Working Paper Series
ISSN 1170-487X

Object-Orientation in Standard Z (New title: Object Orientation without Extending Z)

Mark Utting and Shaochun Wang

Working Paper: 12/02
December 2002

© 2002 Mark Utting and Shaochun Wang
Department of Computer Science
The University of Waikato
Private Bag 3105
Hamilton, New Zealand

Object Orientation without Extending Z

Mark Utting and Shaochun Wang

The University of Waikato, Hamilton, NZ.
Email: {marku,sw19}@cs.waikato.ac.nz
WWW: http://www.cs.waikato.ac.nz/~marku

Abstract. The good news of this paper is that without extending Z, we can elegantly specify object-oriented systems, including encapsulation, inheritance and subtype polymorphism (dynamic dispatch). The bad news is that this specification style is rather different to normal Z specifications, more abstract and axiomatic, which means that it is not so well supported by current Z tools such as animators. It also enforces behavioural subtyping, unlike most object-oriented programming languages. This paper explains the proposed style, with examples, and discusses its advantages and disadvantages.

1 Introduction

Object orientation offers a technology for structuring large, complex software systems [Mey97], so many Z researchers have proposed different approaches for extending Z with an object-oriented structuring mechanism [SBC92]. These include attempts to use standard Z in a more object oriented style, and proposed extensions to Z to allow fully object oriented specifications. Some of them are being widely accepted, some are not.

One of the most popular extensions is Object-Z [DKRS91]. From our experience of Object-Z, we found that its state semantics in modelling objects is too complex. It is a better match for software implementation, rather than for software specification and design. In other words, its explicit state modelling is a structuring mechanism simulating object oriented programming (OOP), not emphasizing the abstract nature of object oriented analysis and design (OOAD). We believe that some of the mechanisms of object-oriented programming, such as non-monotonic inheritance and the use of reference semantics as the default paradigm, need to be specified abstractly in an object-oriented specification language and that OOAD can be supported with a simple but powerful semantics.

Our intention is to explore the semantics of object oriented concepts, and to specify object oriented systems in Z. The key insight of this paper is that by using an abstract model of objects, subtypes can be modelled as subsets; moreover, we can use subsets to model inheritance and dynamic dispatch. We also introduce an elegant encoding of objects into standard Z, which is described in Section 6.1.

The following sections illustrate the approach with a series of examples, gradually introducing more features and discussing their ramifications. Section 9

describes our conclusions and areas for future work. This paper uses a value semantics for objects, rather than reference semantics, but the conclusion briefly discusses how our approach can also support references and object identity.

2 Encoding Object-Orientation into Z

This section describes how we represent objects and methods in Z. The four key ideas, explained in the following subsections, are that:

1. **Objects are black boxes.**
2. **Subtypes are subsets.**
3. **Methods are functions/relations.**
4. **Observations allow model-oriented specification.**

2.1 Objects are black boxes

Unlike most object-oriented extensions of Z, we do not specify a concrete model of objects. Instead we view each object as a black box whose internal details are hidden. In Z, we do this by defining a given type for each hierarchy of classes. An object is simply a member of this given type.

To model a single-rooted inheritance hierarchy where all classes inherit from the *Object* class (as in Java and Smalltalk), we define a single given type

$$[Object]$$

To model a multi-rooted inheritance hierarchy (as in C++), we define one given type for each root class (e.g., $[Document, Window]$). With this multi-rooted approach, errors such as applying a method to an object of the wrong class can often be detected statically by the Z type system, whereas in the single-rooted approach, those errors would be caught by the domain checks of Z/EVES instead. The multi-rooted approach has the disadvantage (or advantage) that it is impossible to later define an object that inherits from two different hierarchies. For example, $x \in Document \cap Window$ is ill-typed in Z. In other words, the multi-rooted approach ensures that two class hierarchies with separate roots are *disjoint*. For this reason, when specifying a new system whose class hierarchies are likely to evolve, we usually commence with the single-rooted approach because it is more flexible.

2.2 Subsets model subtypes

This approach to inheritance and behavioural subtyping is refreshingly simple. To define a new type of objects, *Document*, which inherits from an existing type (say, *Object*), we simply define *Document* to be a subset of *Object*. To write $Document \subseteq Object$ as a declaration in Z, we must write this in the slightly less obvious form:

$$\mid \; Document : \mathbb{P}\, Object$$

This extends elegantly to multiple inheritance. For example, we might want to specify that an *Pane* is a *Document* that is displayed in a *Window*. That is, $Pane \subseteq Document$ and $Pane \subseteq Window$. We write this in Z as:

$$\mid \; Window : \mathbb{P}\, Object$$

$$\mid \; Pane : \mathbb{P}(Document \cap Window)$$

2.3 Methods are functions/relations

In a programming language, a method call is written as:

```
outputs := object.method(inputs)
```

This method call typically changes the internal state of the object, and may have side-effects on other parts of the system such as the outputs.

Given $object, object' \in Class$, $inputs \in Inputs$ and $outputs \in Outputs$, we model the above method call by the Z predicate:

$$(object', outputs) \in method(object, inputs)$$

where *method* is a loosely defined axiomatic relation:

$$\begin{array}{|l} method : (Class \times Inputs) \leftrightarrow (Class \times Outputs) \\ \hline PreAx \\ PostAx \end{array}$$

When a method must modify other objects, these must be passed as inputs and returned as outputs of the method. For example, an execute method of Command in [GHJV94] on page 233-242, can be specified as:

$$\begin{array}{|l} execute : (Command \times Document) \rightarrow Document \\ \hline \end{array}$$

The preconditions and postconditions give a partial specification of the behaviour of *method*. It is easy in Z to specify contradictions when writing arbitrary axioms, but we reduce the danger of this by writing precondition and postcondition axioms in a standard style: the *PreAx* above is written as:

$$\begin{array}{l} (\forall\, self : Class;\ in : Inputs \\ \quad \mid Precondition \\ \quad \bullet (self, in) \in \operatorname{dom} method) \end{array}$$

while *PostAx* is written as:

$$\begin{array}{l} (\forall\, self, self' : Class;\ in : Inputs;\ out : Outputs \\ \quad \mid (self', out) \in method(self, in) \\ \quad \bullet Postcondition) \end{array}$$

In this paper, all of our methods happen to be deterministic and total, so we use total functions rather than relations, and do not need to specify explicit preconditions. But in the general case, we use precondition axioms to specify lower bounds on the domain of the method, and postcondition axioms to specify the range.

