SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-02

Using Object-Z to Specify Object-Oriented Programming Languages

Jin Song Dong and Roger Duke

February 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

Using Object-Z to Specify Object-Oriented Programming Languages

Jin Song Dong **Roger Duke**

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

jason@cs.uq.oz.au rduke@cs.uq.oz.au

Abstract

In this paper, we present a case study for using formal object-oriented specification languages, such as Object-Z, to specify the semantics of object-oriented programming languages. The key idea in this case study is to model language constructs, such as expressions, statements, methods and classes, as objects and to specify the semantics of each language construct using an Object-Z class. This case study demonstrates the feasibility of using object orientation to define the semantics of programming languages. The advantage of this approach is that the abstract syntax and static and dynamic semantics of each programming language construct can be captured within one semantic structure. A consequence is that the semantics of a programming language can be readily extended when the language is enhanced. This paper in effect presents an object-oriented approach to the formalisation of the concepts of object-orientation.

Keywords: formal specification, Object-Z, object-orientation, programming language semantics.

1 Introduction

Formal descriptions of programming languages support language standardisation, program verification and software reliability. The formal specification languages VDM[18] and Z[26] have been used as meta-languages to describe the denotational semantics of programming languages[2, 27]. The general lack of modularity and reusability of the traditional denotational semantics representations has been realised in the literature as a drawback[20]. In this paper, we take an object-oriented approach to specify the denotational semantics of object-oriented programming languages (OOPLs) using the Object-Z[13, 15, 23] notation (Object-Z is an object-oriented extension of Z). In this approach, the abstract syntax and static and dynamic semantics of each programming language construct are specified by a single Object-Z class construct. Not only does this simplify the semantic domains, but the semantics of a programming language can be readily extended when the language is enhanced.

The denotational semantics of object-oriented programming languages has been studied and developed. Wolczko[28] and Kamin[19] presented denotational semantics of the Smalltalk programming language. The denotational semantics of inheritance has been investigated by Cook and Palsberg[5] based on Cardelli's work[4]. Others [17, 22] have also contributed to the research work on the semantics of OOPLs. The denotational semantics of the parallel object-oriented programming language POOL has also been studied in [1].

Our approach to the semantics of OOPLs is quite different. The key idea behind this approach is to use a class to model each language construct of OOPLs. That is, each language construct of OOPLs, such as expressions, statements, methods and classes, and even run-time program entities such as objects (instances of program classes), are modelled by Object-Z classes, whereas traditionally they have been modelled as more primitive mathematical entities such as mappings or composite types (in VDM terms). The aim of our approach is to achieve a more structured and extendible semantics.

In Section 2, the object-oriented approach used to model OOPLs is outlined. In Section 3, a small OOPL is specified in Object-Z and then in Section 4 the specification is extended to accommodate more language constructs and demonstrate the extendibility of the model. Familiarity with the basic features of the Object-Z notations as given in [13, 23] is assumed.

To distinguish between the terminology (such as 'class' and 'object') used in both an OOPL (the language being modelled) and in Object-Z (the meta language of the model), the sans serif type style is used to express terms in OOPLs, e.g. class refers to the term 'class' in the context of OOPLs, rather than in the context of Object-Z.

2 An OO Approach to Modelling OOPLs

If we take an object-oriented approach to specify OOPLs, then an object-oriented program can be modelled as an object which consists of a collection of static objects, such as expressions, methods, statements and classes, and also dynamic objects, such as instances of classes (objects). Given this, we can use different Object-Z classes[1]

[1] In Object-Z, a class is also a type: instances of that type are the identities which reference objects of the class.

