A survey of object orientation in Z

by Susan Stepney, Rosalind Barden and David Cooper

Two technologies offer much to the software industry; formal specification aids precision and object orientation aids structuring. One popular formal specification language is Z. In this paper, we survey techniques for adding object-oriented structuring to Z and look at three of them [1–3] in more detail.

1 Introduction

Z is one of the more popular formal specification languages [4–7]. It makes use of *schemas* to structure specifications [8]. However, the message coming from many authors is that the Z schema is not sufficient for structuring large specifications [9]. There are many proposals for extensions to provide various degrees of modularity. Some of these are based on simple textual devices, such as formal chapters with facilities including import and export statements, generic parameters and library chapters [10, 11]. Object orientation is a popular approach in other areas of software engineering, promising many benefits in structuring, understandability, incremental development and reuse. Many authors are now proposing a more object-oriented approach to formal specification in order to gain these benefits.

Many object-oriented variants of Z have appeared in the literature, attemping to provide Z with the benefits of object orientation's structuring mechanisms. These include Hall's style [1, 12], Schuman and Pitt's variant [2, 13], Object-Z [3, 14], using Z with HOOD [15, 16], OOZE [17], Whysall's approach [18, 19], Z^{++} [20], MooZ [21] and Cusack's Object Oriented Z [22]. The first three of these are the best documented, and we illustrate them in detail, by specifying the same example in each. This example is specified in Z, in Hall's style, with Schuman and Pitt's variant and in Object-Z. Other approaches are summarised later. We also discuss how well Z and the object-oriented approach mesh.

Object-oriented means different things to different people. We assume fairly standard definitions of objects, classes, and inheritance as given in, for example, References 23–25. Hence, we say that an **object** has state, exhibits well defined behaviour and has a unique identity. Z is a good starting point for specifying object-oriented systems, since it is state-based and can define operations (behaviour) in terms of changes to the state. However, it fails when trying to specify classes. **Classes** are the abstraction mechanism; the structure and behaviour of similar objects are defined in their common class. Z has no support for bundling together the state and operation definitions into a class definition. Hence, it cannot specify inheritance. **Inheritance** is a powerful structuring mechanism that allows classes to be specified in an incremental way. Conformant inheritance specifies behaviourally compatible descendants, which can always be used in place of their ancestors. Non-conformant inheritance captures the notion of similarity, rather than compatibility, and operations can be modified or removed.

As Wegner [24] points out, data abstraction (encapsulation) is an orthogonal concept to inheritance (structuring). In this paper, we are more interested in the structuring aspects of the objected oriented approach, and we emphasise these. We also do not address the interesting research area of the subsequent development of code from an object-oriented specification.

2 Conventional Z

In this Section, we specify in Z the example chosen to illustrate each of the identified object-oriented approaches; it has been written in a style to aid such illustration. The specification chosen is that of different sorts of quadrilaterals, as may be needed for a drawing package. For an introduction to the Z notation, see, for example, Reference 5.

2.1 Vectors and scalars

Vectors are used to specify the edges and position of a quadrilateral. The length of a vector is a scalar, which is not further defined.

$[VECTOR, SCALAR]$

The vector operations addition and modulus (length) are used, with their conventional definitions. The relation $\perp$ holds if two vectors are perpendicular. **0** is the null vector.

$$
\begin{array}{|l}
\_ + \_ : VECTOR \times VECTOR \rightarrow VECTOR \\
|\_| : VECTOR \rightarrow SCALAR \\
\_ \perp \_ : VECTOR \leftrightarrow VECTOR \\
\mathbf{0} : VECTOR \\
\hline
\text{(definitions omitted)} \\
\end{array}
$$

2.2 Edges

The edges of a general four-sided closed figure can be specified by a sequence of four vectors that sum to zero.

$$
\begin{array}{|l}
\textit{Edges} \\
\hline
edge : \text{seq } VECTOR \\
\hline
\# edge = 4 \\
edge1 + edge2 + edge3 + edge4 = \mathbf{0} \\
\end{array}
$$

Five kinds of four-sided figure, defined by various constraints on their edges, are distinguished.

$$EDGEKIND ::= quadrilateral \mid parallelogram \mid rhombus \mid rectangle \mid square$$

A *quadrilateral* has non-zero edges. Notice this definition allows edges to cross. If this is not desired, a stronger constraint should be used.

$$
\begin{array}{|l}
\textit{IsaQuadrilateral} \\
\hline
Edges \\
\hline
0 \notin \text{ran } v \\
\end{array}
$$

A *parallelogram* is a *quadrilateral* with opposite *Edges* equal in length and opposite in direction.

$$
\begin{array}{|l}
\textit{IsaParallelogram} \\
\hline
Edges \\
\hline
IsaQuadrilateral \\
edge1 + edge3 = \mathbf{0} \\
\end{array}
$$

A *rhombus* is a *parallelogram* with adjacent sides of the same length.

$$
\begin{array}{|l}
\textit{IsaRhombus} \\
\hline
Edges \\
\hline
IsaParallelogram \\
|edge1| = |edge2| \\
\end{array}
$$

A *rectangle* is a *parallelogram* with perpendicular adjacent sides.

$$
\begin{array}{|l}
\textit{IsaRectangle} \\
\hline
Edges \\
\hline
IsaParallelogram \\
edge1 \perp edge2 \\
\end{array}
$$

A *square* is a *rhombus* and a *rectangle*.