Note that these pre and postcondition axioms often give a *partial* specification (that is, a *loose* specification) of *method* at the point it is declared. Then each subtype adds additional precondition or postcondition axioms to more tightly specify the behaviour of *method* on that subtype. For example, if we add a subtype $Class_2 \subseteq Class$, then we would specify the extra behaviour by adding an extra postcondition axiom:

$$\begin{array}{l} (\forall\, self : Class_2;\ self' : Class;\ in : Inputs;\ out : Outputs \\ \qquad \mid (self', out) \in method(self, in) \\ \qquad \bullet\ ExtraPostcondition) \end{array}$$

This has the effect of giving us *more* information about the possible outputs of *method* when the input object happens to belong to the subtype. (Note how the type of $self'$ is still the original supertype—this ensures that all possible outputs are constrained). If we combine the original axiomatic definition of *method* with the extra postcondition, we see that the effect is to *strengthen* the whole postcondition:

$$\begin{array}{l} method : (Class \times Inputs) \leftrightarrow (Class \times Outputs) \\ \hline (\forall\, self, self' : Class;\ in : Inputs;\ out : Outputs \\ \qquad \mid (self', out) \in method(self, in) \\ \qquad \bullet\ Postcondition \wedge \\ \qquad\quad (self \in Class_2 \Rightarrow ExtraPostcondition)) \end{array}$$

So, in a complex hierarchy of subtypes, the final postcondition axiom for a method will typically contain one implication ($self \in SubClass_i \Rightarrow Post_i$) for each class in the hierarchy—this models the effect of dynamic dispatch in an object-oriented language. If $Post_i$ and $Post_j$ are contradictory, they must belong to disjoint subtypes in the hierarchy. We discuss the issue of overriding a method with contradictory behaviour more in Section 5.

Preconditions are different. If we add an extra precondition axiom:

$$\begin{array}{l} (\forall\, self : Class_2;\ in : Inputs \\ \qquad \mid ExtraPrecondition \\ \qquad \bullet\ (self, in) \in \operatorname{dom} method) \end{array}$$

and combine this with the original axiomatic definition of *method* (showing only the precondition parts) we see that the whole precondition is actually *weakened*, because *more* values are now known to be in the domain of *method*.

$$
\begin{array}{|l}
method : (Class \times Inputs) \leftrightarrow (Class \times Outputs) \\
\hline
(\forall\, self : Class;\ in : Inputs \\
\quad \bullet\ (Precondition \Rightarrow (self, in) \in \operatorname{dom} method) \land \\
\quad\quad (self \in Class_2 \land ExtraPrecondition \Rightarrow (self, in) \in \operatorname{dom} method))
\end{array}
$$

Those readers who are familiar with the usual notions of Z refinement will recognise that this strengthening-postconditions and weakening-preconditions property means that the behaviour of *method* at a subtype (like $Class_2$) is a *refinement* of its behaviour at the supertype. In object-oriented circles, this is called *behavioural subtyping*. Our axiomatic style of specifying methods guarantees behavioural subtyping, and we will have more to say about this in later sections.

2.4 Observations allow model-oriented specification

Given that objects are just members of some given type, which has no internal structure, it is not clear how we *can* write preconditions and postconditions for a method. How can a postcondition compare $self'$ with $self$? We want to specify more than just equality or inequality!

To support model-oriented specification, we declare *observations* of each class, which effectively give us a partial view of the internal state of the object. An observation is simply a total function from the class to some other type. For example:

$$
\begin{array}{|l}
size : Class \rightarrow \mathbb{N} \\
count : Class_2 \rightarrow \mathbb{N}
\end{array}
$$

Since $Class_2 \subseteq Class$, the *size* observation is applicable to $Class_2$ objects as well. So the further down the subtype hierarchy we go, the more observations we can make of an object.

These observations should not be regarded as part of the implementing of the object – an observation *may* be implemented by a data field, but could be implemented by a method which calculates and returns a value, or it may not be implemented at all, because it is defined only for specification purposes (in such cases, all uses of it will be refined into calls to other methods).

3 The MagicBall Example

In order to illustrate our approach on specifying objects and methods in value semantics, we start from a simple example of MagicBall.

3.1 MagicBall specification

Let us say we have an object – a magic ball which has three different sizes: small, medium and large. The changes of sizes are observable.

A specification of such magic balls in Z is:

$[MagicBall]$

$Size ::= small \mid medium \mid large$

$$size : MagicBall \rightarrow Size$$

$$inc : MagicBall \rightarrow (MagicBall \times Size)$$
$$dec : MagicBall \rightarrow (MagicBall \times Size)$$

$$\begin{array}{l}
\forall\, ball, ball' : MagicBall;\ s : Size \\
\quad \bullet\ (inc\ ball = (ball', s) \Rightarrow \\
\qquad size\ ball' = s \land \\
\qquad (size\ ball = small \Rightarrow s = medium) \land \\
\qquad (size\ ball = medium \Rightarrow s = large) \land \\
\qquad (size\ ball = large \Rightarrow s = large)) \\
\quad \land\ (dec\ ball = (ball', s) \Rightarrow \\
\qquad size\ ball' = s \land \\
\qquad (size\ ball = small \Rightarrow s = small) \land \\
\qquad (size\ ball = medium \Rightarrow s = small) \land \\
\qquad (size\ ball = large \Rightarrow s = medium))
\end{array}$$

We call a set of related axioms like the above, which defines the object type MagicBall, an *object specification* or (informally) a *class specification* of MagicBall.

3.2 Implementations of MagicBall

As we pointed out in section 2.4, the MagicBall specification is a partial view of the internal state of an object. We can have many implementations for this specification, each of them may have different number of states, and each implementation must conform to the observations of its specification. These implementations are also called models of the specification in this paper.

