Object-Z: a Specification Language Advocated for the Description of Standards

Roger Duke *Gordon Rose* *Graeme Smith*

Software Verification Research Centre
Department of Computer Science
University of Queensland, Australia

Abstract

The importance of formalising the specification of standards has been recognised for a number of years. This paper advocates the use of the formal specification language Object-Z in the definition of standards. Object-Z is an extension to the Z language specifically to facilitate specification in an object-oriented style. First, the syntax and semantics of Object-Z are described informally. Then the use of Object-Z in formalising standards is demonstrated by presenting a case study based on the ODP Trader. Finally, a formal semantics is introduced that suggests an approach to the standardisation of Object-Z itself. Because standards are typically large complex systems, the extra structuring afforded by the Object-Z class construct and operation expressions enables the various hierarchical relationships and the communication between objects in a system to be succinct ly specified.

Key words: object-orientation, formal specification, formal semantics, standards.

1 Introduction

The use of formal methods for the precise and unambiguous specification of standards has been strongly advocated for a number of years. This paper advocates the use of the formal specification language Object-Z[17, 18] in the definition of standards. The aim of the paper is to introduce the syntax and semantics of Object-Z informally, to demonstrate its use in formalising standards by presenting a case study based on the ODP Trader[13], and to outline a formal semantics that could form the basis for the standardisation of Object-Z itself.

The Z specification language[1, 21, 16] has already been used to describe standards[14, 9, 19]. Object-Z is an extension to Z specifically to facilitate specification in an object-oriented style. An object-oriented specification describes a system as a collection of interacting objects, each of which has a prescribed structure and behaviour. Such decomposition improves the clarity of large specifications and facilitates the separation of concerns. Object-Z commenced development within a collaboration between the University of Queensland and the Overseas Telecommunications Corporation (OTC) of Australia to investigate the role of formal methods in telecommunications standards. Because standards are typically large and complex, the structuring afforded by the Object-Z class construct and operation expressions enables the various hierarchical relationships and the communication between objects in a system to be specified succinct ly.

Object-Z has been applied to a range of case studies including a mobile phone system[18], communications protocols[6], a buttons console[17] and the denotational semantics of programming languages[5].

Section 2 gives an overview of Object-Z, outlining its syntax and semantics informally. In particular, those features of Object-Z that were not discussed in Version 1[6] are examined, namely, the operators of angelic choice ($[]$) and sequential composition ($\fatsemi$), secondary state variables, class union and object containmen t. In addition, the semantics of Object-Z is now based on reference semantics rather than the value semantics of Version 1.

As an illustration of the use of Object-Z in the description of standards, Section 3 gives a formal specification of a study based on a simplified view of the ODP Trader standard. This case study shows how the class construct together with object instantiation and containment in Object-Z can be used to capture the structural relationships between objects. Furthermore, it shows how Object-Z's operation expressions specify communication between objects.

If a language is advocated as suitable for the description of standards, then clearly that language must itself be standardised. Section 4 introduces a formal axiomatic semantics for Object-Z upon which the standardisation of Object-Z could be based. This semantic approach extends the approach taken in the standardisation of Z[1].

In describing Object-Z, familiarity with Z is assumed. The Appendix outlines aspects of Z pertinent to this paper.

2 Object-Z Overview

2.1 Introduction

A Z specification defines a number of state and operation schemas. Inferring which operations may affect a particular state schema requires examining the signatures of every operation. In contrast, Object-Z associates individual operations with one state schema. The collective definition of a state schema with its associated operations (and other components to be detailed later) constitutes the definition of a *class*.

A class is template for objects: each object of the class has a state which conforms to the class' state schema and is subject to state transitions which conform to the class' operations. A class is also used as a type: instances of that type are identities which reference objects of that class. This enables objects to refer to other objects.

An Object-Z specification of a system comprises a number of class definitions possibly related by inheritance, a mechanism for class adaptation by modification or extension.

Subsection 2.2 specifies a simple stack as a first introduction to Object-Z notation. Subsection 2.3 introduces a dual view of objects: an external view addressing object identity, and an internal view addressing object state and its transitions. Subsection 2.4 illustrates the role of object identity in the specification of aggregates. Subsection 2.5 describes operators for building operation expressions and Subsection 2.6 outlines inheritance in Object-Z. Subsection 2.7 introduces secondary state variables, Subsection 2.8 polymorphism, and Subsection 2.9 object containment. Subsection 2.10 briefly discusses history predicates which further constrain specifications by imposing conditions on the temporal order of operations to support notions such as *fairness*.

2.2 The class Stack

Object-Z is introduced by progressive examples, the first being a stack. A formal specification of a stack of items of type *Item* in Object-Z notation is:

$[Item]$ introduces *Item* as a given type

$$
\begin{array}{|l}
\textit{Stack} \\
\hline
\begin{array}{|l}
max : \mathbb{N} \\
\hline
max \leqslant 100
\end{array} \\
\\
\begin{array}{|l|}
\hline
items : \operatorname{seq} Item \\
\hline
\#items \leqslant max \\
\hline
\end{array} \\
\\
\begin{array}{|l|}
\textsc{Init} \\
\hline
items = \langle\, \rangle \\
\hline
\end{array} \\
\\
\begin{array}{|l|}
\textit{Push} \\
\hline
\Delta(items) \\
item? : Item \\
\hline
\#items < max \\
items' = \langle item? \rangle \frown items \\
\hline
\end{array} \\
\\
\begin{array}{|l|}
\textit{Pop} \\
\hline
\Delta(items) \\
item! : Item \\
\hline
items \neq \langle\, \rangle \\
items = \langle item! \rangle \frown items' \\
\hline
\end{array} \\
\hline
\end{array}
$$

It is necessary to define *Item* global to *Stack* as communication with stack objects involves input/output parameters of type *Item*. *Stack* has a constant *max* which is at most 100. Distinct *Stack* objects may have different values of *max*, each remaining constant. The state schema has one state variable *items* denoting a sequence of elements of type *Item*. The state predicate requires that the size of the sequence is at most *max*. An initialised stack has no items.

The operations of *Stack*, namely *Push* and *Pop*, are interpreted in the context of the given type *Item* which is global, the constant *max* and the pre- and post-state schemas (i.e. the state in unprimed and primed form respectively).

Operation *Push* prepends a given input $item?$ of type *Item* to the existing sequence of items, provided (as stated by its first conjunct) the stack has not already reached its maximum size. The understanding of $\Delta(items)$ is that the state variable *items* is subject to change – the second conjunct of *Push* specifies the change. If that conjunct were omitted, $items'$ need only conform to its type and the *class invariant* (the conjunction of the predicates of the state and that of the constant definition). In general, state variables not included in the Δ-list of an operation are implicitly unchanged by the operation; however as will be seen, when operations are combined, their Δ-lists are united so that only variables not in the union are unchanged. Omitting a Δ-list is equivalent to an empty Δ-list.

Operation *Pop* outputs a value $item!$ which is the head item of the sequence *items* and reduces *items* to the tail of its pre-value.

The *attributes* of class *Stack* are its constants and state variables (i.e. $max, items$), and its *features* are its attributes, INIT and its operations (i.e. *max, items,* INIT, *Push, Pop*).

Classes may be generic: thus, instead of confining the stack to have items of specific type *Item*, *Stack* could be specified as generic by including a formal generic type parameter, say T. The definition would be headed $Stack[T]$ and all occurrences of *Item* would be replaced by T.

When a class is used, formal generic types if included are instantiated by actual types and features and operation parameters may be renamed. For example, the following class descriptor characterises stacks of natural numbers with state variables renamed to *nats* and with input/output parameters renamed to $nat?$ and $nat!$ respectively:

$$Stack[\mathbb{N}][nats/items, nat?/item?, nat!/item!]$$

Renaming is simultaneous and the scope of renaming is the whole class.

2.3 The dual view of objects

Objects have an *external view* which is a persistent *identity*, and an *internal view* which is characterised by constants, state, initialisation and operations. History predicates may also be included in the internal view as discussed later.

For the external view, if C is a class, the declaration $c : C$ declares c to be a variable whose value is a reference to (i.e. the identity of) an object of class C. Distinct references identify distinct objects. Thus in this context, C denotes the set of references to potential objects of class C. There is no implication that an object reference declaration introduces a distinct reference and therefore a distinct object, nor does the declaration imply that the introduced object is initialised. Thus, declaration $c, d : C$ does not imply that c and d reference different objects. To ensure distinction, the conjunct $c \neq d$ would be included in the state predicate. If c and d are to be distinct initially, but subsequently are to refer synonymously to the same object, then $c \neq d$ would be a conjunct of INIT and an operation would, for example, include $c' = d$.

For the internal view, the semantics of an Object-Z class comprises a set of attribute bindings consistent with their types and the class invariant, a set of initial bindings, and, for the operations, a set of relations on attribute bindings with input/output parameters and possibly auxiliary variables.