$$
\begin{array}{|l}
\textit{IsaSquare} \\
\hline
Edges \\
\hline
IsaRhombus \\
IsaRectangle \\
\end{array}
$$

2.3 A quadrilateral

For something like a drawing package, the position of the quadrilateral is also needed. Therefore, a general quadrilateral is specified by its edges, its position and its kind (Fig. 1).

$$
\begin{array}{|l}
\textit{Figure} \\
\hline
Edges \\
position : VECTOR \\
kind : EDGEKIND \\
\hline
(IsaQuadrilateral \land kind = quadrilateral) \\
\lor\ (IsaParallelogram \land kind = parallelogram) \\
\lor\ (IsaRhombus \land kind = rhombus) \\
\lor\ (IsaRectangle \land kind = rectangle) \\
\lor\ (IsaSquare \land kind = square) \\
\end{array}
$$

The predicate ensures that the edges of the four-sided figure are consistent.

2.4 Operations on a figure

2.4.1 Moving the figure: a figure can be moved (translated) by changing its position component.

$$
\begin{array}{|l}
\textit{Move} \\
\hline
\Delta\ Figure \\
move? : VECTOR \\
\hline
edge' = edge \\
position' = position + move? \\
kind' = kind \\
\end{array}
$$

2.4.2 Querying the angle between two edges: a general quadrilateral can have four different internal angles. However, parallelograms and rhombi have only two interior angles, and these are related; one is π minus the other. Rectangles and squares are even simpler. Therefore, it makes sense for all except general quadrilaterals to query the angle between two adjacent sides.

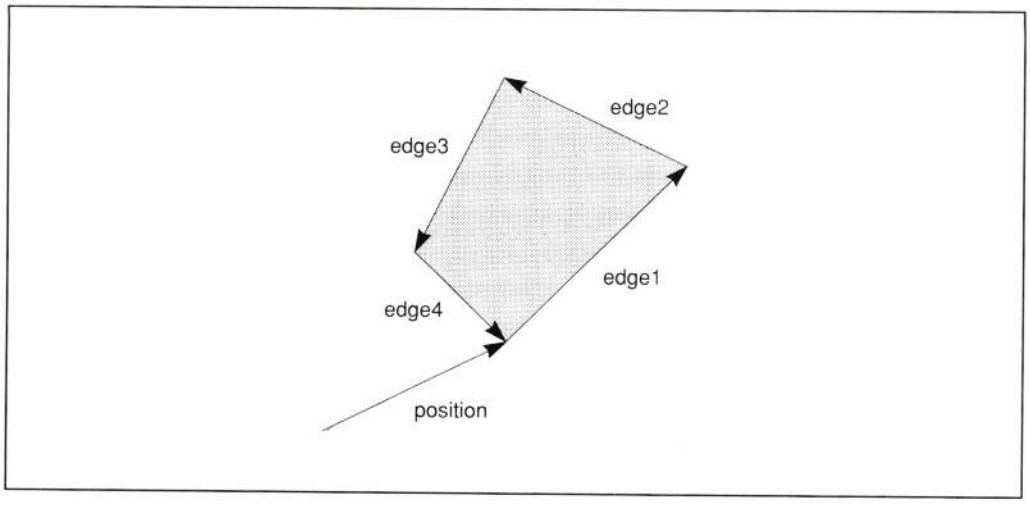


Fig. 1

[*ANGLE*]

$\angle : VECTOR \times VECTOR \leftrightarrow ANGLE$
$rightAngle : ANGLE$

$\forall\, v, w : VECTOR \mid v \perp w \bullet \angle(v, w) = rightAngle$
(rest of definition omitted)

Angle

Ξ *Figure*
$a! : ANGLE$

$(kind \in \{parallelogram, rhombus\} \land$
$a! = \angle(edge1, edge2))$
$\lor$
$(kind \in \{rectangle, square\} \land$
$a! = rightAngle)$

The separate predicate for squares and rectangles is not strictly necessary (it is a consequence of the other definition and the predicate on edges), but it is included explicitly to emphasise this fact.

2.4.3 Shearing the figure: the operation of shearing makes sense for general quadrilaterals and parallelograms, but not for squares, rhombi and rectangles, since the operation does not maintain their invariants.

[*SHEAR*]

Shear

Δ *Figure*
$s? : SHEAR$

$kind \in \{quadrilateral, parallelogram\} \land$
(rest of definition omitted)

Notice that it is not strictly necessary to include the condition on the edge kind, since for a non-null shear it would be impossible to satisfy both the shear and edge conditions for the other kinds.

2.5 Promoting to a drawing system

The state of the drawing system consists of a mapping from identifiers to figures.

[*ID*]

DrawingSystem

$drawing : ID \leftrightarrow Figure$

Analogues to the operations defined on individual figures are required for the complete drawing system. A standard technique for doing this is *promotion* [26, 27]. First, define a general updating schema that performs an as yet unspecified change to a particular figure in the drawing system.

Φ *Update*

Δ *DrawingSystem*
Δ *Figure*
$id? : ID$

$id \in \text{dom } drawing$
$\theta\, Figure = drawing\ id?$
$drawing' = drawing \oplus \{id? \mapsto \theta\, Figure'\}$

The operations on an individual figure can be promoted to operations on a figure in the drawing system, by conjoining them with the general updating schema and hiding the *Figure*.

