, c_{SUPER} : CLASS \bullet$$
$$\quad c_{SUB} \mapsto c_{SUPER} \in subSuper \Rightarrow$$
$$\quad extension(\!|\{c_{SUB}\}|\!) \subseteq extension(\!|\{c_{SUPER}\}|\!)$$

Every member of a subclass is a member of its superclass.

For example, from these definitions we can prove that

$$\vdash TEACHER \subseteq RIDER$$

Note that the definition means that an object may belong to more than one class—the inverse of *extension* is not necessarily a function. Indeed, if an object belongs to a class, it must also belong to all the superclasses of that class.

We may choose to restrict the model so that each object is a *direct* instance of exactly one class. A direct instance of a class is an object which is an instance of that class but not an instance of any subclasses of that class.

The idea of direct instances can be introduced into the model: the direct instances are the extension of a class, with the extensions of its subclasses subtracted.

$$direct : CLASS \leftrightarrow OBJECT$$
$$direct = extension \setminus (subSuper^{\sim} \fatsemi extension)$$

We can then use *direct* to describe restrictions on the class structure such as a requirement that each object is a direct instance of exactly one class.

$$direct^{\sim} \in OBJECT \nrightarrow CLASS$$

Unless we add such a restriction, subclasses need not be disjoint. For example, we might divide riders into men and women, quite independently of whether they were teachers or not.

$$ManClass, WomanClass : CLASS$$
$$\{(ManClass, RiderClass), (WomanClass, RiderClass)\} \subseteq subSuper$$

$$MAN, WOMAN : \mathbb{P}\ RIDER$$
$$extension(\!|\{ManClass\}|\!) = MAN$$
$$extension(\!|\{WomanClass\}|\!) = WOMAN$$

MAN and WOMAN are themselves disjoint, and that can be added to the model if required:

$$\mathsf{disjoint}\ \langle MAN, WOMAN \rangle$$

Now objects can be direct members of both *ManClass* and *TeacherClass*.

Multiple inheritance is natural in this model; for example, we could easily define a new class for male teachers:

$$MaleTeacherClass : Class$$
$$\{(MaleTeacherClass, TeacherClass), (MaleTeacherClass, ManClass)\} \subseteq subSuper$$

From this we can deduce that

$$\vdash extension(\!|\{MaleTeacherClass\}|\!) \subseteq (MAN \cap TEACHER)$$

If we had the rule that objects could only be direct members of one class, we would require that the class of male teachers included *everyone* who was both male and a teacher. We would then have the stronger condition

$$extension(\!|\{MaleTeacherClass\}|\!) = (MAN \cap TEACHER)$$

Of course, if we try to inherit from disjoint superclasses, then the class will have an empty extension.

4 Intensional Semantics

The intension of a class is the definition of its behaviour, and in Z this is typically represented by its state schema and the collection of operations. In standard Z there is no way of formally relating this intension to the class, for two reasons:

1. There is no single Z object which represents the whole, encapsulated, behaviour.
2. Even if we chose to represent the behaviour by a surrogate such as the state schema, the state schemas for objects of different classes would, in general, be of different types. There would therefore be no reasonable way of representing anything like a meaning function which was a function from a class to the corresponding state.

We could, of course, overcome these problems by adopting a meta-modelling approach whereby the state schemas and operations themselves were represented by objects of some Z type: for example the state could be a mapping from names to some universe of values, and operations could be relations between such objects. This approach would be suitable for writing a Z specification of an object-oriented programming or design language, but not very convenient for specifying a particular object-oriented system.

In my method, therefore, I establish a connection between the intension and the extension by convention:

1. The name of the state schema is the name of the class.
2. The value of *self* in the state schema is constrained to be in the extension of the class.

The definition of the *Rider* schema, above, illustrates this convention.

4.1 Intensional Meaning of Subclass

If we are to be able to use an instance of a subclass wherever an instance of a superclass is expected, then the subclass instance must be a valid implementation of the superclass: this is similar to the refinement relation which exists between specifications and correct implementations. The superclass can be regarded as a specification which all its instances, including instances of its subclass, must meet.