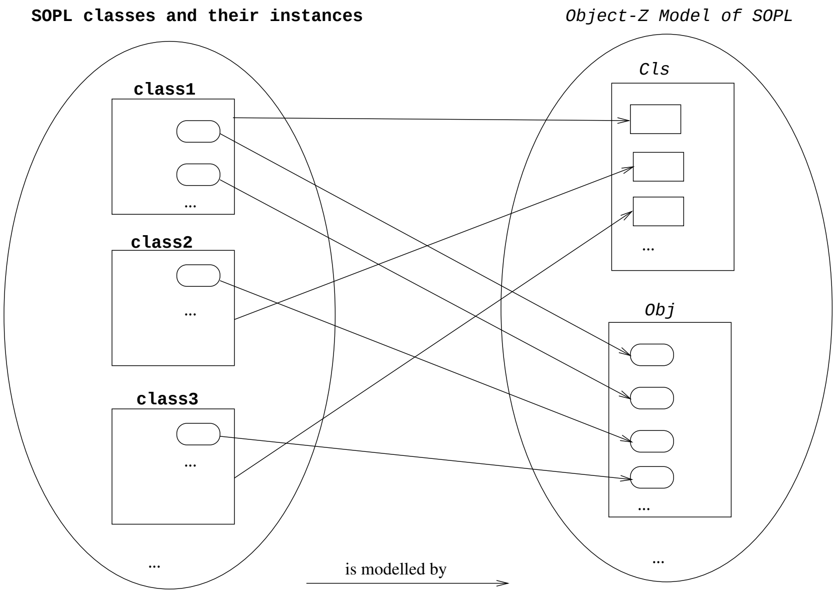

Figure 1: Classes and Objects of SOPL and their Object-Z Model.

to model the different language constructs of an OOPL. For instance, if expressions are modelled by an Object-Z class, say *Expr*, then any individual expression in a program is modelled as an instance of the Object-Z class *Expr*. Similarly, if the construct method is modelled by an Object-Z class, say *Method*, then any individual method in a program is modelled as an instance of the Object-Z class *Method*.

When modelling a language construct, if it needs to be further classified into more specific constructs, then one can use different Object-Z classes to model these constructs. For example, suppose there are two different kinds of statements, such as an assignment statement (like a := exp;) and a method-call (like a.method(...);). Different Object-Z classes, say *AssignStmt* and *MethodCall*, can then model the assignment statement construct and the method-call construct respectively. Any assignment statement in a program is modelled as an instance of the Object-Z class *AssignStmt* and any method-call statement in the program is modelled as an instance of the Object-Z class *MethodCall*. Alternatively, if an entity, say s, refers to a general statement (either an assignment or a method-call) then s can be modelled in Object-Z as an element of the class-union[8] $AssignStmt \cup MethodCall$. (Class-union is a generalised polymorphic type that explicitly constructs collections of classes. For a detailed discussion of class-union see [6, 8].)

If the OOPL construct class is modelled by an Object-Z class, say *Cls*, then any individual class in a program is modelled as an instance of the Object-Z class *Cls*. A run-time object in a program can also be modelled as a class, say *Obj*, in Object-Z: that is, each individual object in the program is modelled as an instance of the

Object-Z class *Obj*. (Figure 1 shows the relationship of SOPL and the Object-Z model of SOPL.) Again, if objects need to be further classified into different categories, say predefined integer objects and those objects which are instantiated from classes defined by a programmer, then different Object-Z classes can be used to model these categories.

Classes and objects are the crucial concepts in object-oriented programming. The relationship between a class and an object in a program is that an object is an instance of a class and a class is a collection of objects. If classes and objects are modelled as Object-Z classes *Cls* and *Obj* respectively, then the relationship between a class and an object can be modelled as cross-references[2] in *Cls* and *Obj*:

Cls

$insts : \mathbb{P}\, Obj_{©}$

$\ldots$

$\forall\, o : insts \bullet o.type = self$

$\ldots$

$\ldots$

Obj

$type : Cls$

$\ldots$

$\ldots$

Note that *Cls* and *Obj* are examples in this section only.

In *Cls*, the semantics of the subscript '©' (unique object containment[3][9]) appended to the type of the attribute *insts* (instances) implies that:

$$\forall\, c_1, c_2 : Cls \bullet c_1 \neq c_2 \Rightarrow c_1.insts \cap c_2.insts = \varnothing$$

i.e. the object containment notation captures the notion that objects are partitioned into classes.