$MoveDS \triangleq (\Phi\, Update \land Move) \backslash \Delta\, Figure$
$AngleDS \triangleq (\Phi\, Update \land Angle) \backslash \Delta\, Figure$
$ShearDS \triangleq (\Phi\, Update \land Shear) \backslash \Delta\, Figure$

Adding or deleting a figure could be promoted in an analogous manner. However, that is unnecessary for our purposes; the drawing system operations can be defined directly. The operation *AddFigure* adds a new figure to the *DrawingSystem*.

AddFigure

Δ *DrawingSystem*
$f? : Figure$
$id? : ID$

$id? \notin \text{dom } drawing$
$drawing' = drawing \cup \{id? \mapsto f?\}$

The operation *DeleteFigure* deletes an existing figure from the *DrawingSystem*.

DeleteFigure

Δ *DrawingSystem*
$id? : ID$

$id? \in \text{dom } drawing$
$drawing' = \{id?\} \lhd drawing$

2.6 Summary of the Z approach

In this simple example, all the state variables are used by all the 'classes' but, in a more complex example, it could well be that classes lower down the hierarchy wish to introduce new state variables. The style used here is to include all possible state variables for all objects in the class hierarchy, whether relevant or not. If a new subclass with new state variables is added later, the state associated with every existing object in the hierarchy would change. This style also requires the use of a *kind* tag and a sort of 'case statement' selection, based on this tag, in each operation. Adding a new subclass means updating each operation to know about it. For larger 'inheritance' systems with more state and more operations, such a style would become very clumsy.

3 Hall's style [1]

Hall [1] introduces some conventions for an object-oriented specification style. Brownbridge [12] describes a substantial implementation project, where this style was used successfully.

3.1 Overview of Hall's style

Hall's style adds no new features to Z; a specification written in this style has the advantage of a sound theoretical base [6]. The style consists of *conventions* for writing an object-oriented specification.

There are five main features of the style discussed in Hall's paper:

- conventions for modelling object states.
- use of object identities to refer to objects and express their individuality.
- a convention for expressing the state of a system in terms of the objects it contains.
- use of object identities to model relationships between objects.
- a method of defining operations in terms of single objects, and calculating their effect on the whole system or on defined sets of objects.

Another three aspects of the style are identified, but not covered, in Hall's paper. These are a convention for modelling classes and their relationships; a convention for representing meta-class information; guidance on the meaning of inheritance and the description of subclass states and operations. Hall discusses the way in which this approach contrasts with the conventional style of specification; he also promises future support for inheritance.

3.2 Figures and the drawing system

Instead of describing the different kinds of four-sided figure, which is similar to before, we concentrate on illustrating Hall's style for defining state and operations. In describing the state, we see the way in which the object-oriented approach of giving each object a self property is used and how the use of functions ensures uniqueness of the various objects.

A figure is defined using an identifier (the 'self' notion of object-oriented design), as well as its other properties already defined in Section 2.3.

Figure

$self : ID$
$Edges$
$position : VECTOR$
$kind : EDGEKIND$

(invariant as in Section 2.3)

The whole drawing system may be described in terms of a set of identifiers and a function that relates the identifier to the instance of a figure. This function ensures the individuality of the figures. Note that we are assuming here that the drawing system deals only with four-sided figures.

DrawingSystem

$figure : \mathbb{P}\ Figure$
$idFigure : ID \pfun Figure$

$idFigure = \{f : figure \bullet f.self \mapsto f\}$

The drawing system is thus described in terms of the objects it contains. If, in turn, this was part of some larger system, the whole system could be defined as a conjunction of each of the 'smaller' system schemas.

Schemas analogous to *DrawingSystem* (a collection of objects and a function from object identifiers to these objects) occur frequently in this style of specifications. Hall proposes an extension to the Z notation of $\mathbb{S}Figure$ as a shorthand for *DrawingSystem*.

3.3 Operations on figures

In this Section, we see the method for defining operations in terms of single objects and calculating their effect on the whole system (or on defined sets of objects).

A general operation on a figure does not change the *self* component.

Op

Δ *Figure*

$self' = self$

Moving a figure is given by

Move

Op
$move? : VECTOR$

$position' = position + move?$
$edge' = edge$

However, this describes only a single object. The following approach is adopted in order to describe the effect on the whole drawing system of translating one figure.

MoveDS

$\Delta\ DrawingSystem$
$id? : ID$
$move? : VECTOR$

$idFigure' = idFigure \oplus \{id? \mapsto (\mu\ Move_1 \mid$
$\quad \theta\ Figure_1 = idFigure\ id?$
$\quad \wedge\ move_1? = move?$
$\quad \bullet\ \theta\ Figure_1')\}$

This replaces the selected figure with the suitably translated figure. The definition relies on *Move* being deterministic. Hall describes how to specify the effect on the entire system of non-deterministic operations, by using a relational extension of functional override $\oplus_{rel}$.

3.4 Adding a new figure

When a new figure is added to the drawing system, its features must be set.

InitFigure

$Figure$
$edges? : Edges$
$position? : VECTOR$
$kind? : EDGEKIND$

$\theta\ Edges = edges?$
$position = position?$
$kind = kind?$

When the new figure object is added to the system, it must have an identifier that is different from any existing ones. This ensures uniqueness of new objects when they are created.

AddFigure

$InitFigure$
$\Delta\ DrawingSystem$

$self \notin \text{dom}\ idFigure$
$idFigure' = idFigure \cup \{self \mapsto \theta\ Figure\}$

This technique of calculating the effect on the system can be extended to operations on *sets* of objects.

3.5 Summary of Hall's style

This approach, being a *style* for using unchanged Z, has no new language constructs. Hence, it has the sound theoretical basis of Z. It is capable of being understood by those who already know Z.