There are some differences between the subclass relation and normal refinement, however. The rules for refinement in Z are given in the Z Reference Manual [5]. A simplified version, omitting input and output parameters, is as follows. There must exist an abstraction schema Abs which represents a relation between the abstract state $Astate$ and the concrete state $Cstate$, such that for every abstract operation Aop and concrete operation Cop the following are true:

R1 If the abstract operation is possible, then the corresponding concrete operation must be possible.

$$\forall Astate; Cstate \bullet \text{pre } Aop \wedge Abs \Rightarrow \text{pre } Cop$$

R2 If the abstract operation is possible then the concrete operation must give a state which could have been reached by the abstract operation.

$$\forall Astate; Cstate; Cstate' \bullet \\ \quad \text{pre } Aop \wedge Abs \wedge Cop \Rightarrow (\exists Astate' \bullet Abs' \wedge Aop)$$

R3 All possible initial concrete states must be related to possible initial abstract states.

$$\forall Cstate \bullet Cinit \Rightarrow (\exists Astate \bullet Ainit \wedge Abs)$$

Note that these rules do not require that the abstraction relation is total (that all concrete states are related to abstract states) nor that it is a function (that each concrete state is related to just one abstract state).

A number of rules have been proposed for the corresponding relation between a class and its subclass, particularly in [1], [3] and [4].

The rules proposed here are close to all those. They are similar to, but not the same as, the Z rules for refinement, with *superclass* substituted for *abstract* and *subclass* for *concrete*.

S1 If the superclass operation is possible on a valid state of the subclass, then the subclass operation must be possible.

$$\forall Superstate; Substate \bullet \text{pre } Superop \wedge Abs \Rightarrow \text{pre } Subop$$

This is the same as the corresponding refinement rule.

S2 If the superclass operation is possible then the subclass operation must give a state which could have been reached by the superclass operation.

$$\forall Superstate; Substate; Substate' \bullet \\ \quad \text{pre } Superop \wedge Abs \wedge Subop \Rightarrow (\exists Superstate' \bullet Abs' \wedge Superop)$$

This, too, is the same as the corresponding rule for refinement.

S3 All states of the subclass object must correspond to at least one state of the superclass object.

$$\forall Substate \bullet (\exists Superstate \bullet Abs)$$

My third rule is different from normal refinement. I do *not* require that all initial states of the subclass represent valid initial states of the superclass. Although it is reasonable to expect any subclass object to behave as a superclass object once created, we assume that objects are created with an explicit class in mind and therefore there is no need for creation of a subclass object to behave like creation of a superclass object.

On the other hand, unlike the refinement case, I do insist that *all* subclass states are related to valid superclass states. This is because the subclass may have more operations than the superclass, so state changes

in a subclass object may occur via operations not related to superclass operations.

I do not require the converse—it is not at all necessary, unlike in the refinement case, that all reachable states of the superclass have corresponding subclass states. To see this, consider an example suggested in [4] : a set which can only grow. Suppose the initial of such a set is empty. My rules would admit as a valid subclass a set whose initial state has ten members. Such a subclass can never reach states with fewer than ten members; however, a client, on being given such a subclass set, cannot tell that it did not start life with no members and grow to its current size.

The corresponding rules in the references cited are as follows:

Cusack In [1] Cusack defines two kinds of inheritance, including subtyping inheritance, in the framework of an object-oriented extension to Z. Her rules for subtyping are the very close to those given here. She chooses as the relation *Abs* a particular function from the subclass state to the superclass state. In the following sections I follow the same route though the function proposed here is less general that Cusack's. Furthermore her precondition and postcondition rules are more general than those here because they consider changes in the types of the output variables of the operation as well as the state variables.

Lano and Haughton Lano and Haughton in [3] also discuss subclassing in the context of an object-oriented language which is richer than Z. They give five rules for refinement, and then show that one class is a subtype of another if there is an abstraction *function* between them satisfying those rules. Their rules are equivalent to the three rules given here for subclassing, plus two others. One extra rule requires that the invariant of the subclass implies the invariant of the superclass. This rule is subsumed by my third rule when the abstraction relation is a function. The fifth rule is of a completely different kind: they include within their specification language a *history predicate* for each class, and require that the history predicate of the concrete specification is stronger than that for the abstract. Since the normal state plus operations style of Z does not include explicit history predicates, I do not discuss them here.