3 An Object-Oriented Model of an OOPL

In this section, a simple object programming language (SOPL) is introduced and its concrete syntax presented. Then the semantics of SOPL is specified in Object-Z. In our model of SOPL we model run-time objects and classes individually, with their relationship modelled by cross-referencing. As the definition of classes requires the understanding of SOPL constructs, such as methods, models of run time objects and variable references (pointers to objects) are first developed, followed by models of expressions, statements, methods, classes and programs.

[2] In contrast to the schema construct of Z, self and mutual recursive referencing is allowed in the class construct of Object-Z[14].

[3] Unique object containment ensures that no object directly or indirectly contains itself; and no object is directly contained in two distinct objects. For a detailed discussion see [9].

3.1 SOPL and Its Concrete Syntax

The simple object programming language (SOPL) modelled in this paper is based on the typed basic object programming language BOPL[4][21] with single-inheritance. The rules for assignments and method-calls follow Eiffel's conformance rule. For example, for the assignment $\mathsf{b} := \mathsf{a}$ (where the type of b is $\mathsf{class\ B}$ and the type of a is $\mathsf{class\ A}$) to be valid, either A must be equal to B or A must be a descendant of B. In this section, we suppose SOPL contains only one predefined type, *integer*; in Section 4, the language is extended to have also the predefined type *boolean*.

The concrete syntax of SOPL is:

$$
\begin{array}{rcl}
\mathit{Program} & \rightarrow & \mathit{Classes}\ \text{`Program'}\ \mathit{Id}\ \mathit{Declist}\ \mathit{SeqStmt} \\
\mathit{Classes} & \rightarrow & \epsilon \mid \mathit{Class}\ \mathit{Classes} \\
\mathit{Class} & \rightarrow & \text{`Class'}\ \mathit{Id}\ \ [\text{`inherit'}\ \mathit{Id}\ ]\ \mathit{Declist}\ \mathit{Methods} \\
\mathit{Id} & \rightarrow & \mathit{Letter} \mid \mathit{Letter}\ \mathit{Id} \\
\mathit{Letter} & \rightarrow & \text{`a'} \mid \text{`b'} \mid \text{`c'} \mid \ldots\ \text{`z'} \\
\mathit{Declist} & \rightarrow & \epsilon \mid \mathit{Declaration}\ \mathit{Declist} \\
\mathit{Declaration} & \rightarrow & \mathit{Var}\ \text{`:'}\ \mathit{Type}\ \text{`;'} \\
\mathit{Methods} & \rightarrow & \epsilon \mid \mathit{Method}\ \mathit{Methods} \\
\mathit{Method} & \rightarrow & \text{`Method'}\ \mathit{Id}\ \text{`('}\ \mathit{Declist}\ \text{`);'}\ \mathit{SeqStmt} \\
\mathit{Var} & \rightarrow & \mathit{Id} \\
\mathit{Type} & \rightarrow & \mathit{Id} \mid \text{`Int'} \\
\mathit{SeqStmt} & \rightarrow & \text{`begin'}\ \mathit{Stmtseq}\ \text{`end'} \\
\mathit{Stmtseq} & \rightarrow & \epsilon \mid \mathit{Stmt}\ \mathit{Stmtseq} \\
\mathit{Stmt} & \rightarrow & \mathit{AssignStmt} \mid \mathit{MethodCall} \mid \mathit{CreateStmt} \mid \mathit{ReleaseStmt} \\
\mathit{AssignStmt} & \rightarrow & \mathit{Var}\ \text{`:='}\ \mathit{Exp}\ \text{`;'} \\
\mathit{Exp} & \rightarrow & \mathit{IntExp} \mid \mathit{Var} \mid \mathit{ComponentExp} \\
\mathit{IntExp} & \rightarrow & \mathit{IntConst} \mid \mathit{IntExp}\ \text{`+'}\ \mathit{IntExp} \\
\mathit{IntConst} & \rightarrow & \mathit{Digit} \mid \mathit{Digit}\ \mathit{IntConst} \\
\mathit{Digit} & \rightarrow & \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'} \\
\mathit{ComponentExp} & \rightarrow & \mathit{Id}\ \text{`.'}\ \mathit{Id} \mid \mathit{Id}\ \text{`.'}\ \mathit{ComponentExp} \\
\mathit{MethodCall} & \rightarrow & \mathit{Id}\ \text{`.'}\ \mathit{Id}\ \text{`('}\ \mathit{Explist}\ \text{`);'} \\
\mathit{Explist} & \rightarrow & \epsilon \mid \mathit{Exp}\ \text{`,'}\ \mathit{Explist} \\
\mathit{CreateStmt} & \rightarrow & \text{`Create('}\ \mathit{Id}\ \text{`);'} \\
\mathit{ReleaseStmt} & \rightarrow & \text{`Release('}\ \mathit{Id}\ \text{`);'}
\end{array}
$$