The style consists of *conventions* for encouraging an object-oriented style of specification. These include

- ☐ the operations defined represent *all* the possible operations; this is required since there is no notation in Z to gather up the operations into a class definition.
- ☐ a *self* property is included in each state description.
- ☐ unchanging object identity is explicitly incorporated into the Δ schema, *Op*.
- ☐ uniqueness of object identities is guaranteed by the approach to describing the system of objects; a function is set up between identifiers and the objects themselves.

The style provides techniques for calculating the effect of an operation on a set of objects; promotion does not work if you wish to apply an operation to more than one object. Reference 1 shows how to perform this calculation for both deterministic and non-deterministic operations.

4 Schuman and Pitt's variant

Schuman and Pitt's notational variant of Z is described in References 2 and 13. Semantic issues are discussed in References 28 and 29.

This notation is described as object-oriented. In fact, it is more concerned with fundamental issues of composition of schemas and reasoning about the resulting composition than with specifying object-oriented systems, or specifying systems in an object-oriented way. However, its principle of specifying no more than is required (for example, it is not necessary to say that everything else stays the same for most operation schemas) makes it particularly useful for specifying systems that consist of several small states, operated on by local operations, and then combined together to form the total system state.

4.1 Overview of Schuman and Pitt's notation

A state schema has three parts: state component declarations, the state invariant predicate and the initialisation condition predicate. If a predicate part is omitted, it defaults to *true*.

State

state component declarations

---

state invariant predicate

==========

initialisation condition predicate

Corresponding operation (or event) schemas also have three components: input and output parameter declarations, the precondition predicate and the postcondition predicate.

State.Op(params)

parameter declarations

---

precondition predicate

===

postcondition predicate

By convention, the name of an operation schema is the state schema name followed by the operation name. The corresponding state component declarations and state invariant predicate are implicitly included in operation schema.

4.2 Quadrilaterals

A quadrilateral is defined by a sequence of four non-zero edge vectors and a position vector.

Quadrilateral

$edge : \text{seq } VECTOR$
$position : VECTOR$

$\# edge = 4$
$edge1 + edge2 + edge3 + edge4 = 0$
$\mathbf{0} \notin \text{ran } edge$

We can move and shear a quadrilateral, supplying the relevant input parameters.

Quadrilateral.Move(move)

$move : VECTOR$

===

$position' = position + move$

Note that, in the postcondition, we do not have to say that everything else stays the same. This is supplied by the semantics in terms of *historical inference*.

Quadrilateral.Shear(s)

$s : SHEAR$

===

definition omitted

4.3 Parallelograms

We can inherit from the general *Quadrilateral* to obtain a *Parallelogram* by adding the relevant extra constraint (we could also add extra state variables at this point).

Parallelogram

Quadrilateral

$edge1 + edge3 = \mathbf{0}$

Operations must be inherited explicitly. All conditions, pre- and post-, may be strengthened by conjoining new constraints. If the precondition is strengthened, it implies that we could not necessarily use a *child* wherever we could use its *parent*. However, note that, if we *can* use it, it behaves in a way that is consistent with the behaviour of its *parent*.

Parallelogram.Move

Quadrilateral.Move

Parallelogram.Shear

Quadrilateral.Shear

Parallelogram has a new operation, enquiring about the angle between adjacent sides (the arrow shows that *a* is an output parameter).

Parallelogram.Angle$(\rightarrow a)$

$a : ANGLE$

===

$a = \angle(edge1, edge2)$

4.4 Rhombus

We can further inherit from *Parallelogram*, and add another constraint, to obtain a *Rhombus*.

Rhombus

Parallelogram

$|edge1| = |edge2|$

The move and angle operations are inherited without change.

Rhombus.Move

Parallelogram.Move

Rhombus.Angle$(\rightarrow a)$

Parallelogram.Angle$(\rightarrow a)$

It is not appropriate to shear a rhombus (the state invariant cannot be met), and so no such operation is defined.

4.5 Rectangle

Alternatively, we can inherit from *Parallelogram*, and add a different constraint, to obtain a *Rectangle*.

Rectangle

Parallelogram

$edge1 \perp edge2$

Instead of inheriting the form of the parallelogram angle operation, we can *redefine* it.

Rectangle.Angle($\rightarrow a$)

a : *ANGLE*

$a = rightAngle$

In the same way as for *Rhombus*, *Rectangle* inherits *Move* from *Parallelogram* without change, and since it is not appropriate to shear a rectangle, no such operation is defined.

4.6 Square

We can multiply inherit from two parents to get a square.

Square

Rhombus
Rectangle

We could inherit the move operation from either parent. Arbitrarily choosing the rhombus gives

Square.Move

Rhombus.Move

The angle operation is more appropriately inherited from the rectangle.

Square.Angle($\rightarrow a$)

Rectangle.Angle($\rightarrow a$)

4.7 Promotion

We define a drawing system of quadrilaterals to include a special one representing the selected quadrilateral. It is on this selected quadrilateral that all the promoted operations will act.

DrawingSystem

figures : $\mathbb{P}$ *Quadrilateral*
q : *Quadrilateral*

$q \in figures$

$figures = \varnothing$

We can select a chosen quadrilateral from a set and apply an operation to this one element, promoting the operation up to work on the larger state.