Liskov and Wing A third set of rules is given in [4] independently of any particular specification language. They assume that the abstraction relation is a function and give the slightly simpler rules to which mine reduce in that case. They also consider explicit exceptions and give rules for those. They do consider history, but in a different way from [3].

Rather than having an explicit history predicate, they require that only histories allowed for the supertype are available for the subtype (discounting parts of the subtype state which are meaningless for the supertype). This is ensured by requiring that the effect of every subtype operation on the supertype state is expressible as a program using only the supertype operations. For example, they would not allow a set which can only grow to have as a subtype a set which also had a remove operation.

I do not make such a restriction on the extra subtype operations, and hence would allow this particular subtype. I can justify this if I take the view that, while a client of a subtype object requires its own operations on the object to do what is expected, it cannot have any expectations about other programs accessing the object except that they maintain its invariant. It is also consistent with my not requiring the initial state of a subtype object to correspond with an initial state of the supertype.

The rest of this section describes a particular approach to writing Z specifications of subclasses. Subclasses defined according to this approach will be subtypes according to my rules; in this sense the approach is *sound*. However, the approach is very simple-minded—for example I use a particularly simple function as an abstraction relation—so it only generates a few of the possible subtypes, and in that sense it is far from *complete*.

4.2 Subclassing the Object State

4.2.1 Schema Inclusion

Although formally any relation satisfying the above conditions is allowable, the most obvious way to define a subclass state is by *inclusion* of the superclass schema. The subclass can add to the declaration or the predicate or both.

The simplest case is where nothing is added to the declaration in the subclass.

For example, supposing we had a class *Lesson*, defined as follows:

Lesson

$$self : LESSON$$
$$time : TIME$$
$$riders : \mathbb{F}\ RIDER$$
$$horses : \mathbb{F}\ HORSE$$
$$maxRiders : \mathbb{N}$$

$$\#riders \leq maxRiders$$

$$indoorLimit : \mathbb{N}$$

Indoor

$$Lesson$$

$$self \in INDOOR$$
$$maxRiders \leq indoorLimit$$

The Z types of *Lesson* and *Indoor* are the same; *Indoor* is simply a restricted kind of *Lesson*. The abstraction relation is the identity relation. (Note, however, that even in this simple case we cannot tell whether *Indoor* is a subclass of *Lesson* without checking the operation definitions. This is discussed in the section on method inheritance.)

Often, however, a subclass needs more variables in its state than a superclass. It includes its superclass schema but extends it. We might define *Teacher* as follows, for example:

Teacher

$$Rider$$
$$qualification : SKILL$$

$$self \in TEACHER$$
$$qualification \leq skill$$

A teacher is qualified to teach up to some skill level, which must not be greater than their own skill. (We assume that there is an order relation $\leq$ on skill levels.)

The Z type of *Teacher* is not the same as that of *Rider*. However, all teachers are riders in the following sense:

$$\{Teacher \bullet \theta Rider\} \subseteq \{Rider\}$$

Here the abstraction relation is the projection function

$$(\lambda\, Teacher \bullet \theta Rider)$$

In general, the abstraction function is the projection function from Subclass state to Superclass state:

$$(\lambda\, Substate \bullet \theta Superstate)$$

which is the identity function when nothing is added to the state in the subclass.

Subclassing by schema inclusion allows multiple inheritance quite naturally. For example, the state of a male teacher includes the state of *Man* and the state of *Teacher*:

MaleTeacher

$$Man$$
$$Teacher$$

$$self \in MALETEACHER$$

4.2.2 Subclasses in the System State

If classes are related in a hierarchy and the states are related by projection, then the state of a collection of objects has the following properties:

- The object identities in a subclass are a subset of the object identities in the superclass
- The states associated with the superclass are projections of the states of any subclass objects.

By saying that Teacher is a subclass of Rider, I intend that the state is as follows:

$$
\begin{array}{|l}
\hline
RiderHierarchy \\
\S\, Rider \\
\S\, Teacher \\
\hline
teacherIds = riderIds \cap TEACHER \\
(\forall\, t : teacherIds \bullet \\
\quad (\lambda\, Teacher \bullet \theta Rider)(idTeacher(t)) = idRider(t)) \\
\hline
\end{array}
$$

Here the ids of known teachers are a subset of the ids of known riders. Furthermore, the states of teachers, when viewed as riders, are just the *Rider* states corresponding to those ids.