The term $c.att$ denotes the value of attribute att of the object referenced by c, and $c.\mathit{INIT}$ is a predicate which denotes whether or not the object c conforms to C's initial state schema. The term $c.Op$ denotes an operation which transforms the object referenced by c according to the definition of C's Op: operations provide the only mechanism for transforming the internal view of objects.

2.4 Modelling aggregates

The advantage of object declarations introducing references to objects is that object identity becomes a first-class entity; thus, the object referenced by c may evolve (internally) without changing the value of c. This is particularly significant in modelling aggregates. For example, the declaration $sc : \mathbb{P}\, C$ ($\mathbb{P}$ denotes *power-set-of*) models an aggregate of objects of class C which may evolve internally without changing the value of sc, because the object references do not change. Evolution of a constituent object is specified by defining an operation such as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Select \quad}} \\
c? : sc \\
\hline
\end{array}$$

and using it as in:

$$Op \mathrel{\widehat=} Select \bullet c?.Op$$

This construct effectively 'promotes' the operation Op of the nominated object ($c?$) to be an operation of the class with attribute sc. In effect, $Select$ is a selection environment. The notation $operation_1 \bullet operation_2$ denotes environment enrichment in that the schema text of $operation_1$ enriches the environment in which $operation_2$ is interpreted.

Several objects of an aggregate may cooperate using a multiple selection environment such as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad SelectTwo \quad}} \\
c_1?, c_2? : sc \\
\hline
c_1? \neq c_2? \\
\hline
\end{array}$$

and using it as in:

$$Together \mathrel{\widehat=} SelectTwo \bullet (c_1?.Op_1 \land c_2?.Op_2)$$

An object reference is only Δ-listed if the reference is to change value and refer to another object. An object reference set, such as sc above, is only Δ-listed if the set is subject to change, i.e. if references are added, removed or substituted. For further discussion on object identity in Object-Z see [7].

(In Z, the modelling of aggregates and selection from aggregates typically requires the use of 'framing' techniques which are complex compared to those used above.)

2.5 Composite Operations

This section introduces Object-Z's operators for combining operations. Consider the class $StackPair$ which has two individually named references to stacks of natural numbers.

$$NatStack == Stack[\mathbb{N}][nats/items, nat?/item?, nat!/item!]$$

$$
\begin{array}{|l}
\hline
\textit{StackPair} \\
\quad \begin{array}{|l}
\hline
s_1, s_2 : \textit{NatStack} \\
\hline
s_1 \neq s_2 \\
\# s_1.\textit{nats} \leqslant \# s_2.\textit{nats} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textsc{Init} \\
\hline
s_1.\textsc{Init} \land s_2.\textsc{Init} \\
\hline
\end{array} \\
\quad \textit{Push}_1 \mathrel{\widehat=} s_1.\textit{Push} \\
\quad \textit{Push}_2 \mathrel{\widehat=} s_2.\textit{Push} \\
\quad \textit{Pop}_1 \mathrel{\widehat=} s_1.\textit{Pop} \\
\quad \textit{Pop}_2 \mathrel{\widehat=} s_2.\textit{Pop} \\
\quad \textit{PushBoth} \mathrel{\widehat=} \textit{Push}_1 \land \textit{Push}_2 \\
\quad \textit{Transfer} \mathrel{\widehat=} (\textit{Pop}_1 \parallel \textit{Push}_2) \setminus (\textit{nat}!) \\
\quad \textit{PushOne} \mathrel{\widehat=} \textit{Push}_1 \mathbin{[\,]} \textit{Push}_2 \\
\quad \textit{TransferAll} \mathrel{\widehat=} \textit{Transfer} \mathbin{;} \textit{TransferAll} \\
\qquad\qquad\qquad\quad [\,] \\
\qquad\qquad\qquad\quad [\, s_1.\textit{items} = \langle\,\rangle \,] \\
\hline
\end{array}
$$

It is intended that the stacks s_1 and s_2 be distinct, hence the references are explicitly distinguished. The class invariant also requires that the size of the first stack does not exceed that of the second. It is also intended that initially each stack be initialised, hence the explicit initialisation provision.

Operation $\textit{Push}_1$ promotes s_1's *Push* operation to be an operation of *StackPair*. Similarly, $\textit{Push}_2$, $\textit{Pop}_1$ and $\textit{Pop}_2$ are promotions.

As s_1 and s_2 do not appear in any Δ-list, they continue to refer to their respective objects.

Operation *PushBoth* illustrates operation conjunction '$\land$': in this example, the individual push operations proceed independently except that the same value $\textit{nat}?$ applies to both stacks. The conjunction operator conjoins constraints and equates variables with the same name (e.g. in *PushBoth*, $\textit{nat}?$ of $\textit{Push}_1$ and of $\textit{Push}_2$ are equated).

Operation *Transfer* illustrates the parallel operator '$\parallel$'. It behaves like conjunction but also equates inputs and outputs with the same basename. Thus conceptually, the item popped from s_1 ($\textit{nat}!$) is communicated to and pushed onto s_2 ($\textit{nat}? = \textit{nat}!$). Like conjunction, the parallel operator is commutative and therefore supports communication in either direction (or both directions if the operations interchange several items of information). Inputs which receive communication ($nat?$ above) are hidden, but to make the parallel operator associative, outputs are not hidden. Outputs which are not intended for the environment, such as *Transfer*'s $nat!$, must be explicitly hidden. As with conjunction, outputs with the same basename are equated. Residual inputs (those not paired with matching outputs and therefore not hidden) with the same name are also equated.

An alternative definition of *Transfer* is:

$$Transfer \mathrel{\widehat{=}} s_1.Pop \mathbin{;} Push_2.$$

This definition illustrates the sequential operator '$\mathbin{;}$'. It behaves like forward relational composition of the relations corresponding to the operation's operands. The operator is non-commutative, as would be expected, but it is also non-associative because of the following communication mechanism. Communication is left-to-right and hidden, i.e. outputs of the left operand equate to inputs of the right operand with the same basename and both are hidden. Thus, in the sequential definition of *Transfer*, $Push_2$'s $nat?$ is equated to $nat!$ of $s_1.Pop$ and $nat?$ and $nat!$ are hidden. Residual inputs with the same name are equated and residual outputs (those not engaged in communication within the expression) with the same name are equated. Sequential chains are interpreted left-associatively. The semantics of sequential composition requires the notion of intermediate states.

Operation *PushOne* illustrates the choice operator '$[]$'. The operator indicates non-deterministic choice of one operand (operation) from those operands with satisfied preconditions. Thus, the choice is deterministic if exactly one precondition is satisfied and the construct fails if no precondition is satisfied. As an enabled operand is chosen in preference to any disabled operand, the choice is angelic. Depending on the states of s_1 and s_2, there are three possible outcomes of *PushOne*: the operation fails (both are full or s_2 is full and s_1 is already the same size as s_2), $nat?$ is pushed onto s_1, or $nat?$ is pushed onto s_2. There is no notion of communication between operands of the choice operator because at most one operand is selected. The operator is commutative and associative.

Operation *TransferAll* illustrates the sequential composition operator in a recursive definition. The recursion is a relational composition chain of transfer operations terminated by the identity relation. The construction of the chain is deterministic as exactly one precondition of $Transfer$ or $Empty_1$ is true. The overall effect of $TransferAll$ is to place all of s_1's items in reverse sequence onto s_2.

All operations in Object-Z are relations or combinations of relations (e.g. intersection and sequential composition) which are also relations; thus operations are atomic, including recursively-defined operations such as $TransferAll$.

The precedences of the conjunction and parallel operators are equal. Sequential composition has lower precedence, and choice is lower than sequence.

Distributed versions of conjunction, parallel, sequence and choice are also available.

For example, the operation $Collate$ below transfers in sequence the head item from each of ten stacks $stacks(1), \cdots stacks(10)$ to a single receiving stack s.

$$Collate \mathrel{\widehat{=}} \fatsemi\ (i : 1 \mathrel{..} 10 \bullet stacks(i).Pop \fatsemi s.Push)$$

The distribution construct extends maximally to the right, similarly to universal and existential quantification. $Collate$ expands to:

$$(stacks(1).Pop \fatsemi s.Push) \fatsemi (stacks(2).Pop \fatsemi s.Push) \fatsemi \cdots$$

The order of the terms of this distribution is clearly important: in this case the required order is $1, 2, \cdots 10$, as implied by the ordering of the natural numbers in the quantifier set.

Renaming of input, output or auxiliary variables of operations within operation expressions is available. For example, in an application in which the selection of an item of a set arises internally rather than by external selection, an existing operation definition $Select \mathrel{\widehat{=}} [\, c? : C \mid c? \in sc \,]$ could be rephrased to $Select\, [\, c/c? \,] \bullet c.Op$. Writing $c!$ instead of c would output $c!$ to the environment.