DrawingSystem.Select(s)

s : *Quadrilateral*

$s \in figures$

$q = s$

DrawingSystem.Move(*move*)

move : *VECTOR*
q.*Move*(*move*)

4.8 Summary of Schuman and Pitt's variant

Schuman and Pitt's variant has better support for object orientation than plain Z. Its main features are

- support for concurrency (not covered in this paper).
- a schema-naming convention for operations, *state.operation*, which binds operations to the relevant state. (However, note that there is no direct syntactic support for classes, for grouping together the operation and state definitions.) The convention also helps in promotion, where operations can be applied more naturally to only part of the state.
- an 'everything else stays the same' semantics for operations, different from the plain Z requirements to state what happens to every variable (otherwise *anything* may happen). When a number of operation schemas are combined to create a multiply inherited operation, we often want to say everything else stays the same *except those parts that must change in order to uphold the state invariant*. We cannot say this in Z, but we can say it quite naturally in this approach.

With this approach, each 'inherited' operation has to be respecified explicitly, even if it is unchanged.

5 Object-Z

Object-Z extends Z by introducing a class construct, which encapsulates state and operation schemas. Classes, and hence state and operations, can be inherited by other classes. The language is introduced in Reference 3 and given a semantics in terms of possible histories (sequences of operations) in Reference 14. Various example specifications using Object-Z have appeared: Reference 30 specifies a simple card game; Reference 31 specifies a cache coherence protocol for a shared memory multiprocessor; and Reference 32 specifies a simplied mobile telephone system.

The notation used in this Section is that defined in Reference 33, which describes Version 1 of the language.

5.1 Overview of Object-Z notation

In Object-Z, a class is defined by

ClassName

[inherited classes]
[local constants]
[state schema]
[initial state schema]
[operation schemas]

[history invariant]

The [inherited classes] are the names of superclasses to be inherited; a subclass incorporates all the features of its

superclasses, including their constants, state and operations. Operations and variables may be renamed in this list.

The [local constants] cannot be changed by any operations, but different instances (objects) can have different values of the constants. The unnamed [state schema] declares state variables and a state invariant that constrains the constants and variables. Together these give the class attributes.

The [initial state schema] is defined in a way similar to plain Z; unprimed variable names are used.

The [operation schemas] define the operations in a way very similar to plain Z, defining a relation between the before and after state. The declaration includes a Δ list, which lists those variables that may be changed by the operation; the other variables remain unchanged. The declaration and predicate of any inherited operation with the same name are implicitly conjoined with this definition.

Class attributes can be objects, and as such, it is often necessary to apply their operations to them. *obj.Op* is an object of the same class as *obj* resulting from performing the operation *Op* on *obj*.

The [history invariant] enables constraints to be included in the allowable order of operations, using notation from temporal logic. It is not discussed further here; neither are the operators used to compose objects in parallel.

5.2 Figures

A general quadrilateral is specified by its edges and its position. It can be moved and sheared. The class *Quadrilateral* is defined by

Quadrilateral

> *Edges*
> *position* : *VECTOR*
> ———
> $0 \notin \operatorname{ran} edge$

> *Move*
> $\Delta(position)$
> *move?* : *VECTOR*
> ———
> $position' = position + move?$

> *Shear*
> $\Delta(edge, position)$
> *s?* : *SHEAR*
> ———
> [(definition omitted)]

The class *Parallelogram* inherits *Quadrilateral*, and hence the operations *Move* and *Shear*. It has an extra constraint on its edges. The operation *Angle*, which outputs the interior angle between two sides, is new.

Parallelogram
Quadrilateral

> $edge1 + edge3 = \mathbf{0}$

> *Angle*
> *a!* : *ANGLE*
> ———
> $a! = \angle(edge1, edge2)$

The class *Rhombus* inherits from *Parallelogram*, with a stronger constraint on its edges. The *Shear* operation is no longer applicable.

Rhombus
Parallelogram

> $|edge1| = |edge2|$

> *Shear*
> *s?* : *SHEAR*
> ———
> *false*

Similarly, the class *Rectangle* inherits from *Parallelogram*, with a stronger constraint on its edges (that they are perpendicular), and with the *Shear* operation no longer applicable.

The class *Square* inherits from *Rhombus* and *Rectangle*. There is no extra constraint on its edges.

Square
Rhombus
Rectangle

5.3 A drawing system

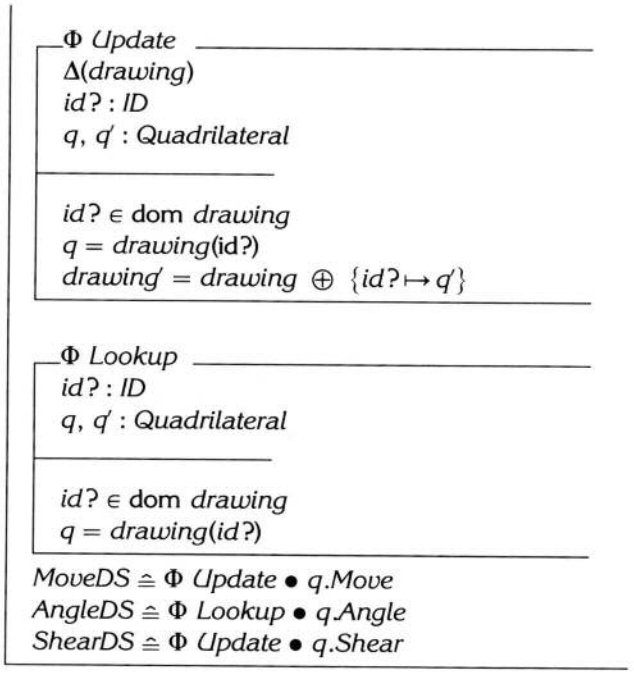
The state of the drawing system consists of a mapping from quadrilateral identifiers to quadrilaterals.