Three *models* of the magic ball specification are shown in Fig. 1. For example, model *M1* could be defined in Z as:

$$\begin{array}{l}
M1 ::= S \mid M \mid L \\
size_{M1} == \{S \mapsto small, M \mapsto medium, L \mapsto large\} \\
inc_{M1} == \{S \mapsto M, M \mapsto L, L \mapsto L\} \\
dec_{M1} == \{S \mapsto S, M \mapsto S, L \mapsto M\}
\end{array}$$

Informally, we say that the state spaces of each of these models are subsets of *MagicBall* (or possible instantiations of *MagicBall*). However, we never equate *MagicBall* with an explicit concrete model like $M1$, $M2$ or $M3$, because we want

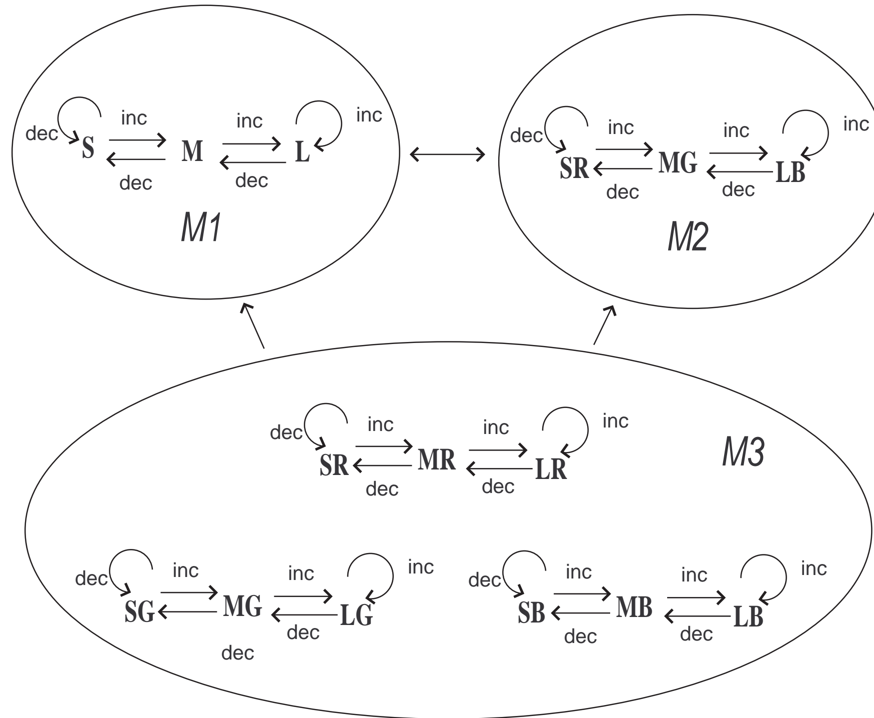

Fig. 1. Models of MagicBall

the freedom to continue making further subtypes, which specify more complex models. Hence, we always keep the *MagicBall* set abstract.

One thing we should notice here is that *all* models of the MagicBall specification must have at least three states, because the axioms specify observations of at least three distinct values.

4 Extending the MagicBall Example with Colour

An extended ColourMagicBall example is given here to show how to specify a subtype by *subsetting*, and how to deal with the frame problem.

4.1 ColourMagicBall specification

A colour magic ball, in addition to its size attribute, has a colour which may be red, green or blue. The changes of colours are observable.

An object type ColourMagicBall as a subtype of MagicBall is specified as a *subset* of MagicBall:

$$\mid \; ColourMagicBall : \mathbb{P}\, MagicBall$$

$$Colour ::= red \mid green \mid blue$$

$$\mid \; colour : ColourMagicBall \rightarrow Colour$$

ColourMagicBall has an extra method *paint*.

$$paint : (ColourMagicBall \times Colour) \rightarrow ColourMagicBall$$
$$\forall\, ball, ball' : ColourMagicBall;\ c : Colour \bullet paint\,(ball, c) = ball' \Rightarrow colour\ ball' = c$$

Obviously in Fig. 1, *M2* and *M3* are *models* of this specification. It is not so obvious that *M1* is also a *model* of the above specification. The easiest way to prove it is that *M1* and *M2* are *isomorphic*. Method *paint* is non-deterministic in the specification, and is not displayed.

One possible $M2$ with paint method could be:

$$
\begin{aligned}
&M2 ::= SR \mid MG \mid LB\\
&size_{M2} == \{SR \mapsto small, MG \mapsto medium, LB \mapsto large\}\\
&colour_{M2} == \{SR \mapsto red, MG \mapsto green, LB \mapsto blue\}\\
&inc_{M2} == \{SR \mapsto MG, MG \mapsto LB, LB \mapsto LB\}\\
&dec_{M2} == \{SR \mapsto SR, MG \mapsto SR, LB \mapsto MG\}\\
&paint_{M2} == \{(SR, red) \mapsto SR, (MG, red) \mapsto SR, (LB, red) \mapsto SR,\\
&\qquad (SR, green) \mapsto MG, (MG, green) \mapsto MG, (LB, green) \mapsto MG,\\
&\qquad (SR, blue) \mapsto LB, (MG, blue) \mapsto LB, (LB, blue) \mapsto LB\}
\end{aligned}
$$

The fact that $M2$ is a model of ColourMagicBall means that when we paint a ball, its size can change. This is perhaps a little surprising, but is simply because we forgot to specify that painting a ball should not change its size. We can do this by adding one more postcondition:

$$\forall\, ball, ball' : ColourMagicBall;\ c : Colour \bullet paint\,(ball, c) = ball' \Rightarrow size\ ball' = size\ ball$$

Note that *M3* is the *model* of this revised ColourMagicBall specification, and neither *M1* nor *M2* anymore. This revised ColourMagicBall is a *subtype* of the MagicBall. Some models of MagicBall may not be models of ColourMagicBall, but all models of ColourMagicBall are models of MagicBall.