where ϵ denotes the empty string. Here is an example of a small SOPL program.

[4]The language BOPL contains the main features (classes, objects, assignments and late binding) common to Simula, Smalltalk, C++ and Eiffel.

```
Class Point
x: Int; y: Int;
Method SetToOne();
begin
x := 1; y := 1;
end
Method Move(a: Int; b: Int;);
begin
x := x + a;
y := y + b;
end
```

```
Class TwoPoints
p: Point; q: Point;
Method Init();
begin
Create(p); Create(q);
p.SetToOne(); q.SetToOne();
end
Method MoveQbyP();
begin
q.Move(p.x, p.y);
end
```

```
Program TP
twopts: TwoPoints;
begin
Create(twopts); twopts.Init; twopts.MoveQbyP();
end
```

In the following sub-sections, the Object-Z model of SOPL is presented.

3.2 Modelling Identifiers, Objects and Variable References

3.2.1 Modelling Identifiers

Let the Object-Z class *Id*

denote all the possible identifiers (including program names, class names, class attributes and method names) in SOPL. Because we are not interested in the content of an identifier, the class *Id* has no attributes or operations. *Id* is modelled as a class (rather than a given type, such as [*Id*]) so that various Object-Z class modifiers such as object containment can be applied as required.

3.2.2 Modelling Objects

Run-time objects in SOPL are integer objects (predefined) and objects which are instantiated from the user-defined classes. An integer object has an integer value, while an instantiated object has a set of variable references (pointers to objects). Objects of user-defined classes need to be explicitly created by a create-statement.

When an object is created, it is assumed that attributes (variable references) of the object are initially unlocated. In this case, the unlocated variable reference points to a special kind of object called the *nil* object. In this model, three Object-Z classes *IntObj*, *InstObj* and *NilObj* define these three different kinds of objects (i.e. integer object, instantiated object, nil object) respectively. Objects in general are modelled as a class-union:

$$pObject \mathrel{\widehat=} IntObj \cup InstObj \cup NilObj$$

Let the Object-Z class

$$\begin{array}{|l}
\hline
\textit{NilObj} \\
\begin{array}{|l|}
\hline
obj : pObject \\
\hline
obj = self \\
\hline
\end{array}
\quad
\begin{array}{|l|}
\hline
\textit{Output} \\
obj! : pObject \\
\hline
obj! = obj \\
\hline
\end{array} \\
\hline
\end{array}$$

model a nil object. A nil object has an attribute *obj* which is a self identity so that the self identity can be broadcast. The class *NilObj* is inherited to define the classes *IntObj* and *InstObj* as will be shown below.

The range of the integer values in SOPL is from -2^{29} to $2^{29} - 1$,

$$IntRange == -2^{29}..(2^{29} - 1).$$

We view each integer value of SOPL as a unique object. Let

$$\begin{array}{|l}
\hline
\textit{IntObj} \\
NilObj \\
\begin{array}{|l|}
\hline
type : IntClass \\
val : IntRange \\
\hline
\#\{i : IntObj \bullet i.val\} = \#IntRange = \#IntObj \\
\hline
\end{array} \\
\hline
\end{array}$$

model all possible integer objects in SOPL. (The integer class is modelled by *IntClass* which will be defined in Section 3.5.) The class invariant of the class *IntObj* specifies that each integer object is uniquely associated with an integer value of *IntRange* and vice-versa.