DrawingSystem

> $drawing : ID \rightarrowtail\!\!\!\!\!\!\!\; \leftarrow Quadrilateral$

> *AddFigure*
> $\Delta(drawing)$
> *q?* : *Quadrilateral*
> *id!* : *ID*
> ———
> $id! \notin \operatorname{dom} drawing$
> $drawing' = drawing \cup \{id! \mapsto q?\}$

> *DeleteFigure*
> $\Delta(drawing)$
> *id?* : *ID*
> ———
> $id? \in \operatorname{dom} drawing$
> $drawing' = \{id?\} \lhd\!\!\!- drawing$



The operation *AddFigure* adds a figure to the *DrawingSystem*. The operation *DeleteFigure* deletes a figure from the *DrawingSystem*. The two promotion schemas Φ *Update* and Φ *Lookup* are used to select a particular figure for updating or for interrogating, and *MoveDS*, *AngleDS* and *ShearDS* define the promoted operations. Although these operations are defined in terms of *Quadrilaterals*, they can also be applied to instances of subclasses. This is why there is no equivalent of the previously necessary *EDGETYPE*; the inheritance mechanism handles all this for us automatically.

5.4 Summary of Object-Z

The main feature of Object-Z are

- ☐ a class definition mechanism that directly binds operations together with the relevant state.
- ☐ an inheritance mechanism, allowing incremental development and reuse of specifications. Inheritance can be non-conformant; it is posible to strengthen the precondition of an operation, or completely redefine it, and so make an operation fail on a child when it would work on a parent.
- ☐ some conventions and new symbols to support defining new operations in terms of instance variables' operations.
- ☐ the delta list, giving an 'everything else stays the same' style, making operation definitions concise.
- ☐ a mechanism for constraining the order of operations, based on temporal logic.
- ☐ a formal semantics for classes based on event histories.

Object-Z extends the graphical component of Z, the boxes, to define its classes. This gives an immediate visual indication of the scope of the definitions, as contrasted with the need to search for keywords such as 'begin' and 'endclass' in textual-based variants. This makes it easier to navigate specifications.

Although not widely used outside its birthplace, Object-Z has been used to specify real systems, mainly in the telecommunications area. It appears to be more than just an extension to Z; the feel is rather an object-oriented specification language that uses the Z notation. Although it does not yet have such a solid mathematical base as Z [6], the work on providing it with a formal semantics is progressing [14].

6 Other approaches

Here we briefly describe various other object-oriented Z approaches. Some of these are described in more detail in Reference 34.

6.1 Z and HOOD

HOOD (Hierarchical Object Oriented Design) is the approved European Space Agency design method for Ada. It is not actually object-oriented, since it has no idea of classes or inheritance; it is object-*based* (as is Ada). The hierarchical design approach consists of decomposing a parent object into several child objects, which act together to provide the functionality of the parent.

In conventional HOOD, the more 'formal' parts of the specification are achieved by using Ada as a program description language. Reference 15 describes a way to use Z to specify the HOOD objects. A parent object is specified abstractly (a WHAT specification), using Z to specify an abstract state and abstract operations. The child objects, identified by the HOOD design process, are specified abstractly (as a WHAT specification, or as a WHAT-WITH specification for objects that use other objects). The parent object is then respecified, more concretely in terms of these child objects (a HOW specification), by defining how its abstract state and operations are built from its children and their operations.

The approach modifies the use of Z in two ways; it uses HOOD constructs to limit the scope of the Z definitions, for example, child operation definitions are not visible outside the parent, and it extends the Z dot notation to refer to operations of particular objects. These two modifications give Z a full object-based capability: local scope and the ability to apply operations to objects. Reference 16 proposes a notation for iterating operation schemas (giving something analogous to a 'while loop'), since such constructs can occur in the internals of object specifications.

HOOD also supports concurrency (in the Ada style); the authors want to extend their Z work to also cover this aspect.

6.2 OOZE

OOZE (Object Oriented Z Environment) is described in Reference 17. The 'Environment' in the name refers to support for animation and a database to hold indexes, dependency relations, module expansions, and so on.

The approach combines an algebraic formal semantics, based on OBJ3, with a Z-like notation and style. The algebraic underpinnings become clearer as more of the OOZE language is described; for example, classes can be parameterised by 'theories', which can require properties of the generic parameters.

Schema boxes are nested to group together the class operation definitions. The schema syntax has been

Table 1

	Z	Hall	Schuman/Pitt	Object-Z
object-based	~	✓	✓	✓
class-based	×	×	~	✓
object-oriented	×	×	×	✓

× = an approach does not enjoy a property.
✓ = an approach does enjoy a property.
~ = an approach enjoys a property to a partial degree.

changed from Z to 'enhance readability'; the pre- and post-conditions are separated, there is an *if* statement, the keyword *self* has been added, and there are separate exception schemas.

This approach has the advantage that it stands on a firm mathematical base, algebras, while using a more friendly syntax, Z.

6.3 *Whysall's approach*

Whysall's approach is described in References 18 and 19. The aim of this work is to provide additional structuring of Z specifications in order to aid the refinement process. This is done by splitting a specification into an algebraic-based export part and a model-based body part. The former is intended for use by a client of the class and the latter for refinement by the developer. Standard Z is used, although some extensions are suggested in order to make the process easier.