2.6 Inheritance

Inheritance is a mechanism for incremental specification, whereby new classes may be derived from one or more existing classes.

Essentially, all definitions are pooled with the following provisions. Inherited type and constant definitions and those declared in the derived class are merged. The state and initialisation schemas of inherited classes and those declared in the derived class are conjoined. Operations with the same name are conjoined. Name clashes, which would lead to unintentional merging or conjunction, can be resolved by renaming when inheriting.

As a simple example of inheritance, consider the derivation of a generic queue from $Stack[T]$.

$Queue[T]$

$\upharpoonright(max, \textsc{Init}, Join, Leave)$

$Stack[T][Leave/Pop]$

$Join$

$\Delta(items)$
$item? : T$

$\#items < max$
$items' = items \frown \langle item? \rangle$

The projection symbol '$\upharpoonright$' indicates visibility, i.e. introduces the list of those features which are accessible via the dot notation. If the list is omitted, as in previous examples, all features are visible.

Operation *Pop* is renamed more appropriately to *Leave*, and *Join* which appends *item*? is introduced. Operation *Push* is inherited, but as it is not a queue operation it is not listed as visible. Visibility is not inherited, i.e. a derived class may nominate any feature of the inherited class as visible. Thus, should $Queue[T]$ be inherited, operation *Push* could become a visible feature of the derived class.

The derivation of $Queue[T]$ from $Stack[T]$ illustrates that not all properties of inherited classes are preserved. For example, a stack has the *first-in-last-out* property whereas a queue is characterised by *first-in-first-out*. Derived properties such as *first-in-first-out* are generally sensitive to the addition of new operations.

Multiple inheritance is semantically equivalent to a chain of single inheritance derivations. For an example of multiple inheritance in Object-Z see [17].

Operation redefinition is effected by defining an operation of the same name in the derived class. Conjunction with the definition in the inherited class is implicit. Should conjunction not be required, the inherited operation is renamed when inherited leaving the definition in the derived class to be absolute. The new definition may use the inherited operation under the new name.

2.7 Secondary Variables

The state variables described so far are *primary*. They can only change via those operations whose Δ-lists explicitly include them. It is often convenient to introduce additional state variables to improve the readability of a specification. For example, consider the addition of *size* to *Stack*'s state schema.

$$
\begin{array}{|l}
items : \mathrm{seq}\, Item \\
size : \mathbb{N} \\
\hline
size = \# items \\
size \leqslant max
\end{array}
$$

Availability of *size* would allow rephrased conjuncts such as $size = 0$, $size \leqslant max$ and $size > 0$. Although the improvement in readability is debatable in this example, in general it is an advantage to be able to replace an expression by an appropriately named variable.

State variable *size* adds no further information to the specification in that it is derivable from *items*; moreover, because *size* changes with *Push* and *Pop*, it needs to be listed in their Δ-lists. To preserve the convenience of dependent variables without the need to explicitly include them in Δ-lists which contain any variable they depend on, the notion of *secondary* variable is introduced.

Secondary variables ultimately depend on primary variables and are implicitly included in every Δ-list. Syntactically, secondary variables are introduced by a Δ declarator suggestive of implicit inclusion in every Δ-list.

Returning to class *Stack*, to introduce *size* as a secondary variable, the state schema would be revised to:

$$
\begin{array}{|l}
items : \mathrm{seq}\, Item \\
\Delta \\
size : \mathbb{N} \\
\hline
size = \# items \\
size \leqslant max
\end{array}
$$

Secondary variable *size* is now implicitly in the Δ-lists of *Push* and *Pop*.

The post-form conjunct $size' = \#items'$ holds in every operation: this preserves the dependency of *size* on *items* after every operation.

A derived class may add a new variable which depends on inherited variables. Without the facility to declare the new variable as secondary, all inherited operations which change any variable on which the new variable depends would need to be redefined to enable the new variable to change.

2.8 Polymorphism

There are two mechanisms for polymorphism in Object-Z. The first is conventional in that it is based on the inheritance hierarchy. Declaration $c : \downarrow C$ introduces c as a reference to an object of class C or any derivative of C. The internal view ascribed to c depends on the particular class of c.

The second basis for polymorphism is the *class union* construct [3], which explicitly lists the collection of classes to be considered as polymorphic. Following is a brief explanation of the concept which is detailed in [3]. Features common to all classes so grouped constitute a *polymorphic core*. Thus, provided applications use only core features, specifications can accommodate references to objects of any of the classes nominated in the definition of the class union. An advantage of class union over inheritance-based polymorphism is that the concerns of derivation and polymorphism can be detached. Given an inheritance hierarchy, it is generally difficult to modify it without major redesign; in contrast, class union can group classes independently of their hierarchical status.

The advantages of polymorphism are well known; however, polymorphism is a relaxation of strong typing and its dynamic implications increase semantic complexity.

2.9 Containment

Consider the situation where a campus contains a set of buildings, each building contains a set of rooms and each room contains a set of terminals. A specification in Object-Z would be

Terminal

[details omitted]

Room

$ts : \mathbb{P}\, Terminal$

[operations omitted]

Building

$rs : \mathbb{P}\, Room$

$$\begin{array}{l} \forall\, r_1, r_2 : rs \bullet \\ \quad r_1 \neq r_2 \Rightarrow \\ \qquad r_1.ts \cap r_2.ts = \varnothing \end{array}$$

[operations omitted]

Campus

$bs : \mathbb{P}\, Building$

$$\begin{array}{l} \forall\, b_1, b_2 : bs \bullet \\ \quad b_1 \neq b_2 \Rightarrow \\ \qquad b_1.rs \cap b_2.rs = \varnothing \\ \qquad \forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet \\ \qquad\qquad r_1.ts \cap r_2.ts = \varnothing \end{array}$$

[operations omitted]

The class invariant of class *Building* specifies that no terminal can be in two distinct rooms in a building. Similarly, the class invariant of class *Campus* specifies that no room can be in two distinct buildings of the campus. Furthermore, despite the fact that the predicate of class *Building* states that no terminal can be in two distinct rooms, as this applies only to the rooms of a given building it says nothing about rooms in distinct buildings. Hence the conjunct

$$\forall\, r_1 : b_1.rs;\ r_2 : b_2.rs \bullet r_1.ts \cap r_2.ts = \varnothing$$

must be included in the of the predicate of the *Campus* class.

Clearly, capturing the properties of object containment explicitly in this way is cumbersome, particular if the system is large and complex. We would like to be able to give a global invariant that captures directly the condition that distinct rooms anywhere contain distinct terminals, and distinct buildings anywhere contain distinct rooms. The condition that distinct rooms contain distinct terminals, for example, is not an internal invariant of the *Room* class, but rather an invariant of any system containing room objects; nevertheless, it would be convenient to be able to attach such glob al conditions directly to the *Room* class.

If the role of an attribute is to always identify directly contained objects, this can be indicated when the attribute is declared by appending a subscript ‘$_{\copyright}$’ to the appropriate type, removing the necessity to write explicit predicates to capture containment. For example, adopting this syntactic convention, the relevant classes in the specification of the terminal location system become

Room
$$ts : \mathbb{P}\ Terminal_{\copyright}$$

Building
$$rs : \mathbb{P}\ Room_{\copyright}$$

Campus
$$bs : \mathbb{P}\ Building_{\copyright}$$

The subscript '$_{\copyright}$' is appended to the type of the attribute rather then the attribute itself because the attribute may identify a complex data structure rather than an object reference. For example, the declaration

$$ts : \mathbb{P}\ Terminal_{\copyright}$$

in class *Room* declares *ts* to be a set, not an object reference. ¿From its type, *ts* is a set of references to objects of class *Terminal*; the $_{\copyright}$ implies these references are to contained objects.

In general, two properties of object containment are implied when the $_{\copyright}$ abbreviation is used:

- no object directly or indirectly contains itself; and
- no object is directly contained in two distinct objects.

For a detailed treatment of object containment see [4].

2.10 History Invariants

A *history invariant* is an optional predicate over histories of objects of the class expressed in temporal logic. Such predicates further constrain possible behaviour. For example, suppose we wish to specify in the generic stack that whenever *Pop* is continuously enabled it must eventually occur.

FairStack[*T*]

$$Stack[T]$$
$$\Box(items \neq \langle\rangle \Rightarrow \Diamond((\bigcirc \# items) < \# items))$$

In this example, the history invariant is expressed using the temporal logic operators $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next). For an introduction to temporal logic and a semantic description of these operators see Moszkowski[15].

Informally, the history invariant for *FairStack* determines that for any history of a *FairStack* object, if at time t_1 the stack is not empty ($items \neq \langle\rangle$) then, at some future time t_2 the size of the stack will be decremented (i.e. immediately following t_2 the size will be less than at t_2 as captured by $(\bigcirc \# items) < \# items$), i.e. *Pop* must occur, as this is the only operation that decrements the size of the stack.