4.2 The frame problem

We should also notice that the *inc* and *dec* operation may behave weirdly, i.e. we don't know whether these methods will change the colour of the colour magic

balls or not. In fact, our current axioms allow the *inc* method to mutate a ColourMagicBall into a MagicBall! Most OO programming languages do not support such mutations, but Smalltalk does. We can specify that the type remains unchanged to advoid this:

$$\begin{array}{l}\forall\, ball : ColourMagicBall;\; ball' : MagicBall;\; s : Size \bullet \\ \quad (inc\ ball = (ball', s) \Rightarrow ball' \in ColourMagicBall) \land \\ \quad (dec\ ball = (ball', s) \Rightarrow ball' \in ColourMagicBall)\end{array}$$

Similarly, if we want these inherited methods to leave new observations unchanged (this is the default in most object-oriented programming languages), we can easily specify this by adding some restrictions on *inc* and *dec* for ColourMagicBall:

$$\begin{array}{l}\forall\, ball : ColourMagicBall;\; ball' : MagicBall;\; s : Size \bullet \\ \quad (inc\ ball = (ball', s) \Rightarrow colour\ ball = colour\ ball') \land \\ \quad (dec\ ball = (ball', s) \Rightarrow colour\ ball = colour\ ball')\end{array}$$

This "frame problem" arises when we always want to constrain the inherited methods from changing subtype observations. Stating these no-change facts can be unwieldy and verbose. However, there are situations where inherited methods *do* need to change new attributes, so banning this possibility is undesirable. The verbosity problem could be easily solved by adding "macro" or structural syntax, which may result in an extended Z (same semantics, but extended syntax) or Z tools.

Fig. 2 shows a model of ColourMagicBall $M3$ with unchanged subtype observation inheritance. To make the figure more readable, we omitted the inputs of the paint method, and the bidirectional arrows of all *paint* transitions are not shown.

5 Behavioural Subtyping versus Inheritance

In this section, we discuss the differences between behavioural subtyping, as used in this paper, and inheritance, as used in typical object-oriented programming languages.

Informally, we say that type B is a *behavioural subtype* of type A iff [MRT98]:

- the interface of B conforms to that of A, and
- the methods of B have the same (or refined) behaviour as those of A.

Interface conformance means that B has methods with the same names, and compatible signatures, as the methods of A. It may have additional methods too. In our approach, subtypes are always interface conformant with their supertypes, because the set of objects B is defined to be a subset of A, which means that all the methods of A are automatically applicable to B objects.

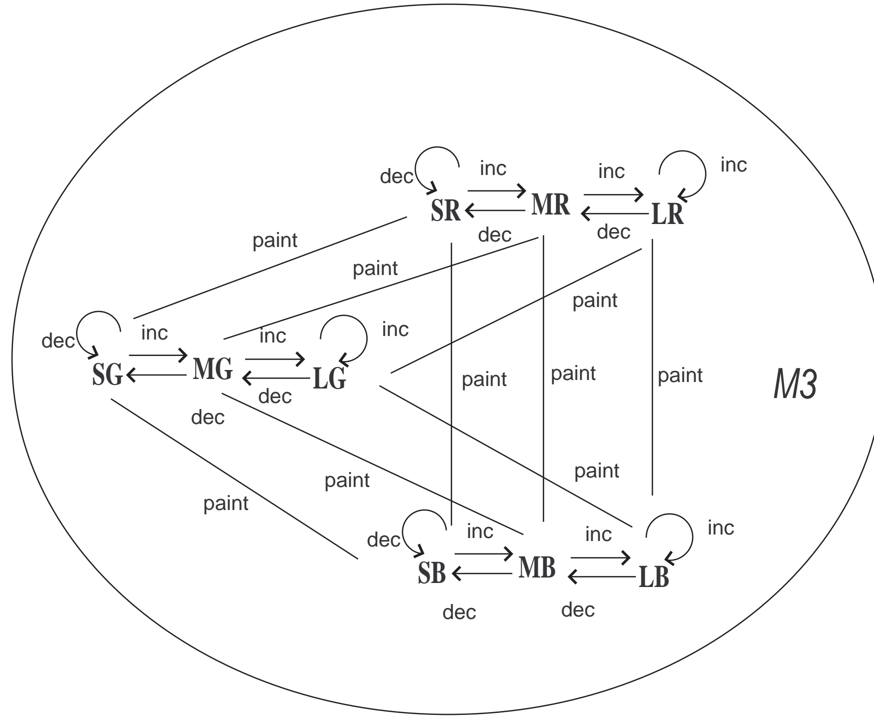

Fig. 2. A model of ColourMagicBall

There are many different ways of defining behaviour, but one simple one is to view the behaviour of an object as being characterized by the set of all the properties (observations) that the behaviour satisfies. To ensure behavioural subtyping, subtypes must preserve all the properties of their supertypes. Typically, they add *more* properties. In our approach, the set of properties associated with a type is simply all the theorems that are derivable from its axioms. Since our subtypes *add* axioms (and cannot retract axioms–impossible in Z), our approach guarantees that subtypes enjoy all the properties of their supertypes if these subtypes exist. The pre/post refinement relationship discussed in Section 2.3 is simply a consequence of this axiomatic extension property.

We see that our approach ensures behavioural subtyping. However, it is common in programming languages to define inheritance hierarchies that are *not* behavioural subtypes, because subtype methods use dynamic dispatching to override the default behaviour of the corresponding supertype methods [LW94]. What happens if we try this in our approach? Is there any way of specifying such *non-monotonic* inheritance hierarchies?

5.1 The bird/emu example

A classic example in the object-oriented literature is birds and emus. The *Bird* superclass has a *canFly* attribute that returns true, but the *Emu* subclass overrides this to return false, because emus are an exception to the default behaviour of birds, which is to fly. We can specify this as follows.

$[Bird]$

$CanFly ::= yes \mid no$

$$
\begin{array}{|l}
canfly : Bird \rightarrow CanFly \\
\hline
\forall\, bird : Bird \bullet canfly\ bird = yes
\end{array}
$$

Now we add the Emu subtype, and try to override *canfly*.

$$
\begin{array}{|l}
Emu : \mathbb{P}\, Bird \\
\hline
\forall\, emu : Emu \bullet canfly\ emu = no
\end{array}
$$

This might look okay, but attempting to 'create an emu' by proving an initialization theorem like $\exists\, e : Emu \bullet true$, fails. In fact, from the above axioms we can prove that $Emu = \emptyset$. This is the lesson, *if one specifies subtype behaviour that is inconsistent with the supertype behaviour, the subtype will be empty.* [1]

Nevertheless, we can obtain some of the desired effect if we are prepared to go back and change the *supertype* specification. Essentially, we must remove the contradiction by modifying the supertype to weaken the faulty assumption that all birds can fly, and instead allow for the possibility of non-flying birds.