6.4 Z^{++}

Reference 20 describes some early ideas for overcoming some of the problems, in particular inheritance, which are encountered when endeavouring to specify object-oriented systems in Z. Z^{++} shares some ideas with Object-Z. From a high-level viewpoint, they use similar techniques of class definitions, but whereas Object-Z has a fairly abstract approach, Z^{++} appears to have been more influenced by object-oriented programming language constructs. This influence is probably due to the roots of Z^{++} in the REDO project, concerned with reverse engineering.

Z^{++} directs some of its attention to three levels of software that are identified in the paper. The claim is made that, by separating out the specification of these levels, each can be changed more or less independently of the others. The paper takes a conventional database problem and shows how some of the techniques of object orientation can be applied.

6.5 *MooZ*

Meira and Cavalcanti's work is described in Reference 21. On the surface, this approach looks similar to Object-Z, but the semantics of MooZ is different. A MooZ class also specifies an abstract data type; the semantics of a MooZ object is that of a record (in Object-Z, the semantics is defined in terms of a sequence of operations). The authors claim that this difference results in more natural and concise specifications, and a simpler treatment of message-passing.

7 Summary and conclusions

We use Wegner's classification scheme [24, 25] to classify the various approaches.

- A language is **object-based** if it supports objects as a language feature. Objects are *first-class* values.
- An object-based language is **class-based** if every object has a class.
- A class-based language is **object-oriented** if class hierarchies may be incrementally defined by an inheritance mechanism.

Z used in the conventional style can be used to specify objects, but it does not 'support objects as a language feature', and objects are not 'first-class values', and so it cannot truly be called even object-based by Wegner's definition. However, by straining the definition, we say it is approximately object-based.

Hall's style, although using unmodified Z language, can be considered object-based, because of the way object identities are used. It does as well as can be done without extending Z, which gives it the advantage of the Z theoretical base.

Schuman and Pitt's variant can be considered approximately class-based, since it binds the operations to the state to some degree; objects of the same type have the same behaviour. However, there is no mechanism for grouping together the state and operations into a true class definition. It cannot be considered object-oriented, since 'inherited' operations have to be explicitly respecified, even when unchanged. Work is progressing on providing the variant with a formal semantics.

Object-Z is fully object-oriented; it has classes and inheritance. It is also the best developed approach; there are many case studies using the language in the literature. Work is well advanced on providing the language with a formal semantics.

In summary, based on the case studies detailed in this paper, the approaches can be classified as in Table 1. Can large Z specifications be better structured and made more understandable by using object-oriented ideas? Hall's object-based style of writing Z has been used successfully on a substantial real project [12]. Fully object-oriented Object-Z specifications are eminently readable. Therefore,

the answer is an unequivocal 'yes'. The clarity and precision of model-based Z specifications and the structuring power of object orientation can be combined beneficially.

8 Acknowledgments

This work was part funded by the ZIP project, part of the IED program, project number IED/4/1/1639. The authors would like to thank Trevor King, David Pitt and Gordon Rose for helpful comments.

9 References

[1] HALL, J.A.: 'Using Z as a specification calculus for object-oriented systems' *in* BJØRNER, D., HOARE, C.A.R., and LANGMAACK, H. (Eds.): 'VDM '90: VDM and Z — formal methods in software development', *Lect. Notes Comp. Sci.*, 1990, **428,** pp. 290–318

[2] SCHUMAN, S.A., and PITT, D.H.: 'Object-oriented subsystem specification' *in* MEERTENS, L. G.L.T. (Ed.): 'Program specification and transformation' (North Holland, 1987), pp. 313–341

[3] CARRINGTON, D.A., DUKE, D., DUKE, R., KING, P., ROSE, G.A., and SMITH, G.: 'Object-Z: an object-oriented extension to Z' *in* VUONG, S. (Ed.): 'Formal description techniques FORTE '89' (North Holland, 1990), pp. 401–420

[4] HAYES, I.J. (Ed.): 'Specification case studies' (Prentice Hall, 1987)

[5] POTTER, B., SINCLAIR, J., and TILL, D.: 'An introduction to formal specification and Z' (Prentice Hall, 1991)

[6] SPIVEY, J.M.: 'Understanding Z: a specification language and its formal semantics' (Cambridge University Press, 1988), Vol. 3

[7] SPIVEY, J.M.: 'The Z notation: a reference manual' (Prentice Hall, 1989)

[8] WOODCOCK, J.C.P.: 'Structuring specifications in Z', *Softw. Eng. J.*, 1989, **4,** (1), pp. 51–66

[9] BARDEN, R., STEPNEY, S., and COOPER, D.: 'The use of Z'. Proc. Sixth Annual Z User Meeting, York, 1991

[10] FLYNN, M., HOVERD, T., and BRAZIER, D.: 'Formaliser — an interactive support tool for Z' *in* NICHOLLS, J.E. (Ed.): 'Z User Workshop, Proc. Fourth Annual Z User Meeting, Oxford, Workshops in Computing' (Springer Verlag, 1990), pp. 128–141

[11] SAMPAIO, A., and MEIRA, S.L.: 'Modular extensions to Z' *in* BJØRNER, D., HOARE, C.A.R., and LANGMAACK, H. (Eds.): 'VDM '90: VDM and Z — formal methods in software development', *Lect. Notes Comp. Sci.*, 1990, **428,** pp. 211–232