3 Specifying Trader

This section illustrates the application of Object-Z to the description of standards by formally specifying some aspects of the ODP Trader[13]. The Object-Z class construct, operation operators and object containment features are particularly useful to formally capture the structural relationships that exist between Trader objects. The Z language has been used previously to specify the functionality of Trader[12]; however, without object-oriented constructs it is not easy to use Z to capture the above structural relationships.

Trader is a large and complex standard and it would be inappropriate to attempt a complete formal specification in this paper. The structural view presented here is a significant simplification of Trader and ignores detail irrelevant to that view.

We start with an informal description of the simplified Trader system.

A trader consists of a collection of exporters (suppliers) each offering some service, and a collection of importers (consumers) each wishing to use the services supplied by the exporters. The role of the trader is to facilitate contact between an importer and an exporter who can supply the requested service. Collections of traders are federated so that if the exporters within the importer's trader cannot supply the requested service, the search is widened to the other traders in the federation.

To begin a formal specification of the simplified Trader system, let

$$[Service]$$

denote the set of all possible services that could be offered by an exporter or requested by an importer. A particular service would in general be a number of values giving, for example, the cost, location, etc. of the service.

An exporter is modelled as an object that offers a particular service, as specified by the class *Exporter*.

Exporter

$service : Service$

ChangeService

$\Delta(service)$

$service? : Service$

$service' = service?$

The operation *ChangeService* enables the details of the service offered by an exporter to be modified from time to time as circumstances change.

For simplicity, the formal specification developed here does not address visibility; in particular, the omission of visibility lists does not convey the usual meaning that all features are visible.

Exporters are aggregated, as specified by the class *Exporters*.

Exporters

$exporters : \mathbb{P}\, Exporter_{\copyright}$

INIT

$exporters = \varnothing$

Add

$\Delta(exporters)$

$ex? : Exporter$

$ex? \notin exporters$

$exporters' = exporters \cup \{ex?\}$

Remove

$\Delta(exporters)$

$ex? : Exporter$

$ex? \in exporters$

$exporters' = exporters \setminus \{ex?\}$

Search

$services? : \mathbb{P}\, Service$

$exs! : \mathbb{P}\, Exporter$

$exs! = \{e : exporters \mid e.service \in services?\}$

$ChangeService \mathrel{\widehat{=}} [ex? : exporters] \bullet ex?.ChangeService$

In this model it is required that an exporter can belong to only one aggregate, e.g. if $exps_1, exps_2 : Exporters$ are distinct objects (i.e. $exps_1 \neq exps_2$) then

$$exps_1.exporters \cap exps_2.exporters = \varnothing.$$

This condition is captured in the specification using the notion of object containment by appending $_{\copyright}$ to the type $Exporter$ in the declaration of $exporters$.

Initially the aggregate has no exporters, but a new exporter can be added (operation Add) and an existing exporter removed (operation $Remove$).

The operation $Search$ models the searching of the aggregate to find those exporters (output as $exs!$) whose service is in the given set of services (input as $services?$). The input $services?$ of this operation would, in practice, stipulate services covering a range of costs, locations, etc. The output $exs!$ is the set of exporters in the aggregate whose service is in the range of services requested.

Operation $ChangeService$ is the promotion of the nominated exporter's $ChangeService$ operation to the aggregate level.

An importer is modelled as an object of the class $Importer$.

$Importer$

  $Request$

  $services! : \mathbb{P}\, Service$

  $exs? : \mathbb{P}\, Exporter$

This class has no attributes (and hence no internal state) and has only one operation, $Request$, which models the importer requesting a set of services ($services!$) and receiving back a set of exporters ($exs?$). As we shall see, the set of exporters returned to the importer consists of those exporters that fulfil the requested service. In practice, an importer would then apply some preference criteria so as to select from the set $exs?$ (if it is not empty) one exporter to supply the service requested. This selection aspect of the standard is omitted in the model formalised here.

Like exporters, importers are aggregated, as specified by the class $Importers$.

$$
\begin{array}{l}
\textbf{Importers} \\
\quad importers : \mathbb{P}\, Importer_{\copyright} \\[4pt]
\quad \textsc{Init} \\
\quad importers = \varnothing \\[4pt]
\quad Add \\
\quad \Delta(importers) \\
\quad im? : Importer \\
\quad \hline \\
\quad im? \notin importers \\
\quad importers' = importers \cup \{im?\} \\[4pt]
\quad Remove \\
\quad \Delta(importers) \\
\quad im? : Importer \\
\quad \hline \\
\quad im? \in importers \\
\quad importers' = importers \setminus \{im?\} \\[4pt]
\quad Request \mathrel{\widehat=} [im? : importers] \bullet im?.Request
\end{array}
$$

As for aggregates of exporters, we suppose that an importer belongs to only one aggregate and capture this condition using the notion of object containment. Initially the aggregate contains no importers, but a new importer can be added and an existing importer removed.

The operation *Request* indicates that any importer in the aggregate can be nominated to perform its request operation.

A trader is now modelled as an exporter aggregate and an importer aggregate, as specified by the class *Trader*.

$$
\begin{array}{|l}
\hline
\textit{Trader} \\
\quad \begin{array}{|l}
\hline
exps : Exporters_{\copyright} \\
imps : Importers_{\copyright} \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textsc{Init} \\
exps.\textsc{Init} \\
imps.\textsc{Init} \\
\hline
\end{array} \\
\quad AddExporter \mathrel{\widehat=} exps.Add \\
\quad AddImporter \mathrel{\widehat=} imps.Add \\
\quad RemoveExporter \mathrel{\widehat=} exps.Remove \\
\quad RemoveImporter \mathrel{\widehat=} imps.Remove \\
\quad ExporterChangeService \mathrel{\widehat=} exps.ChangeService \\
\quad SearchExporters \mathrel{\widehat=} exps.Search \\
\quad ImporterRequest \mathrel{\widehat=} imps.Request \\
\quad Contact \mathrel{\widehat=} (ImporterRequest \parallel SearchExporters) \setminus (services!, exs!) \\
\hline
\end{array}
$$

In our model an aggregate of exporters or importers can belong to only one trader, and again this uniqueness requirement is captured by the notion of object containment.

Initially, each aggregate is in its initial state (i.e. empty) but a new exporter can be added ($AddExporter$), a new importer can be added ($AddImporter$), an existing exporter can be removed ($RemoveExporter$), an existing importer can be removed ($RemoveImporter$) and an exporter in the aggregate can have its service modified ($ExporterChangeService$). The aggregate of exporters can be searched for specific services ($SearchExporters$) and a nominated importer can request a range of services ($ImporterRequest$).

Each of these operations is a promotion of an operation defined on either the $exps$ of the $imps$ aggregates.

The final operation, $Contact$, models an importer $im?$ requesting a set of services $services!$ (i.e. $ImporterRequest$) in parallel with $SearchExporters$. According to the semantics of parallel composition, $services!$ is equated to $services?$ of $SearchExporters$, and $services?$ is hidden. $SearchExporters$ also communicates with $ImporterRequest$: this equates their $exs!$ and $exs?$ respectively, and hides $exs?$. As there is no communication required with the environment, the outputs *services*! and *exs*! are explicitly hidden. As will be seen, the notion of 'contact' is generalised when traders are federated.

The federated trader system is now modelled by the class *FederatedTraders*: it consists of an aggregate of traders.

$$
\begin{array}{l}
\textit{FederatedTraders} \\
\hline
\quad \textit{traders} : \mathbb{P}\, \textit{Trader}_{\copyright} \\
\quad \textit{INIT} \\
\quad \forall\, t : \textit{traders} \bullet t.\textit{INIT} \\
\quad \textit{AddExporter} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{AddExporter} \\
\quad \textit{AddImporter} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{AddImporter} \\
\quad \textit{RemoveExporter} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{RemoveExporter} \\
\quad \textit{RemoveImporter} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{RemoveImporter} \\
\quad \textit{ExporterChangeService} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{ExporterChangeService} \\
\quad \textit{TraderRequest} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{ImporterRequest} \\
\quad \textit{TraderSuccessSearch} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{SearchExporters} \bullet \left[\, \textit{exs}! \neq \varnothing \,\right] \\
\quad \textit{TraderFailSearch} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \textit{tr}?.\textit{SearchExporters} \bullet \left[\, \textit{exs}! = \varnothing \,\right] \\
\quad \textit{Tally} \\
\quad\quad \textit{exs}? : \mathbb{P}\, \textit{Exporter} \\
\quad\quad \textit{addexs}? : \mathbb{P}\, \textit{Exporter} \\
\quad\quad \textit{exs}! : \mathbb{P}\, \textit{Exporter} \\
\quad\quad \hline
\quad\quad \textit{exs}! = \textit{exs}? \cup \textit{addexs}? \\
\quad \textit{OtherTradersSearch} \mathrel{\widehat{=}} [\textit{tr}? : \textit{traders}] \bullet \\
\quad\quad\quad \mathbin{\S}\, \textit{tr} : \textit{traders} \setminus \{\textit{tr}?\} \bullet (\textit{tr}.\textit{SearchExporters}[\textit{addexs}!/\textit{exs}!] \mathbin{\S} \textit{Tally}) \\
\quad \textit{Search} \mathrel{\widehat{=}} \textit{TraderSuccessSearch} \\
\quad\quad\quad\quad [\!] \\
\quad\quad\quad\quad \textit{TraderFailSearch} \mathbin{\S} \textit{OtherTradersSearch} \\
\quad \textit{Contact} \mathrel{\widehat{=}} (\textit{TraderRequest} \parallel \textit{Search}) \setminus (\textit{services}!, \textit{exs}!) \\
\hline
\end{array}
$$

A trader belongs to only one federation, and again this uniqueness condition is captured by the notion of object containment. Initially each trader in the federation is in its initial state. Any trader in the federation can be nominated to have an exporter or an importer added or removed, or to have the services of any of its exporters changed.