$[Bird]$

$$
\begin{array}{|l}
canfly : Bird \rightarrow CanFly \\
\hline
\forall\, bird : Bird \mid bird \notin Emu \bullet canfly\ bird = yes
\end{array}
$$

$$
\begin{array}{|l}
Emu : \mathbb{P}\, Bird \\
\hline
\forall\, emu : Emu \bullet canfly\ emu = no
\end{array}
$$

Here we have the effect that is sometimes desired in object-oriented programs: the supertype-only objects (the ordinary, non-emu birds) have $canfly = yes$,

[1] An alternative approach would be to specify subtypes using $\mathbb{P}_1$ rather than $\mathbb{P}$, to ensure that subtypes are non-empty. But this would make the whole specification inconsistent. We prefer the $\mathbb{P}$ approach, since it localizes the effects of inconsistency to the subtype that causes it.

whereas the subtype objects (the emus) have $\mathit{canfly} = \mathit{no}$. Effectively, the complete set of birds is a union partitioned by two subclasses: ordinary birds which can fly and emu-like birds which can't fly.

Note that the resulting system still satisfies behavioural subtyping, because at the *Bird* level, the value of the *canfly* attribute on emus is unknown, while the *Emu* level simply strengthens this by adding the property that $\mathit{canfly} = \mathit{no}$ for emus. The bird-only objects ($\mathit{Bird} \setminus \mathit{Emu}$) have different behaviour to the *Emu* subtype objects, but behavioural subtyping still holds between the whole *Bird* set and its *Emu* subset.

The lesson here is: *we can specify systems where the supertype-only objects have different behaviour to the subtype objects, but to do this, we must carefully specify the supertype behaviour to allow exceptions in the subtype.*

In object-oriented programming languages, this non-monotonic overriding effect can be implemented by late binding of methods, without modifying the supertype code. But in our strictly behavioural subtyping approach, the supertype specifications must be modified. This insistence on purity could be regarded as a disadvantage of our approach, but we prefer to regard it as a desirable discipline that leads to clearer specifications that are easier to reason about.

6 The Quadrilaterals Example

In this section, we specify the widely used quadrilaterals example [SBC92] for comparison with other styles of object orientation in Z.

We also introduce a more familiar object-oriented notation to specify the example. For instance, instead of using

$$\mathit{method} : (\mathit{Class} \times \mathit{Inputs}) \rightarrow (\mathit{Class} \times \mathit{Outputs})$$

to declare an operation, we declare it as a special infix operator

$$_\,{}_{\bullet}\mathit{method}(_) : (\mathit{Class} \times \mathit{Inputs}) \rightarrow (\mathit{Class} \times \mathit{Outputs})$$

so that the method can be called as $(x', o) = x\,{}_{\bullet}\mathit{method}(in)$. This looks more like traditional object-oriented syntax for method calls. Note that ${}_{\bullet}\mathit{method}$ is a legal *Word* in Standard Z: it is a subscripted bullet followed by an alphabetic name, and would be written in Unicode as '$\searrow \bullet \nwarrow$ *m e t h o d*'. Similarly, we sometimes declare observation functions as postfix operators (and add the subscript bullet), so that we can write calls to them as $x\,{}_{\bullet}\mathit{size}$.

6.1 The Quadrilaterals example with OO-like syntax

The classes of quadrilaterals are shown in Fig. 3. It is assumed that readers are familiar with the context of this example from the specifications in [SBC92].

$$[\mathit{Vector}, \mathit{Scalar}]$$

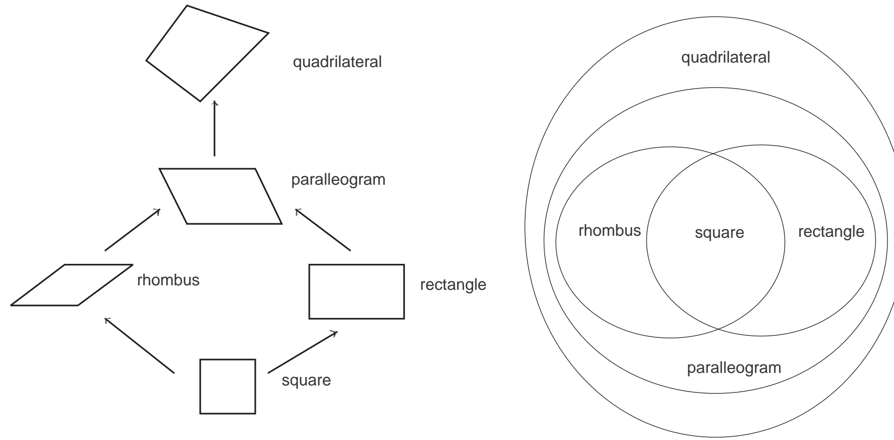

Fig. 3. Quadrilaterals

Some operations of *vector* are defined as the following. Note that we use polar coordinate to represent an angle. In our approach, there is no difficulty to define cartesian and polar coordinates at the same time, because we treat them as observable properties rather than internal representations.

$$
\begin{array}{|l}
_ + _ : \mathit{Vector} \times \mathit{Vector} \rightarrow \mathit{Vector} \\
_ \cdot _ : \mathit{Vector} \times \mathit{Vector} \rightarrow \mathit{Scalar} \\
\mathbf{0} : \mathit{Vector} \\
0, 1 : \mathit{Scalar} \\
_ + _ \ : \mathit{Scalar} \times \mathit{Scalar} \rightarrow \mathit{Scalar} \\
_ - _ \ : \mathit{Scalar} \times \mathit{Scalar} \rightarrow \mathit{Scalar} \\
_ / _ \ : (\mathit{Scalar} \times \mathit{Scalar}) \nrightarrow \mathit{Scalar} \\
_ \times _ \ : (\mathit{Scalar} \times \mathit{Scalar}) \rightarrow \mathit{Scalar} \\
_ _\bullet x \ : \mathit{Vector} \rightarrow \mathit{Scalar} \\
_ _\bullet y \ : \mathit{Vector} \rightarrow \mathit{Scalar} \\
_ _\bullet \rho \ : \mathit{Vector} \rightarrow \mathit{Scalar} \\
_ _\bullet \theta \ : \mathit{Vector} \rightarrow \mathit{Scalar} \\
\mathit{tan}(_) : \mathit{Scalar} \nrightarrow \mathit{Scalar} \\
\hline
\mathbf{0} _\bullet \rho = 0 \\
\forall\, q : \mathit{Quadrilateral} \bullet \\
\qquad q _\bullet \rho \times q _\bullet \rho = q _\bullet x \times q _\bullet x + q _\bullet y \times q _\bullet y \wedge \\
\qquad \mathit{tan}(q _\bullet \theta) = q _\bullet y / q _\bullet x \\
[\mathit{definitions\ omitted}]
\end{array}
$$