[12] BROWNBRIDGE, D.: 'Using Z to develop a CASE toolset' *in* NICHOLLS, J.E. (Ed.): 'Z User Workshop, Proc. Fourth Annual Z User Meeting, Workshops in Computing', (Springer Verlag, 1990), pp. 142–149

[13] SCHUMAN, S.A., PITT, D.H., and BYERS, P.J.: 'Object-oriented process specification' *in* RATTRAY, C. (Ed.): 'Specification and verfication of concurrent systems', Workshops in Computing, Stirling (Springer Verlag, 1990), pp. 21–70

[14] DUKE, D., and DUKE, R.: 'Towards a semantics for Object-Z' *in* BJØRNER, D.J., HOARE, C.A.R., and LANGMAACK, H. (Eds): 'VDM '90: VDM and Z — formal methods in software development', *Lect. Notes Comput. Sci.*, 1990, **428,** pp. 244–261

[15] DI GIOVANNI, R., and IACHINI, P.L.: 'HOOD and Z for the development of complex systems' *in* BJØRNER, D., HOARE, C.A.R., and LANGMAACK, H. (Eds.): 'VDM '90: VDM and Z — formal methods in software development', *Lect. Notes Comp. Sci.*, 1990, **428,** pp. 262–289

[16] IACHINI, P.L.: 'Operation schema iterations' *in* NICHOLLS, J.E. (Ed.): 'Proc. Fifth Annual Z User Meeting, Workshops in Computing', Oxford (Springer Verlag, 1991), pp. 27–49

[17] ALENCAR, A.J., and GOGUEN, J.A.: 'OOZE: an object-oriented Z environment' *in* AMERICA, P. (Ed.): 'ECOOP '91, European Conf. on Object-Oriented Programming', *Lect. Notes Comput. Sci.*, 1991, **512,** pp. 180–199

[18] WHYSALL, P.J., and McDERMID, J.A.: 'An approach to object oriented specification using Z', *in* NICHOLLS, J.E. (Ed.): 'Proc Fifth Annual Z User Meeting, Workshops in Computing' (Springer Verlag, 1991), pp. 193–215

[19] WHYSALL, P.J., and McDERMID, J.A.: 'Object oriented specification and refinement' *in* MORRIS, J.M., and SHAW, R.C. (Eds.): '4th Refinement Workshop, Workshops in Computing', Cambridge (Springer Verlag, 1991), pp. 150–184

[20] LANO, K.C.: 'Z++, an object-orientated extension to Z', *in* NICHOLLS, J.E. (Ed.): 'Proc. Fifth Annual Z User Meeting, Workshops in Computing' (Springer Verlag, 1991), pp. 151–172

[21] MEIRA, S.L., and CAVALCANTI, A.L.C.: 'Modular object oriented Z specifications' *in* NICHOLLS, J.E. (Ed.): 'Proc. Fifth Annual Z User Meeting, Workshops in Computing' (Springer Verlag, 1991), pp. 173–192

[22] CUSACK, E.: 'Inheritance in object oriented Z' *in* AMERICA, P. (Ed.): ECOOP '91, European Conf., on Object-Oriented Programming, *Lect. Notes Comp. Sci.*, 1991, **512,** pp. 167–179

[23] BOOCH, G.: 'Object oriented design with applications' (Benjamin Cummings, 1991)

[24] WEGNER, P.: 'Dimensions of object-based language design'. OOPSLA '87 Proc., (*ACM SIGPLAN Not.*, 1987, **22,** (12), pp. 168–182)

[25] WEGNER, P.: 'The object-oriented classification paradigm' *in* SHRIVER, B., and WEGNER, P. (Eds.): 'Research directions in object-oriented programming' (MIT Press, 1987)

[26] MORGAN, C.C., and SUFRIN, B.A.: 'Specification of the Unix filing system' *in* HAYES, I.J. (Ed.): 'Specification case studies' (Prentice Hall, 1987)

[27] LUPTON, P.J.: 'Promoting forward simulation' *in* NICHOLLS, J.E. (Ed.): 'Proc. Fifth Annual Z User Meeting, Workshops in Computing' (Springer Verlag, 1991), pp. 27–49

[28] PITT, D.H., and BYERS, P.J.: 'The rest stays unchanged'. Technical Report CS-91-07, University of Surrey, Department of Mathematical and Computing Sciences, 1991

[29] SHIELDS, M.W.: 'Let sleeping DatATypeS lie (the rest stays unchanged)'. Technical Report CS-91-08, University of Surrey, Department of Mathematical and Computing Sciences, 1991

[30] DUKE, R., and DUKE, D.: 'Aspects of object-oriented formal specification'. Australian Software Engineering Conf., 1990

[31] SMITH, G., and DUKE, R.: 'Specification and verification of a cache coherence protocol'. Technical Report 126, Department of Computer Science, University of Queensland, August 1989

[32] DUKE, R., ROSE, G.A., and LEE, A.: 'Object-oriented protocol specification', *Protocol Spec. Test. Verif.*, 1990, **10**

[33] DUKE, R., KING, P., ROSE, G.A., and SMITH, G.: 'The Object-Z specification language version 1'. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991

[34] STEPNEY, S., BARDEN, R., and COOPER, D.: 'Comparative study of object orientation in Z'. ZIP document, ZIP/Logica/90/046(3), Logica Cambridge Ltd., December 1991

The paper was first received on 28 February and in revised form on 27 November 1991.

The authors are with Logica Cambridge Ltd., Betjeman House, 104 Hills Road, Cambridge CB2 1LQ.