Operation *TraderRequest* promotes an importer request of the nominated trader to the federated traders level.

So far, the federated trader operations are promotions of operations of a nominated trader.

Operation *TraderSuccessSearch* models a successful search within trader $tr?$, i.e. the set of suitable exporters found is not empty. This operation extends tr's *SearchExporters* operation to include the additional conjunct $exs! \neq \varnothing$.

The operation *TraderFailSearch* models an unsuccessful search within trader $tr?$: the operation is specified similarly to *TraderSuccessSearch* except that the added conjunct is $exs! = \varnothing$.

The key part of the operation *OtherTradersSearch*, namely

$$⨾\ tr : traders \setminus \{tr?\} \bullet (tr.SearchExporters[addexs!/exs!] ⨾ Tally)$$

is a distributed composition over the set of other traders ($traders \setminus \{tr?\}$). The composition searches in turn (in some unspecified order) each other trader tr in the federation with respect to the requested $services?$. The set of exporters found by searching trader tr is renamed $addexs!$ and added (using the *Tally* operation) to the accumulated set of exporters found so far ($exs?$) from traders already searched. The result ($exs!$) is passed to the next search-and-tally stage of the composition and equated to its $exs?$. The $exs!$, $exs?$ communication pair is hidden in accordance with the semantics of sequential composition. The overall effect of the distributed composition is to commence with set $exs?$ from the environment (it will be seen that this will be empty) and add to it the accumulated result of searching each trader other than $tr?$. Thus, the final output $exs!$ is the set containing all exporters (of traders other than $tr?$) that fulfil the search request $services?$. The variables for communication with the environment are $tr?, exs?, services?$ and $exs!$ ($exs?$ being an input to the first stage and $exs!$ being the output from the last stage).

Operation *Search* of the federated trader models the deterministic choice between a nominated trader search and a search of all other traders. If the nominated trader search succeeds, that choice is the outcome and the search is complete; otherwise all of the other traders are searched. The empty set $exs!$ of *TraderFailSearch* and $exs?$

of *OtherTradersSearch* are equated, thereby providing the foreshadowed empty set of exporters to initialise the accumulation of suitable exporters. This *exs*!, *exs*? pair is hidden by sequential composition as are all other such pairs of the sequential chain. Regardless of the outcome of the choice construct, the inputs from the environment are *tr*? and *services*? and the output to the environment is *exs*!.

The final operation, *Contact*, models an importer *im*? of a trader *tr*? requesting a set of services *services*! (i.e. *TraderRequest*) in parallel with *Search*. According to the semantics of parallel composition, *services*! is equated to *services*? of *Search*, and *services*? is hidden. There is also communication from *Search* to *TraderRequest*, which equates their *exs*! and *exs*? respectively, and hides *exs*?. As there is no communication required with the environment other than *tr*?, the outputs *services*! and *exs*! are explicitly hidden.

4 Towards Formal Semantics

Formal notations such as Object-Z can only provide precise and unambiguous descriptions of standards if they themselves have a precise and unambiguous syntax and semantics. For this reason, several specification languages used in standards have been standardised[10, 11, 2] and others, including Z[1], are currently undergoing standardisation.

Informal descriptions of the semantics of a specification language are prone to be as imprecise and ambiguous as informal descriptions of standards. Therefore, a language standard should provide a formal semantics. Such a semantics is expressed in some well-established mathematical domain such as algebra or set theory.

A formal semantics also provides a basis for reasoning about specifications. This enables the specifier to gain confidence in both the correctness of the specification (its well-formedness and consistency) and the behaviour of the specified system (by showing certain properties are logical consequences of the specification). It is also essential in the process of refinement of a specification towards an implementation.

The Z Base Standard[1] provides a sound basis from which a standard for Object-Z could be developed. It includes abstract and concrete syntaxes of Z as well as a *denotational* semantics and a logic for reasoning. The logic, by describing the properties associated with constructs of the language, provides an abstract, *axiomatic* semantics of Z.

In general, both a denotational and axiomatic semantics are required. While an axiomatic semantics is more suitable for reasoning than a denotational semantics, a denotational semantics is required to prove that an axiomatic semantics is *sound*, i.e. that it cannot prove contradictory properties. In this section, we outline the denotational and axiomatic semantics of the Z Base Standard and show how they can be extended for Object-Z. For a more detailed description of the Z semantics the reader is referred to [1].

4.1 Denotational Semantics

4.1.1 Z

The denotational semantics of Z in the Z Base Standard has its basis in classical ZF set theory. The semantic universe in which the meaning of a Z specification is given contains the meanings of *names*, *types* and *values* used in the specification together with an *environment* describing the overall meaning of the specification.

- Names used in a specification are partitioned into schema names, variable names and constant names. This partitioning is only made in the semantics and is not part of the concrete syntax of Z.

  $$\langle SchemaName, Variable, Constant \rangle \text{ partitions } Name$$

- Each name is associated with a type. Types are partitioned into given types, power set types, cartesian product types and schema types.

  $$\langle G\_type, P\_type, Cp\_type, S\_type \rangle \text{ partitions } Type$$

  For each specification there is a set of distinct given types. They are identified by their names taken from the set of constant names.

  $$givenT : Constant \rightarrowtail\!\!\!\!\twoheadrightarrow G\_type$$

  All other types are constructed from the given types using the type constructors defined below.

  $$\begin{array}{l} powerT : Type \rightarrowtail\!\!\!\!\twoheadrightarrow P\_type \\ cproductT : \mathrm{seq}\ Type \rightarrowtail\!\!\!\!\twoheadrightarrow Cp\_type \\ schemaT : (Name \twoheadrightarrow\!\!\!\!\!\mapsto Type) \rightarrowtail\!\!\!\!\twoheadrightarrow S\_type \end{array}$$

The type of a schema depends only on the mapping from the names of its components to their types. Any restrictions imposed by its predicate are considered when giving a meaning to the schema.

- Each type has an associated set of values called its *carrier set*. An *element* of a Z specification, such as a set or tuple, is represented by a pair consisting of its type and its value.
- The environment of a specification is defined as a finite partial function from names to elements.

Given this semantic universe, the meanings of expressions, predicates, declarations and schemas can be given (see [1] for details). These meanings are combined to provide the meaning of a *paragraph* (i.e. a type definition, axiomatic definition, schema or global predicate) as a relation between environments, i.e. between the environment it occurs in and that formed by enriching this environment by the paragraph. The meaning of a specification is found by applying the composition of its paragraph relations to the empty environment.

4.1.2 Object-Z

An extension of the semantic universe of the Z Base Standard to accommodate Object-Z is presented in [8]. An additional set of names, to accomodate the names of classes, and two additional kinds of types, namely object reference types and class instance types are introduced. (Names are also extended with a "role" the details of which are omitted here.)

$$\langle SchemaName, Variable, Constant, ClassName \rangle \text{ partitions } Name$$
$$\langle G\_type, P\_type, Cp\_type, S\_type, O\_type, Cl\_type \rangle \text{ partitions } Type$$

The distinction between the two additional types is central to the reference semantics of Object-Z. An object reference denotes a value which identifies an object in the specified system. A class instance, on the other hand, denotes a collection of values corresponding to the object's attributes. These values must satisfy the class invariant of the object's class.

Object reference types are similar to given types in that they are not constructed from other types. The constructor function for object reference types is

$$objectT : ClassName \rightarrowtail\!\!\!\!\!\twoheadrightarrow O\_type$$

The inverse of this function returns the class of the object identified by a particular object reference.