Then we define the specification of quadrilaterals with four edges of $v1$, $v2$, $v3$, and $v4$:

$[\mathit{Quadrilateral}]$

$$
\begin{array}{l}
_\bullet v1, _\bullet v2, _\bullet v3, _\bullet v4, _\bullet position : Quadrilateral \rightarrow Vector \\
_\bullet edges : Quadrilateral \rightarrow Vector \times Vector \times Vector \times Vector \\
\hline
\forall\, q : Quadrilateral \bullet \\
\qquad q \bullet edges = (q \bullet v1, q \bullet v2, q \bullet v3, q \bullet v4) \wedge \\
\qquad q \bullet v1 + q \bullet v2 + q \bullet v3 + q \bullet v4 = \mathbf{0}
\end{array}
$$

$$
\begin{array}{l}
Parallelogram : \mathbb{P}\, Quadrilateral \\
Rhombus : \mathbb{P}\, Parallelogram \\
Rectangle : \mathbb{P}\, Parallelogram \\
Square : \mathbb{P}\, Rectangle
\end{array}
$$

$$
\begin{array}{l}
\hline
Square = Rectangle \cap Rhombus \\
\forall\, q : Quadrilateral \bullet \\
\qquad q \bullet v1 + q \bullet v3 = \mathbf{0} \Leftrightarrow q \in Parallelogram \\
\forall\, p : Parallelogram \bullet \\
\qquad p \bullet v1 \bullet \rho = p \bullet v2 \bullet \rho \Leftrightarrow p \in Rhombus \wedge \\
\qquad (p \bullet v1).(p \bullet v2) = 0 \Leftrightarrow p \in Rectangle
\end{array}
$$

In short, a quadrilateral can be moved around by changing its position. For all except general quadrilaterals, the angle between two adjacent sides is well defined.

$$
\begin{array}{l}
\bullet move() : (Quadrilateral \times Vector) \rightarrow Quadrilateral \\
_\bullet angle : Parallelogram \rightarrow Scalar \\
\bullet shear() : (Quadrilateral \times Scalar) \rightarrow Quadrilateral \\
\hline
\forall\, q, q' : Quadrilateral;\ v : Vector \bullet \\
\qquad q' = q \bullet move(v) \Rightarrow \\
\qquad\qquad (q' \bullet position = q \bullet position + v \wedge q' \bullet edges = q \bullet edges) \\
\forall\, q : Parallelogram;\ angle : Angle \bullet \\
\qquad q \bullet angle = q \bullet v2 \bullet \theta - q \bullet v1 \bullet \theta \\
\forall\, q, q' : Quadrilateral;\ angle : Scalar \bullet q' = q \bullet shear(angle) \Rightarrow \\
\qquad (q' \bullet position = q \bullet position \wedge \\
\qquad (q' \bullet v1 \bullet x = q \bullet v1 \bullet x + q \bullet v1 \bullet y \times tan(angle) \wedge q' \bullet v1 \bullet y = q \bullet v1 \bullet y \wedge \\
\qquad q' \bullet v2 \bullet x = q \bullet v2 \bullet x + q \bullet v2 \bullet y \times tan(angle) \wedge q' \bullet v2 \bullet y = q \bullet v2 \bullet y \wedge \\
\qquad q' \bullet v3 \bullet x = q \bullet v3 \bullet x + q \bullet v3 \bullet y \times tan(angle) \wedge q' \bullet v3 \bullet y = q \bullet v3 \bullet y \wedge \\
\qquad q' \bullet v4 \bullet x = q \bullet v4 \bullet x + q \bullet v4 \bullet y \times tan(angle) \wedge q' \bullet v4 \bullet y = q \bullet v4 \bullet y))
\end{array}
$$

Here we give a clear and explicit definition for shearing method. Apart from the readability and completeness comparing with other proposed approaches (the definition of shearing is omitted in all other object-oriented Z approaches in [SBC92]), the shearing function defined here is obviously more intuitive and reasonable.

In our shearing function, any quadrilateral can be sheared. A square may become a rhombus after shearing, although the type conversion could make it

harder to reason about. And many object oriented programming languages can not easily support this feature.

Like other approaches in [SBC92], in order to avoid type conversion problems, we can also limit the shearing function on quadrilaterals except squares, rhombi and rectangles.

$$_\bullet shear(_) : (Quadrilateral \times Scalar) \nrightarrow Quadrilateral$$

$$\mathrm{dom}(\mathrm{dom}\, shear) = Quadrilateral \setminus (Rhombus \cup Rectangle)$$

6.2 A Drawing system of quadrilaterals

A drawing system can be simply defined as a sequence of quadrilaterals:

$$DrawingSystem == \mathrm{seq}\, Quadrilateral$$

$$\begin{array}{l}
\bullet add() : (DrawingSystem \times Quadrilateral) \rightarrow DrawingSystem \\
\bullet delete() : (DrawingSystem \times \mathbb{N}) \rightarrow DrawingSystem \\
\bullet move(,_) : (DrawingSystem \times \mathbb{N} \times Vector) \rightarrow DrawingSystem \\
\bullet angle() : (DrawingSystem \times \mathbb{N}) \rightarrow Scalar \\
\bullet shear(,_) : (DrawingSystem \times \mathbb{N} \times Scalar) \rightarrow DrawingSystem
\end{array}$$