4.3 Method Inheritance

It is more difficult to give a simple account of operation inheritance in Z than it is to describe state inheritance. There are two reasons. One is that the syntax of Z does not directly support some things I would like to do, like dynamic binding of messages to methods. The second is more serious, and is not related just to notation but to the whole meaning of inheritance: many classes which seem intuitively to be specialisations of more general classes turn out to have operations which are *not* refinements of the operations of the more general classes.

- Syntactic Issues
  - Encapsulation
    Since there is no way in Z of grouping together operations, there is no way of stating that a subclass must have all the operations of its superclass. However, this can simply be required as an informal property of the specification.
  - Method Names
    If the methods of different subclasses are in fact different in any way at all, it is not possible to give them the same name in Z.
  - Method Types
    Even if we could give different methods the same name, this would not be enough to extend the operation calculus described in [2] to use different methods on different object types. We also need to recognise that if the types of the state schemas are different, then the types of the methods are different too.
- Semantic Issues
  - Preconditions
    There is a contradiction between specialisation of the state, which increases the constraints and thus tends to strengthen the preconditions of operations, and the satisfaction relation which requires that preconditions be weakened.

- Operation signatures

  If the state is extended in a subclass, operations need to define the effect on the extra state. This may require parameters not present in the superclass operation.

4.3.1 Messages and Methods

Object-oriented systems usually allow dynamic binding, whereby the client of an object does not need to know what method is going to be invoked by an object. Instead it issues a *message* to the object and the object translates this to a method depending on the server object's class.

Since Z is a formal language I need to express this fact formally, although it is often glossed over in descriptions of Object-Oriented systems.

- The name of an operation schema is a *method* name
- Different classes have *different* method names
  - even if one is a subclass of another
- Conventionally, I construct *Method* name from *Message* name plus *Class* name.

$$
\begin{array}{|l}
\hline
BookIndoor \\
\Delta Indoor \\
rider? : Rider \\
\hline
\ldots \\
\hline
\end{array}
$$

  Here 'Book' is the message name and 'Indoor' is the name of the class which has the method.

More generally, this raises the question of what we *want* the specification of an object to define. Let us suppose that we have a client object, such as a lesson, and this object refers to other objects (which in this context we can think of as servers) such as riders. Now there may be an operation on a lesson which is defined in terms of its effect on the riders in the lesson. For example, completion of a particular kind of qualifying lesson may involve setting the skill of all the riders to the level to which completion of the lesson entitles them. We can take two views of the operation *CompleteLesson*:

1. Object-oriented approaches typically take the view that the definition of *CompleteLesson* includes the fact that it sends the *ChangeSkill* message to each of the riders. The *effect* of *ChangeSkill* on a rider is a matter for the definition of the *Rider* class and its subclasses. The advantage of this approach is that it separates the concern of what the lesson does from the concern of what the rider does. A possible disadvantage, however, is that in order to know whether sending the *ChangeSkill* message is the *right* thing for the *CompleteLesson* operation to do, we may need to know what *ChangeSkill* actually does, at least to some extent.

2. A more traditional view would be that the definition of *CompleteLesson* should define its effect regardless of how this effect is achieved. In this approach, a specification of *CompleteLesson* would include the effect on the state of all associated Riders. Whether or not this effect was defined directly, by mentioning the state variables, or indirectly by using the definition of an operation on Rider (for example in the style of [2]) would be a secondary issue.

In this paper I take a somewhat intermediate approach, in keeping with the style of the previous work. I assume that we wish to define the actual effect on riders; on the other hand, we would like to do this by using operations defined on riders, rather than by repeating the details of what the operation does in the definition of *CompleteLesson*. The previous paper showed various methods for doing this but ignored the possibility that riders might be of different classes; the question I address here is: "Under what circumstances can these methods be used when Riders may in fact be more specialised objects, such as Teachers?"

The answer I give is that the effect can be defined in terms of the behaviour of a *Rider*, provided that every subclass of *Rider* satisfies the definition of that behaviour.