The constructor function for class instance types is defined as

$$classT : ClassDecl \rightarrowtail\!\!\!\!\twoheadrightarrow Cl\_type$$

where *ClassDecl* denotes the declarations occurring in a class and is defined as follows.

$$
\begin{array}{|l}
\textit{ClassDecl} \\
\hline
visible : \mathbb{F}\, Name \\
state : Name \mathrel{+\!\!\!\!\twoheadrightarrow} Type \\
opsigs : Name \mathrel{+\!\!\!\!\twoheadrightarrow} (Name \mathrel{+\!\!\!\!\twoheadrightarrow} Type) \\
attributes : \mathbb{F}\, Name \\
operations : \mathbb{F}\, Name \\
\hline
attributes = \operatorname{dom} state \\
operations = \operatorname{dom} opsigs \\
visible \subseteq attributes \cup operations \\
\hline
\end{array}
$$

4.2 Axiomatic Semantics

4.2.1 Z

The logic proposed in the Z Base Standard is a Gentzen-style sequent calculus. Axioms and theorems are expressed using *sequents* and further theorems are derived by the application of *inference rules*. Side-conditions on the inference rules are given using *meta-functions* which return information derived from the specification text.

- Sequents are of the form

  $$d \mid \Psi \vdash \Phi$$

  The *antecedent* of the sequent consists of a list of declarations d and a set of predicates Ψ and the *consequent* is a set of predicates Φ. A sequent is *valid* if at least one of the predicates in Ψ is true in the environment of the specification enriched by d and satisfying all predicates in Φ. Antecedents and consequents can be empty: an empty antecedent is equivalent to the empty declaration and the single predicate *true*, and an empty consequent is equivalent to the single predicate *false*.

For example, given a Z specification which includes the schema

$$\begin{array}{|l} \multicolumn{1}{l}{S} \\ \hline x, y : \mathbb{N} \\ \hline x < y \\ \hline \end{array}$$

the following sequents are valid.

$$S \vdash x \in \mathbb{N}$$
$$S \vdash y \in \mathbb{N}$$
$$S \vdash x < y$$

In these sequents S abbreviates $x, y : \mathbb{N} \mid x < y$ corresponding to $d \mid \Psi$.

- Inference rules for manipulating sequents are of the form

$$\frac{\textit{premisses}}{\textit{conclusion}} \quad [\ \textit{name}\ ]\ (\ \textit{proviso}\ )$$

The *premisses* are zero or more sequents and the *conclusion* is a single sequent. The *name* simply identifies the rule and the *proviso* is a predicate side-condition which must be true in the environment of the specification for the rule to be applicable. An inference rule is *sound* if whenever its proviso is satisfied and it is applied to valid premisses, a valid conclusion results.

As an example, consider the following rule for schema disjunction. (S_1 and S_2 are schemas. The notation $[S]$ denotes the predicate of the schema S.)

$$\frac{\begin{array}{l}[S_1] \vdash \\ [S_2] \vdash\end{array}}{[S_1 \lor S_2] \vdash} \quad [\ \textit{SchemaDisjunction} \vdash\ ]$$

Since the consequent of each sequent in the rule is empty (and, hence, equivalent to the single predicate *false*), they are valid only if the predicates of their antecedents evaluate to *false* in the environment of the specification. The rule states, therefore, that if we can deduce *false* from both the predicate of schema S_1 and the predicate of schema S_2 then we can deduce false from the predicate of the schema disjunction $S_1 \lor S_2$.

A "rule-lifting" meta-theorem allows a common declaration to be added to every antecedent in a rule, or a common predicate to be added either to every antecedent or to every consequent. Therefore, given that the above rule is sound so is the following rule which is derived by adding predicate p to every consequent.

$$\frac{\begin{array}{l}[S_1] \vdash p \\ [S_2] \vdash p\end{array}}{[S_1 \lor S_2] \vdash p}$$

Therefore, if we can deduce any predicate p from the predicates of S_1 and S_2 then we can deduce p from the predicate of $S_1 \lor S_2$.

Generally, such a rule would require a proviso that any global variables captured by one schema are also captured by the other (e.g. see Chapter 5 of [20]). However, such provisos are avoided by syntactically distinguishing between global variables and those declared in schemas.

The rules of the logic in the Z Base Standard have been proved sound with respect to its denotational semantics. The proof of soundness is dependent on everything being well-typed.

- The meta-function α returns the names of declared variables in a schema expression or declaration. The meta-function ϕ returns the names of free variables in a schema expression, declaration, predicate or expression. For example, consider the following inference rule for set comprehension. (The notation $\uparrow\downarrow$ indicates that the rule can be used in either direction, i.e. the premiss can also be derived from the conclusion. d is a declaration, p is a predicate and t and u are expressions.)

  $$\frac{\vdash \exists\, d \mid p \bullet t = u}{\vdash t \in \{d \mid p \bullet u\}} \uparrow\downarrow \quad [\ \mathit{SetComprehension}\ ]\ (\ \phi t \cap \alpha d = \varnothing\ )$$

  The proviso ensures variables occurring in t are not captured when it is in the scope of d.

4.2.2 Object-Z

An Object-Z specification consists not only of a global environment but also several local environments; one for each specified class. To reason about properties within these local environments, we extend the notation of the logic in the Z Base Standard so that sequents can be interpreted within a particular class context as follows. (A is a class.)

$$A :: \quad d \mid \Psi \vdash \Phi$$

The validity of the sequent is determined in the environment of the specification enriched with that of the class A. For example, given the following specification

```
┌─ A ──────────────────────────────
│ | c : ℕ
│
│ ┌────────────────────────────
│ │ x : ℕ
│ ├──────────
│ │ x < c
│ └────────────────────────────
│
│ ┌─ Inc_x ─────────────────────
│ │ Δ(x)
│ ├──────────
│ │ x' = x + 1
│ └────────────────────────────
└──────────────────────────────────
```

the following sequents are valid. (The keyword $\textit{State}$ is introduced in the logic to refer to the state schema of a class.)

$$\begin{array}{ll}
A :: & \vdash c \in \mathbb{N} \\
A :: & \textit{State} \vdash x \in \mathbb{N} \\
A :: & \textit{State} \vdash x < c \\
A :: & \textit{Inc_x} \vdash x' = x + 1
\end{array}$$

In the final sequent, $\textit{Inc\_x}$ denotes the schema part of the operation $\textit{Inc\_x}$ (i.e. without its delta-list). In general, an operation appearing as an expression, predicate or declaration in the logic, is interpreted as its schema part only. The schema part of an operation is also denoted explicitly by the notation $[\, OP \,]$ (i.e. the schema which includes (the schema of) OP as a declaration). Therefore, $[\, OP_1 \,] \wedge [\, OP_2 \,]$ is a schema expression whereas $OP_1 \wedge OP_2$ is an operation expression.

The notation for representing provisos is also extended so that they can be interpreted within a class context. For example, the proviso $A ::\ p$ states that p must be true in the environment of the specification extended with that of A.

For each meta-theorem of the logic in the Z Base Standard (e.g. "rule-lifting"), we require a similar meta-theorem to hold within a class context. Furthermore, so that the inference rules of the Z Base Standard can be used in the context of a class, the following meta-theorem is introduced. (d_1 and d_2 are declarations and Ψ_1, Ψ_2, Φ_1 and Φ_2 are sets of predicates.)

If the rule $$\frac{d_1 \mid \Psi_1 \vdash \Phi_1}{d_2 \mid \Psi_2 \vdash \Phi_2} \quad (\ p\ )$$ is sound,

then the rule $$\frac{A :: \quad d_1 \mid \Psi_1 \vdash \Phi_1}{A :: \quad d_2 \mid \Psi_2 \vdash \Phi_2} \quad (\ A :: \ p\ )$$ is also sound.

This meta-theorem also allows us to state rules involving a single class without explicitly declaring a class context. For example, the following rules for operation operators could be used within the context of a class.

Conjunction

The schema of the operation $OP_1 \wedge OP_2$ is the conjunction of the schemas of OP_1 and OP_2.

$$\frac{OP_1 \wedge OP_2 = [\, OP_1 \,] \wedge [\, OP_2 \,] \vdash}{\vdash} \quad [\ OperationConjunction\ ]$$

This rule is meant to be used in combination with "rule-lifting" and simply allows the predicate $OP_1 \wedge OP_2 = [\, OP_1 \,] \wedge [\, OP_2 \,]$ to be used in a proof at any time. It is equivalent to the rule

$$\frac{}{\vdash OP_1 \wedge OP_2 = [\, OP_1 \,] \wedge [\, OP_2 \,]}$$

but is more convenient to use in practice.