$$\begin{array}{l}
\forall\, ds : DrawingSystem;\ q : Quadrilateral;\ n : \mathbb{N};\ v : Vector \mid n \leq \#ds \bullet \\
\quad (ds' = ds \bullet add(q) \Rightarrow ds' = ds \frown \langle q \rangle \wedge \\
\quad ds' = ds \bullet delete(n) \Rightarrow ds' = ((\mathrm{dom}\, ds) \setminus \{n\}) \upharpoonright ds \wedge \\
\quad ds' = ds \bullet move(n, v) \Rightarrow \\
\quad ds' = ((1 \mathinner{..} n-1) \lhd ds) \frown (ds\ n) \bullet move(v) \frown ((n+1 \mathinner{..} \#ds) \lhd ds) \wedge \\
\quad ds' = ds \bullet shear(n, v) \Rightarrow \\
\quad ds' = ((1 \mathinner{..} n-1) \lhd ds) \frown (ds\ n) \bullet shear(v) \frown ((n+1 \mathinner{..} \#ds) \lhd ds) \wedge \\
\quad ds \bullet angle(n) = (ds\ n) \bullet angle)
\end{array}$$

Then we can *add* or *delete* quadrilaterals, inquiring *angles*, *move* and *shear* each quadrilateral in the drawing system respectively.

As a simple rule, if a class is a composition of objects without any other distinguishable observation properties, we can explicitly specify it as a sequence of objects. Otherwise, we must define it as a given set or a subset of a given set (Because the space limit in this paper, we will elaborate this problem in another paper). For instance, we declare the drawing system as:

$$[DrawingSystem]$$

and plus an attribute of composition:

$$_.comps : DrawingSystem \rightarrow \mathrm{seq}\, Quadrilateral$$

$$
\begin{array}{l}
_ \bullet add(_) : (DrawingSystem \times Quadrilateral) \to DrawingSystem \\
_ \bullet delete(_) : (DrawingSystem \times \mathbb{N}) \to DrawingSystem \\
_ \bullet move(_,_) : (DrawingSystem \times \mathbb{N} \times Vector) \to DrawingSystem \\
_ \bullet angle(_) : (DrawingSystem \times \mathbb{N}) \to Scalar \\
_ \bullet shear(_,_) : (DrawingSystem \times \mathbb{N} \times Scalar) \to DrawingSystem \\
\hline
\forall\, ds : DrawingSystem;\ q : Quadrilateral;\ n : \mathbb{N};\ v : Vector \mid n \leq \#ds \bullet \\
\quad (ds' = ds \bullet add(q) \Rightarrow ds' \bullet comps = ds \bullet comps \frown \langle q \rangle \land \\
\quad ds' = ds \bullet delete(n) \Rightarrow \\
\qquad ds' \bullet comps = ((\mathrm{dom}\, ds \bullet comps) \setminus \{n\}) \upharpoonright ds \bullet comps \land \\
\quad ds' = ds \bullet move(n, v) \Rightarrow \\
\qquad ds' \bullet comps = ((1 \mathinner{..} n-1) \lhd ds \bullet comps)^\frown \\
\qquad\quad (ds \bullet comps\ n) \bullet move(v)^\frown \\
\qquad\quad ((n+1 \mathinner{..} \#ds \bullet comps) \lhd ds \bullet comps) \land \\
\quad ds' = ds \bullet shear(n, v) \Rightarrow \\
\qquad ds' \bullet comps = ((1 \mathinner{..} n-1) \lhd ds \bullet comps)^\frown \\
\qquad\quad (ds \bullet comps\ n) \bullet shear(v)^\frown \\
\qquad\quad ((n+1 \mathinner{..} \#ds \bullet comps) \lhd ds \bullet comps) \land \\
\quad ds \bullet angle(n) = (ds \bullet comps\ n) \bullet angle)
\end{array}
$$

7 Explicit Models for Animation and Proof

Using an abstract model of objects is convenient for specification, but makes it difficult to animate specifications (for validation and testing purposes), because there is no explicit finite model of objects. No existing Z animators are capable of animating abstract objects and axiomatic functions and methods over those objects, in the style that we have used.

In Section 3 we showed an explicit model for the *MagicBall* specification that could easily be animated. In this section, we briefly show how a specification of a hierarchy of classes could be converted into an explicit model, which would be more suitable for animation. Also, seeing one possible instantiation of the *Object* given type gives insight into how our specification style works.

First of all, we build a hierarchy of state spaces, in the same way that Object-Z does. Schema inclusion is useful here to model inheritance. Usually, the state space of each class contains just the attributes that were defined as observation functions, but it is possible to write specifications that require additional implicit attributes.

$$
\begin{array}{l}
MagicBallState \mathrel{\widehat=} [size : Size] \\
ColourMagicBallState \mathrel{\widehat=} [MagicBallState;\ colour : Colour]
\end{array}
$$

Next we define a free type that ranges over all the possible object types in the system. Note that we are assuming a closed, non-extensible system here!

$$
\begin{array}{l}
Object ::= mball \langle\!\langle MagicBallState \rangle\!\rangle \\
\qquad\qquad \mid cmball \langle\!\langle ColourMagicBallState \rangle\!\rangle
\end{array}
$$

Now we can define the hierarchy of subsets, starting from the bottom of the hierarchy and defining each supertype to be the union of all its subtypes plus its own members. This is like the $Class\downarrow$ type in Object-Z.

$$ColourMagicBall == \operatorname{ran} cmball$$
$$MagicBall == ColourMagicBall \cup \operatorname{ran} mball$$

Next we define the observation functions, so that they select the desired field out of a class state and out of all of its subclass states.

$$size == (\lambda\, m : \operatorname{ran} mball \bullet m.size) \cup (\lambda\, cm : \operatorname{ran} cmball \bullet cm.size)$$

$$colour == (\lambda\, cm : \operatorname{ran} cmball \bullet cm.colour)$$

Finally we can define each method as a relation that satisfies all the relevant preconditions and postconditions. Preconditions and postconditions that were added in a subtype are guarded by a membership constraint so that they are only applicable to that subtype. For example, *inc* can be defined as:

$$\begin{aligned} inc == \{ & ball, ball' : MagicBall;\ s : Size \mid \\ & size\ ball' = s \land \\ & (size\ ball = small \Rightarrow s = medium) \land \\ & (size\ ball = medium \Rightarrow s = large) \land \\ & (size\ ball = large \Rightarrow s = large)) \land \\ & (ball \in ColourMagicBall \Rightarrow ball' \in ColourMagicBall) \land \\ & (ball \in ColourMagicBall \Rightarrow colour\ ball = colour\ ball') \} \end{aligned}$$

We sometimes find it useful to think of this explicit model as we specify objects abstractly, but we do NOT propose that one should ever write such an explicit model. It is just one possible instantiation of the *Object* type. It is more verbose (the case analysis style explodes as more classes are added), and it is not extensible. We are not yet sure whether Z theorem provers work better with the abstract or explicit model, but we suspect the abstract model is preferable. On the other hand, it is clear that the explicit model is more suitable for animation than the abstract specification. It would be interesting to develop a tool that transformed the abstract style of specification into the explicit model for animation purposes.