Let the Object-Z class

$$\begin{array}{|l}
\hline
\textit{InstObj} \\
NilObj \\
\begin{array}{|l|}
\hline
type : UsrdefClass \\
attris : \mathbb{P}\, VarRef_{©} \\
\hline
\end{array} \\
\hline
\end{array}$$

model instantiated objects from user defined classes and nil objects.

In *IntObj* or *InstObj*, the attribute *type* specifies that the type of an object is the class to which the object belongs. In *IntObj*, the attribute *val* specifies that an integer object has an integer value, while in *InstObj*, the attribute *attris* specifies that an instantiated object has a set of attribute variable references (variable references are modelled by the class *VarRef* which will be defined in the next subsection and the user-defined class is modelled by the *UsrdefClass* which will be defined in Section 3.5). The object containment notation attached to the type of the *attris* in *InstObj* specifies that each instantiated object has its own distinct local variable references.

3.2.3 Modelling Variable References

Following the classification of objects, variable references to objects can be divided into variable references to integer objects and variable references to instantiated objects of user-defined classes.

$$VarRef \mathrel{\widehat{=}} IntRef \cup InstRef$$

An integer variable reference and a variable reference of an instantiated object have similarities. For example, they each have an identifier, a declared type (class) and a pointer to an object, and their common operations enable the reference to be reattached to another object and to output the object to which it refers. Let

$$\begin{array}{l}
\underline{\quad CommonRef \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
\quad \begin{array}{|l}
\hline
id : Id \\
type : \mathbb{P}\,Class \\
obj : \mathbb{P}\,Object \\
\hline
\end{array}
\quad
\begin{array}{|l}
\underline{\ \textsc{Init}\ \qquad\qquad} \\
obj \in NilObj \\
\hline
\end{array}
\quad
\begin{array}{|l}
\underline{\ Reattach\ \qquad} \\
\Delta(obj) \\
obj? : \mathbb{P}\,Object \\
\hline
obj' = obj? \\
\hline
\end{array} \\
\\
\quad Output \mathrel{\widehat{=}} obj.Output \\
\hline
\end{array}$$

model the common structure of these variable references. It is assumed that every variable reference points initially to a nil object.

The integer variable references and instantiated variable references can be modelled as:

$$\begin{array}{|l}
\underline{\ IntRef\ \qquad\qquad\qquad} \\
CommonRef \\
\\
\quad \begin{array}{|l}
\hline
type \in IntClass \\
obj \notin NilObj \Rightarrow \\
\quad obj.type = type \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\underline{\ InstRef\ \qquad\qquad\qquad} \\
CommonRef \\
\\
\quad \begin{array}{|l}
\hline
type \in UsrdefClass \\
obj \notin NilObj \Rightarrow \\
\quad (type = obj.type \lor \\
\quad type \in obj.type.scontain) \\
\hline
\end{array} \\
\hline
\end{array}$$

The class invariants of *IntRef* and *InstRef* specify that a variable reference can point to an object whose class is the declared type of the reference. Furthermore, the invariant of *InstRef* specifies polymorphism by allowing the variable reference to point to an object of any derived class of the declared class. The term $obj.type.scontain$ specifies the collection of all the parent (inheritance) classes of the class specified by $obj.type$. (See Section 3.5 for the specification of classes. For a detailed description of the semantics of the attribute $scontain$ see [9].)

3.3 Modelling Expressions

The meaning of an expression denotes an object, while the meaning of a statement is to let variable references reattach to other objects. An expression in SOPL can be an instantiated variable reference (e.g. twopts), an integer expression (e.g. x + a) or a component expression (e.g. twopts.p). In Object-Z it is modelled as a class-union

$$Exp \mathrel{\widehat=} InstRef \cup IntExp \cup ComponentExp$$

An integer expression can be an integer object (constant) or an integer variable reference or a plus expression or a component integer expression (e.g. 'twopts.p.x' where the type of x is integer), i.e.