As the delta-list of an operation is also required in the side-conditions of some rules, e.g. those involving the application of operations to an object, we introduce a meta-function δ which returns an operation's delta-list. For example,

$$\delta\,[\, \Delta(x_1, \ldots, x_n)\ d \mid p \,] = \{x_1, \ldots, x_n\}$$
$$\delta\,[\, d \mid p \,] = \varnothing$$
$$\delta(a.Op) = \varnothing$$

Note that the delta-list of $a.Op$ is empty because this operation does not change any of the variables of the class in which it occurs. Although the object referenced by a is changed, the reference a itself is not.

The delta-list of the conjunction of two operations is the union of their individual delta-lists.

$$\delta(OP_1 \wedge OP_2) = \delta(OP_1) \cup \delta(OP_2)$$

Parallel

To define the parallel operator, the meta-functions $\beta_?$ and $\beta_!$ are introduced to return the basenames (i.e. apart from the ? or !) of the inputs and outputs of an operation respectively.

The schema of the operation $OP_1 \parallel OP_2$ is equal to the schema formed by

- noting any input variables of the schema of each operation which have the same basename as an output variable of the schema of the other operation,
- renaming each such input variable to the name of the output variable,
- and conjoining the resulting schemas.

$$\frac{\begin{array}{l} OP_1 \parallel OP_2 = [\,OP_1\,][y_1!/y_1?, \ldots, y_m!/y_m?] \\ \qquad\qquad\qquad \wedge \\ \qquad\qquad\qquad [\,OP_2\,][x_1!/x_1?, \ldots, x_n!/x_n?] \vdash \end{array}}{\vdash} \quad [\,\mathit{Parallel}\,]\ (\ p\ )$$

$$\text{where } p \equiv \begin{array}{l} \beta_!(OP_1) \cap \beta_?(OP_2) = \{x_1, \ldots, x_n\} \\ \beta_?(OP_1) \cap \beta_!(OP_2) = \{y_1, \ldots, y_m\} \end{array}$$

The delta-list of a parallel composition of two operations is the union of their individual delta-lists.

$$\delta(OP_1 \parallel OP_2) = \delta(OP_1) \cup \delta(OP_2)$$

Choice

The schema of the operation $OP_1 \;[]\; OP_2$ is equal to the schema formed by

- adding to the schema of OP_1 the predicate which states that all variables in OP_2's delta-list which are not in OP_1's delta-list are unchanged,
- adding to the schema of OP_2 the predicate which states that all variables in OP_1's delta-list which are not in OP_2's delta-list are unchanged,

- and disjoining the resulting schemas.

$$\frac{\begin{array}{l} OP_1 \mathbin{[\!]} OP_2 = \left[\, OP_1 \mid x_1' = x_1 \land \ldots \land x_n' = x_n \,\right] \\ \qquad\qquad\quad \lor \\ \qquad\qquad\quad \left[\, OP_2 \mid y_1' = y_1 \land \ldots \land y_m' = y_m \,\right] \vdash \end{array}}{\vdash} \quad [\, \mathit{Choice} \,]\ (\, p \,)$$

$$\text{where } p \equiv \delta(OP_2) \setminus \delta(OP_1) = \{x_1, \ldots, x_n\}$$
$$\delta(OP_1) \setminus \delta(OP_2) = \{y_1, \ldots, y_m\}$$

The delta-list of the resulting operation is the union of the individual delta-lists.

$$\delta(OP_1 \mathbin{[\!]} OP_2) = \delta(OP_1) \cup \delta(OP_2)$$

Enrichment

If in the environment extended with the schema of OP_1 we can show that the schema of OP_2 is equal to some schema of the form $[d \mid p]$, where the free variables of d do not include any variables declared in OP_1, then the schema of $OP_1 \bullet OP_2$ is equal to $[OP_1;\ d \mid p]$.

$$\frac{OP_1 \vdash OP_2 = \left[\, d \mid p \,\right]}{\vdash OP_1 \bullet OP_2 = \left[\, OP_1;\ d \mid p \,\right]} \quad [\, \mathit{Enrichment} \,]\ (\, \phi d \cap \alpha OP_1 = \varnothing \,)$$

The delta-list of the resulting operation is the union of the individual delta-lists.

$$\delta(OP_1 \bullet OP_2) = \delta(OP_1) \cup \delta(OP_2)$$

The meta-functions α and ϕ of the Z Base Standard are extended for Object-Z. The meta-function α is extended to return the names of the types, constants and state variables of a class. The meta-function ϕ is extended to treat references to the state schema, initial state schema and operations of a class as variables rather than schema references. This is necessary to define inference rules where we want to use definitions from one class in the context of another. In such cases, the definitions must not include references to class schemas as these will, in general, have a different meaning in the other class context. For example, the inherited state of a class (denoted $\uparrow\!\mathit{S\scriptstyle TATE}$) is the conjunction of each of the state schemas of the inherited classes. ($A_1, \ldots, A_n$ and B are classes and $S_1, \ldots, S_n$ are schemas. ι is a meta-function which returns the set of classes inherited by a class.)

$$\frac{\begin{array}{l} A_1 :: \quad \vdash \mathit{S\scriptstyle TATE} = S_1 \\ \ldots \\ A_n :: \quad \vdash \mathit{S\scriptstyle TATE} = S_n \end{array}}{B :: \quad \vdash \uparrow\mathit{S\scriptstyle TATE} = S_1 \wedge \ldots \wedge S_n} \quad [\ \mathit{InheritedState}\ ]\ (\ p\ )$$

$$\begin{array}{ll} \text{where } p \equiv & \iota(B) = \{A_1, \ldots, A_n\} \\ & \phi(S_1) \subseteq \alpha(A_1) \\ & \ldots \\ & \phi(S_n) \subseteq \alpha(A_n) \end{array}$$

The provisos of the form $\phi(S_i) \subseteq \alpha(A_i)$ ensure that the schemas S_i are expressed in fully-expanded form (i.e. in terms of $\alpha(A_i)$ only and not in terms of references to the state, initial state or operations of A_i.)

5 Conclusions

The paper advocates the use of formality in the specification of standards; moreover, given the complexity of modern standards, structured specifications are essential. The object-oriented structuring facilities and the formality of the specification language Object-Z make it a worthy candidate for the presentation of standards.

The paper presented Object-Z, an extension of the formal specification language Z to include a class construct which encapsulates a number of related definitions in order to define objects. Systems (which include standards) are then specified in terms of their constituent objects. Classes are reused in the definition of other classes via inheritance or in the declaration of references to other objects. The declaration of object references, in contrast to object states, simplifies the definition of object aggregates which appear often in real systems. The specification of object interaction and communication is facilitated by a range of operators for composing operation expressions. Object-Z's expressive power is further enhanced by notions of dependent variables, class union (a novel form of polymorphism), object containment and temporal logic history predicates.

To support the advocacy of using Object-Z in standards, a simplified view of the ODP Trader was specified; in particular, the example demonstrated the expressive power of Object-Z's object containment and operation expressions.

Before having confidence in using any notation for the description of standards, a formal semantics of the notation must be developed. Since Object-Z is an extension of Z, the formal semantics of Z provides a basis on which Object-Z's semantics can be built. The paper outlined extensions to both a denotational semantics and an axiomatic semantics of Z towards a semantics for Object-Z.

Acknowledgements

The authors wish to thank Steven Atkinson, Jin Song Dong, Alena Griffiths, Wendy Johnston, Andreas Prinz and Udaya Shankar for contributing to the ideas expressed in this paper. The research is supported by the Autralian Research Council.

References

[1] S.M. Brien and J.E. Nicholls. Z Base Standard: Version 1. Technical Report PRG-107, Oxford University Computing Laboratory, 1992.

[2] CCITT. *CCITT Specification and description language SDL.* Geneva, 1992. Recommendation Z.100 (SDL'92).

[3] J. Dong and R. Duke. Class Union and Polymorphism. In C. Mingins, W. Haebich, J. Potter, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 12 & 9)*, pages 181–190. Prentice-Hall International, 1993.

[4] J. Dong and R. Duke. The Geometry of Object Containment. Technical Report 94-17, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994. To appear in Object-Oriented Systems (OOS).

[5] J. Dong, R. Duke, and G. Rose. An Object-Oriented Approach to the Semantics of Programming Languag es. In G. Gupta, editor, *Proc. 17th Annual Computer Science Conference (ACSC'17)*, pages 767–775, 1994.

[6] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1991.

[7] R. Duke and G. Rose. Modelling object identity. In *Proceedings 16th Australian Computer Science Conference (ACSC-16)*, pages 93–100, 1993.

[8] A. Griffiths and G. Rose. A semantic foundation for object identity in formal specification. Technical Report 94-21, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1994.

[9] I. Hayes, M. Mowbray, and G. Rose. Signalling system No.7: The network layer. In E. Brinksma, G. Scollo, and C. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 3–14. North-Holland, 1990.

[10] ISO TC97/SC21. *Estelle – A Formal Description Technique Based on an Extended State Transition Model*, 1988. Intl. Standard 9074.