We can then define *CompleteLesson* in terms of *ChangeSkillRider*. If some of the riders are in fact teachers, then *ChangeSkillTeacher* may actually do other things as well, which using this approach I do not specify. However, I do require that

$$\text{pre } ChangeSkillRider \land Teacher \Rightarrow \text{pre } ChangeSkillTeacher$$

Furthermore, I require that

$$ChangeSkillTeacher \vdash \theta ChangeSkillRider \in ChangeSkillRider$$

These conditions follow from Rules S1 and S2, where *Abs* is simply the schema *Teacher* itself.

4.3.2 Inheritance as Specialisation

We can always use schema inclusion to guarantee that the effect of a subclass operation is one allowed by the superclass operation. However, schema inclusion does not guarantee the weakening of the precondition—on the contrary, it makes the precondition at least as strong, and it may be stronger. For example, supposing outdoor lessons were unsuitable for novices. Then we might be tempted to write a general operation *BookLesson*, and specialise it as follows:

BookLesson

$$LessonOp$$
$$rider? : Rider$$

$$\#riders < maxRiders$$
$$riders' = riders \cup \{rider?.self\}$$
$$time' = time$$
$$horses' = horses$$
$$maxRiders' = maxRiders$$

BookOutdoor

$$OutdoorOp$$
$$BookLesson$$

$$rider?.skill > NOVICE$$
$$\ldots$$

However, *BookOutdoor* is NOT a refinement of *BookLesson*. It is NOT the case that

$$\text{pre } BookLesson \Rightarrow \text{pre } BookOutdoor$$

So with these definitions we cannot use an *Outdoor* whenever we need a *Lesson*.

If strict subclassing of this sort is required, more care must be taken about preconditions. The operations on the superclasses must allow failures wherever any subclass might fail.

For example, we could define *BookLesson* as follows, allowing for the possibility of failure.

BookLesson

$$LessonOp$$
$$rider? : Rider$$

$$\#riders < maxRiders$$
$$(riders' = riders \cup \{rider?.self\} \lor$$
$$\quad riders' = riders)$$
$$time' = time$$
$$horses' = horses$$
$$maxRiders' = maxRiders$$

Now we could define *BookOutdoor* so as to guarantee success so long as the rider was sufficiently skilled.

BookOutdoor

$$OutdoorOp$$
$$BookLesson$$

$$riders' = riders \cup \{rider?.self\} \Leftrightarrow rider?.skill > NOVICE$$

Alternatively, if OutdoorLesson might have still more restrictive subclasses, it is possible that the strongest thing we would be able to say is that the rider will *not* be added *unless* they are sufficiently skilled:

BookOutdoor

$$OutdoorOp$$
$$BookLesson$$

$$riders' = riders \cup \{rider?.self\} \Rightarrow rider?.skill > NOVICE$$

If the subclass state is an extension of the superclass state, then just as we required the state of a superclass to be a projection of the state of any of its

subclasses, so we can require an operation on a superclass to be a projection of the operation on its subclasses.

For example, we might represent a teacher as a rider who is qualified to teach up to a certain skill level less than or equal to their own skill.

Teacher

$$Rider$$
$$qualification : SKILL$$

$$self \in TEACHER$$
$$qualification \leq skill$$

Now the operation to set the skill of a teacher may also need to change their qualification if their skill has been downgraded:

ChangeSkillTeacher

$$TeacherOp$$
$$ChangeSkillRider$$

$$skill' \geq qualification \land$$
$$\quad qualification' = qualification \lor$$
$$skill' < qualification \land$$
$$\quad qualification' = skill'$$

A strict specialisation of an operation like this has the property that, viewed as an operation on the superclass, it behaves just like the corresponding superclass operation.

1. We require that

$$\text{pre } ChangeSkillRider \land Teacher \Rightarrow \text{pre } ChangeSkillTeacher$$

   Which is true in this case.

2. The schema inclusion guarantees that

$$ChangeSkillTeacher \vdash \theta ChangeSkillRider \in ChangeSkillRider$$

   This is a special case of rule S2.