$$IntExp \mathrel{\widehat=} IntObj \cup IntRef \cup PlusExp \cup IntComponentExp$$

3.3.1 Modelling Plus Expressions

Plus expressions can be modelled as a class:

$$
\begin{array}{|l}
\multicolumn{1}{l}{PlusExp} \\
\hline
\begin{array}{|l}
\hline
e_1, e_2 : IntExp_{Ⓢ} \\
type : IntClass \\
\Delta \\
obj : IntObj \\
\hline
obj.val = (e_1.obj.val + e_2.obj.val) \\
\hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{Output} \\
\hline
obj! : \mathbb{P}\,Object \\
\hline
obj! = obj \\
\hline
\end{array} \\
\hline
\end{array}
$$

The subscript 'Ⓢ' (sharable containment[5]) appended to the type of e_1 and e_2 specifies that the object reference structure of a plus expression is acyclic (e.g. an expression cannot be a sub-expression directly or indirectly of itself). The type of a plus expression is the integer class which is specified by the attribute *type*. The meaning

[5]Sharable containment is a weaker notion of containment. It allows an object to be directly contained by more than one containing object. For a detailed discussion see [9].

(dynamic semantics) of a plus expression is an integer object whose value depends on the sub-components of the plus expression; this meaning is specified by the secondary attribute[6] *obj*. A feature of a plus expression is to output its meaning which is specified by the operation *Output*.

The component integer expression will be modelled after the following general component expression is specified.

3.3.2 Modelling Component Expressions

Syntactically, a component expression is a list of identifiers with dots as the separation marks. Abstractly, a component expression is a list of variable references with a validity constraint (static semantic condition). For example, suppose twopts.p.x is a component expression, then p must be an attribute variable of the object referenced by twopts and x must be an attribute of the object referenced by twopts.p. The type of a component expression is the type of the last variable reference in the list. The meaning of a component expression denotes the object referenced by the last variable reference in the list. The following is a model of component expressions:

$$
\begin{array}{l}
\textit{ComponentExp} \\
\hline
\quad \textit{first} : \textit{VarRef}_{\circledS} \\
\quad \textit{list} : \operatorname{seq} \textit{Id} \\
\quad \Delta \\
\quad \textit{type} : \textit{pClass} \\
\quad \textit{obj} : \textit{pObject} \\
\hline
\quad \#\textit{list} > 1 \\
\quad \exists\, s : \operatorname{seq} \textit{VarRef} \bullet \#s = \#\textit{list} \wedge \{i : \operatorname{dom} s \bullet (i, s(i).\textit{id})\} = \textit{list} \\
\qquad \forall\, i : 1..(\#s - 1) \bullet s(i).\textit{obj} \in \textit{InstObj} \wedge s(i+1) \in s(i).\textit{obj}.\textit{attris} \\
\qquad \textit{first} = \textit{head}(s) \wedge \textit{type} = \textit{last}(s).\textit{type} \wedge \textit{obj} = \textit{last}(s).\textit{obj} \\
\hline
\quad \textit{Output} \mathrel{\widehat{=}} [\textit{obj}! : \textit{pObject} \mid \textit{obj}! = \textit{obj}]
\end{array}
$$

The class invariant specifies the dependency relationship between the first variable reference and the object to which the expression refers.

[6]Attributes of a class are partitioned into primary and secondary. The values of the primary attributes determine the state of an object; the values of the secondary attributes depend upon the primary attributes of this or other objects in the system and enable an object to retain information about various aspects of the state of the system. For a detailed discussion on secondary attributes see [14, 11].