[11] ISO TC97/SC21. *LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988. Intl. Standard 8807.

[12] ISO/IEC JTC1/SC21/WG7 N743. *ANNEX G Z specification of the Trader*, November 1992.

[13] ISO/IEC JTC1/SC21/WG7 N743. *Working Document on Topic 9.1 - ODP Trader*, November 1992.

[14] P. King. A formal specification of signalling system Number 7 link layer. In *Proc. 4th Australian Software Eng. Conf. (ASWEC'89)*, pages 113–118, 1989.

[15] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1986.

[16] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Series in Computer Science. Prentice-Hall International, 1990.

[17] G. Rose. Object-Z. In S. Stepney, R. Barden, and D. Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[18] G. Rose and R. Duke. An Object-Z specification of a mobile phone system. In K. Lano and H. Haughton, editors, *Object-Oriented Specification Case Studies*, pages 110–129. Prentice-Hall International, 1993.

[19] G. Smith. A formal specification of signalling system No. 7 telephone user part. In *Proc. 1989 Singapore Intl. Conf. on Networks (SICON'89)*, pages 50–55, July 1989.

[20] J.M. Spivey. *Understanding Z: A specification language and its formal semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[21] J.M. Spivey. *The Z Notation: A Reference Manual (2nd Ed.)*. Series in Computer Science. Prentice-Hall International, 1992.

6 Appendix: An Overview of Z

This appendix overviews aspects of Z pertinent to the paper. For a complete glossary of the Z notation with respect to logic, sets, relations, etc, and a summary of Z schema conventions, consult [1, 21].

6.1 Structure of a Z Specification

Z specifications comprise precise mathematical (formal) text with informal explanatory text and diagrams. The informal material facilitates the interpretation of the formal text and relates it to the real world. The advantages of formality are that it eliminates ambiguities inherent in informal description and provides a rigorous basis for reasoning.

6.2 Data Types

Z data types are based on sets. Z allows the introduction of arbitrary *given* types using notation such as $[Person, Key, Account]$ which introduces three given types for representing the set of persons, the set of keys and the set of accounts relevant to a particular specification. The set of integers is the standard given type '$\mathbb{Z}$' and may be assumed in any specification. The usual integer arithmetic and comparison operators are available. The set of natural numbers $(0, 1, \cdots)$, denoted '$\mathbb{N}$', is also a standard given type.

Types in Z are built on given types using constructors for power sets '$\mathbb{P}$', Cartesian products '$\times$' and schemas (elaborated below).

Z also has a range of symbols for denoting frequently used constructed types, e.g. the relation symbol '$\leftrightarrow$' applied as in $r : A \leftrightarrow B$, where A and B are arbitrary types, abbreviates the set-of-pairs declaration $r : \mathbb{P}(A \times B)$. Symbols are also available for special kinds of relation, such as partial functions '$\pfun$', finite partial functions '$\ffun$', total functions '$\rightarrow$', partial injections '$\pinj$', bijections '$\bij$', etc. Symbols for denoting relational inversion, restriction, composition, etc. are also available.

A sequence in Z is formalised as a function with a domain which is the contiguous set of natural numbers from 1 to the length of the sequence. The set of all sequences with elements from generic set X is:

$$\operatorname{seq} X == \{s : \mathbb{N}_1 \ffun X \mid \exists\, n : \mathbb{N} \bullet \operatorname{dom} s = 1 \mathrel{..} n\}$$

where 'seq' denotes *sequence of*, '$==$' denotes *syntactic equivalence*, '$\mathbb{N}_1$' denotes the natural numbers excluding zero, '$|$' and '$\bullet$' read *such that*, and 'dom' abbreviates *domain of*. Thus, if $x_1, x_2, \cdots \in X$, then $\{(1, x_1), (2, x_2), \cdots\} \in \operatorname{seq} X$. Z has a more convenient notation for sequences, this sequence being denoted by $\langle x_1, x_2, \cdots \rangle$. The elements of a sequence need not be distinct. The empty sequence is denoted by $\langle\,\rangle$. If $s : \operatorname{seq} X$ is not empty, functions '*head*' and '*tail*' are defined using the equalities:

$$head\ s = s(1) \quad \text{and} \quad s = \langle head\ s \rangle \frown tail\ s$$

where '$\frown$' denotes sequence concatenation.

As a sequence is a set and also a function, conventional set and function notation applies to sequences. Z supports traditional set notions, such as membership '$\in$', intersection '$\cap$', union '$\cup$', subset '$\subseteq$', cardinality '$\#$', etc. Thus for example, if $s_1, s_2 : \operatorname{seq} X$ then $s_1 \subseteq s_2$ is a valid predicate and means that s_1 is a prefix of s_2, e.g. $\langle x_1, x_2 \rangle \subseteq \langle x_1, x_2, x_3, x_4 \rangle$.

6.3 State Schemas

A state schema consists of declarations of variables together with a predicate which imposes constraints on the variables. Schemas are usually written in a sectioned box-like format: the top part contains the declarations and the bottom part contains the predicate expressed in first-order predicate logic. Traditional logic symbols, such as negation '$\neg$ ', conjunction '$\wedge$', disjunction '$\vee$', universal quantification '$\forall$', implication '$\Rightarrow$', equivalence '$\Leftrightarrow$', etc. are available. If the predicate is omitted it is implicitly true, i.e. the variables need only conform to their types. An instance of a schema is a binding of values to the schema's variables consistent with their types and the constraints.

For example, the following schema characterises first quadrant coordinate pairs of integers. The predicate specifies that the variables of *Coord* are never negative.

$$
\begin{array}{|l}
\hline
\textit{Coord} \\
x : \mathbb{Z} \\
y : \mathbb{Z} \\
\hline
x \geqslant 0 \wedge y \geqslant 0 \\
\hline
\end{array}
$$

Schemas may be used as types and component variables are accessed using dot notation, e.g. given $c : Coord$, $c.x$ denotes c's x component.

The schema *Origin* below is a restriction of *Coord*: it includes *Coord* and has an additional predicate enforcing both x and y to be zero.

$$
\begin{array}{|l}
\hline
\textit{Origin} \\
\textit{Coord} \\
\hline
x = 0 \wedge y = 0 \\
\hline
\end{array}
$$

The meaning of schema inclusion is illustrated in the following expanded version of *Origin*.

$$
\begin{array}{|l}
\hline
\textit{Origin} \\
x : \mathbb{Z} \\
y : \mathbb{Z} \\
\hline
x \geqslant 0 \wedge y \geqslant 0 \\
x = 0 \wedge y = 0 \\
\hline
\end{array}
$$

The understanding is that all the variables and predicates of included schemas become part of the new schema, i.e. the declarations are simply collected together (*merged*) and the individual predicates are conjoined. It is an error to attempt to merge declarations of the same variable name but with different types, e.g $x : \mathbb{N}$ and $x : Person$. Note the Z convention that predicates which span several lines may elide end-of-line conjunction symbols.

Compact schemas can be written 'horizontally' as in:

$$Origin \mathrel{\widehat=} [\, Coord \mid x = 0 \land y = 0 \,]$$

6.4 Operation Schemas

An operation schema models state transitions by relating the values of variables before the operation (pre-values) to their values after the operation (post-values). Post-values are primed to distinguish them from their corresponding pre-values. Operations may have inputs (distinguished by names ending with '?'). Operation *AddCoord* which follows adds input coordinates $x?, y?$ to x, y respectively.

```
┌─ AddCoord ─────────────────────────
│ ΔCoord
│ x?, y? : ℤ
├──────────
│ x' = x + x?
│ y' = y + y?
└────────────────────────────────────
```

Operations may also have outputs (identified by names ending with '!'). In the operation *AddCoord*, $\Delta Coord$ indicates that both primed and unprimed versions of the variables and predicates within *Coord* are included. The precise meaning is given by the following expanded version.

```
┌─ AddCoord ─────────────────────────
│ x, x' : ℤ
│ y, y' : ℤ
│ x?, y? : ℤ
├──────────
│ x ≥ 0 ∧ y ≥ 0
│ x' ≥ 0 ∧ y' ≥ 0
│ x' = x + x?
│ y' = y + y?
└────────────────────────────────────
```

The Schema Calculus

Z has a schema calculus for modifying and combining schemas: the calculus operators enable new schemas to be defined using existing schemas in a compact and readable way.

If A and B are schemas, the schema conjunction $A \wedge B$ is defined as the schema obtained by merging the declarations of A and B and conjoining their predicates. Thus in $AddCoord$, above $\Delta Coord \mathrel{\widehat{=}} Coord \wedge Coord'$, where priming a schema means priming its declared variables. Other operators included in the calculus are schema negation (which negates the predicate), disjunction, implication and equivalence (which combine the predicates with the corresponding logic operators). For further details on the calculus, consult [1, 21].