If 2. is true, then we can promote operations on objects to operations over the whole state, even if the objects may be of different classes. For example (if the operations are deterministic):

ChangeSkillRiderSystem

$$\Delta \mathbb{S} Rider$$
$$rider? : RIDER$$
$$skill? : SKILL$$

$$idRider' = idRider \oplus$$
$$\quad ((\{ rider? \} \lhd idRider) \fatsemi (\mathbb{R}\, ChangeSkillRider\ skill?))$$

is true even if some of the riders are teachers, provided the precondition of *ChangeSkill* is satisfied.

Perhaps more surprisingly, the following is also true, even if the rider is not a teacher:

ChangeSkillTeacherSystem

$$\Delta\mathbb{S}\,Teacher$$
$$rider? : RIDER$$
$$skill? : SKILL$$

$$idTeacher' = idTeacher \oplus (((\{rider?\} \lhd idTeacher) \fatsemi (\mathbb{R}\,ChangeSkillTeacher\,skill?))$$

If the rider is a teacher then the appropriate change is made; if not, then it is not in the domain of *idTeacher* so nothing changes.

Therefore, this scheme has the nice property that a complete specification can be put together by simply conjoining the effect of the operation in all the subclasses on the appropriate subclass state, without worrying about whether the object in question is a member of the subclass or not. We can simply write schemas like:

ChangeSkillSystem

$$ChangeSkillRiderSystem$$
$$ChangeSkillTeacherSystem$$

5 Summary and Relation to Other Work

5.1 Summary

Classes and class relationships can be modelled explicitly in Z.

An extensional semantics can be given for this model, and inheritance can be interpreted as set inclusion.

The model can be related to the behavioural definitions of the classes by adopting conventions for specifying classes and naming objects and identities.

Z can be used to specify operations in terms of their overall effect on the state, rather than in terms of the messages that are sent in implementing the operation. This type of definition is in the spirit of other formal specifications, but has some disadvantages in that it does not directly model dynamic binding of messages to methods. However, it can be used to give truthful, if incomplete, specifications of operations which send messages to subclass objects, provided that the subclass *satisfies* the definition of its superclasses.

The following rules allow subclasses to be modelled in Z. The rules are sound, although not complete, in that a subclass defined in this way is a subtype of its superclass.

1. The state schema of a subclass includes the state schema of its superclass.

2. Each operation schema of the subclass includes the corresponding operation schema of the superclass.

3. The precondition of the subclass operation is no stronger than the precondition of the superclass operation.

Under these conditions, the effect of operations on the whole system state can be calculated using the methods for promoting operation definitions given in my previous paper, and the result will be correct even if the objects concerned belong to subclasses of the classes for which the operations and state were defined.

5.2 Other Work

There is now a considerable literature on object-oriented formal specification. The work reported in this paper differs from almost all other approaches in that it is based on pure Z, and does not extend the language with any new constructs. It has, therefore, the advantage of a well understood semantics, but the disadvantage that some object-oriented constructs are awkward to express in standard Z.

The most closely related work on subtyping and inheritance, [1], [3] and [4] has already been discussed. The present paper does not attempt to define new rules for subtyping, but rather to show how to construct Z specifications which conform with well-defined rules.

6 Acknowledgements

I am grateful to all the people who have read early drafts of this paper and suggested improvements in both presentation and content.

References

[1] Cusack E. Inheritance in object oriented Z. In America P (ed), ECOOP '91: European Conference on Object-oriented Programming, Lecture Notes in Computer Science No. 512, pp 167 – 179. Springer Verlag, 1991.

[2] Hall A. Using Z as a Specification Calculus for Object-Oriented Systems. In Bjorner D, Langmaack H (eds), Proceedings of VDM 90, Lecture Notes in Computer Science No. 428, pp 290 – 318. Springer Verlag, 1990.

[3] Lano K, Haughton H. Reuse and Adaptation of Z Specifications. In Bowen JP, Nicholls JE (eds), Z User Workshop London 1992, Workshops in Computing, pp 62 – 90. Springer Verlag, 1992.

[4] Liskov B, Wing JM. A New Definition of the Subtype Relation. In Nierstrasz OM (ed), Proceedings of ECOOP '93 - Object-Oriented Programming, Lecture Notes in Computer Science No. 707, pp 118 – 141. Springer Verlag, 1993.

[5] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall, Second edition, 1992.

[6] Stepney S, Barden R, Cooper D. Object Orientation in Z. Workshops in Computing. Springer Verlag, 1992.