8 Related Work

Our goal of formally specifying object-oriented systems in Z is to specify the object-oriented concepts in first order logic and set theory, and utilize the existing powerful tools of Z. This greatly reduces the burden of learning a new object-oriented formal specification language. The most important thing in our approach is that we give an abstract, concise and consistent perception of object types, subtyping and inheritance.

The abstract view of observable object behaviour closely relates to the research on algebraic specification of abstract data types [CGK$^+$]. Most of the algbaic specification use the initial algebra for the semantics of a specification, but we use refinement theory for modelling and interpreting object-oriented constructs, and defining behavioural subtyping and inheritance.

Our approach is significantly different from any other object oriented approaches in Z. Firstly, we interpret object-oriented concepts in standard Z, rather than extending Z like Object-Z [DKRS91], MOOZ, OOZE, Z++, and ZEST [SBC92]. Secondly, most object-oriented Z extensions explicitly model object state (typically by state schemas, where Hall [Hal90] is an exception), whereas we use an abstract model of objects (given types or subsets of given types). Thirdly, we model methods using functions and relations, whereas most other object oriented styles use operation schemas. For example, Hall's style [Hal90] and ZERO [SBC92]. We use value semantics and separate object identity from its representation, which allows us to consolidate *object type* or *class* with Z *type*. It also makes it possible for us to use subsetting to model inheritance, and gives us an constructive way to build behavioural subtyping.

OOZE [AG91] is a Z-like notation, built on top of order-sorted algebras—a very different semantic basis to standard Z. It supports inheritance, sophisticated modularization and dynamic binding (including the ability for subtypes to override supertype behaviour in non-compatible ways). Its use of axiomatic specification style is similar to ours, but that is the only similarity. We map object-oriented constructs into standard Z sets and relations, which gives a simpler semantics and is more familiar to Z users.

[Rob00] shows how a loose axiomatic specification can be proved to be refined by a constructive concrete model.

9 Conclusions

We have shown that an elegant and simple object-oriented specification style is possible in Z. Modelling objects as black boxes makes it possible to specify subtype hierarchies using subset constraints. Our approach does not provide much in the way of hiding or encapsulation facilities, but this is a problem with Z and standard Z—the simple section mechanism is not sufficient to support modularity. Nevertheless, the way we define methods provides a limit encapsulation. We can group methods of an *object type* together by searching the whole specification for methods which take this *object type* as the first parameter. This could also enhance the extensibility of software specification by adding more methods in other parts of the specification later when it is needed.

Our style of specification is one that can be reasoned about using the standard Z theorem provers, but is not supported by existing animation tools, because of its abstractness. It would be an interesting challenge to try and develop animation support for this abstract style. One promising approach might be to develop a tool that translates our abstract style of object specification into an explicit object model that existing animators can handle.

Our approach uses value semantics rather than references, but again, we believe this is the most elegant approach for a specification language, and closest to the spirit of Z. It is easy to simulate (explicit) reference semantics in a value-semantics specification language (using seq *Quadrilateral* like in Sect. refsec:quads, or $Ref \pfun Object$ mappings), but the converse is not true. Reference semantics is harder to reason about, due to the aliasing problems. An advantage of using explicit references is that the specifier can use them only where necessary, and in a controlled and localised way, thus preserving the ease of reasoning as much as possible.

An interesting, and intrinsic, feature of our approach is that subtypes preserve all the properties of their supertypes. In other words, our specification style enforces behavioural subtyping. This is a restriction that might be considered undesirable in a programming language, where the purpose of inheritance is often code reuse rather than behaviour specialization, so subclasses often override inherited methods with incompatible (non-monotonic) behaviour. With our approach, if one wants to override the behaviour of a supertype in a non-monotonic fashion, one must instead reorganise the hierarchy so that the supertype and subtype become siblings, and their common parent specifies just their common behaviour. This is often better style anyway, and we believe that in a specification language it is good discipline for subtype hierarchies to be behavioural hierarchies.

References

[AG91] Antonio J. Alencar and Joseph A. Goguen. OOZE: An object-oriented Z environment. In P. America, editor, *Proceedings ECOOP'91*, LNCS 512, pages 180–199, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[CGK+] Maura Cerioli, Martin Gogolla, Hlne Kirchner, Bernd Krieg-Brckner, Zhenyu Qian, and Markus Wolf (Eds.). Algebraic system specification and development: Survey and annotated bibliography - second edition -.

[DKRS91] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, The University of Queensland, St. Lucia 4072, Australia, 1991.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[Hal90] J. A. Hall. Using Z as a specification calculus for object-oriented systems. In D. Bjørner, C. A. R. Hoare, and H. Langmaack, editors, *VDM and Z – Formal Methods in Software Development*, volume 428 of *LNCS*, pages 290–318. VDM-Europe, Springer-Verlag, 1990.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Mey97] B. Meyer. *Object-Oriented Software Construction, Second Edition.* The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[MRT98] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A type theory for software architectures. Technical Report UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.

[Rob00] Ken Robinson. Reconciling axiomatic and model-based specification using the B method. In *ZB'2000 – International Conference of B and Z Users*, volume 1878 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 95–106, Helsington, York, UK YO10 5DD, August 2000. Department of Computer Science – University of York.

[SBC92] S. Stepney, R. Barden, and D. Cooper. *Object Orientation in Z*. workshops in computing. Springer-Verlag, 1992.