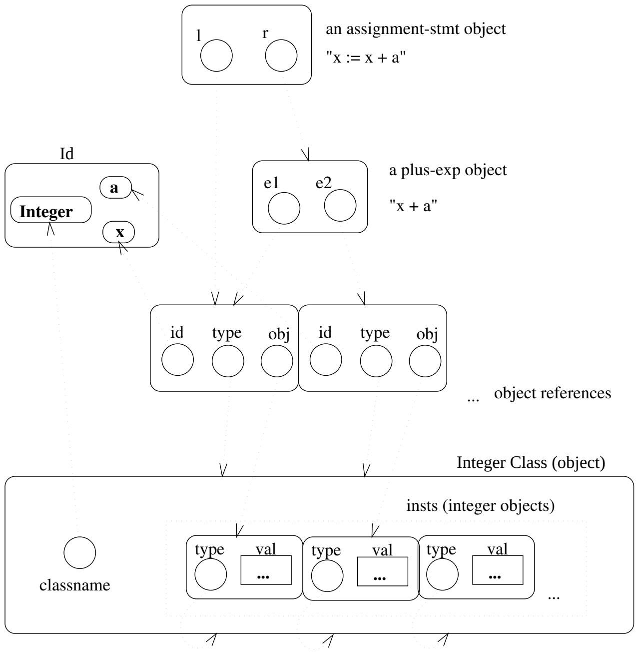

Figure 2: Object Reference Structure of an *Assignment* Statement.

Now a component integer expression is just a particular kind of component expression.

$$
\begin{array}{|l}
\hline
\textit{IntComponentExp} \\
\textit{ComponentExp} \\
\hline
\textit{obj} \in \textit{IntObj} \\
\hline
\end{array}
$$

3.4 Modelling Statements

In SOPL, a statement can be an assignment statement, a create-statement, a release-statement or a method-call statement.

$$\textit{Stmt} \mathrel{\widehat{=}} \textit{AssignStmt} \cup \textit{CreateStmt} \cup \textit{ReleaseStmt} \cup \textit{MethodCall}$$

3.4.1 Modelling Assignment Statements

An assignment statement consists of a left hand side variable reference and a right hand side expression. It is modelled as:

$$
\begin{array}{|l}
\hline
\textit{AssignStmt} \\
\hline
l : \textit{VarRef}_{Ⓢ} \\
r : \textit{Exp}_{Ⓢ} \\
\hline
l.\textit{type} = r.\textit{type} \lor l.\textit{type} \in r.\textit{type}.\textit{scontain} \\
\hline
\textit{Execute} \mathrel{\widehat{=}} r.\textit{Output} \parallel l.\textit{Reattach} \\
\hline
\end{array}
$$

The class invariant specifies that the variable reference (left hand side of an assignment) can reattach to an object of the declared class of the reference or any object of a descendent class of the declared class.

In the definition of the class *AssignStmt*, the abstract syntax of an assignment statement is represented by state attributes l (left hand side of the assignment) and r (right hand side of the assignment). The static semantics of the assignment statement is captured by the state invariant. The dynamic semantics of the assignment statement is modelled by the operation *Execute* which relocates the variable reference l to point to the object which is denoted by the expression r. The parallel operator '$\|$' conjoins constraints and equates variables with the same name and also equates inputs and outputs with the same basename. The object reference structure of an *assignment* statement is illustrated by Figure 2.

3.4.2 Modelling Create and Release Statements

The meaning of a create-statement on a reference, say Create(p) (where p: Point), is to let the declared class Point make a new Point object and to let the reference p point to the new object. In contrast, the meaning of a release-statement on a reference, say Release(p) (where p: Point), is to let the reference p point to a nil object. The statements to create a new object or to release an object from a reference can be modelled by the following classes.

CreateStmt

$r : InstRef$

$Execute \mathrel{\widehat=}$
$\quad r.type.Create \parallel r.Reattach$

ReleaseStmt

$r : InstRef$

$Execute \mathrel{\widehat=} [n : NilObj] \bullet$
$\quad n.Output \parallel r.Reattach$

In the *CreateStmt* definition of the operation *Execute*, *r.type* refers to a user-defined class (the declared type of r). *Create* is an operation of the *UsrdefClass* which will be defined in Section 3.5.

3.4.3 Modelling a Sequence of Statements

Before specifying a method-call, we model a sequence of statements. A sequence of statements consists of a list of statements. Its meaning is to execute the statements in the sequence one by one.

$$\begin{array}{|l}
\hline
\textit{SeqStmt} \\
\hline
\textit{stmts} : \operatorname{seq} \textit{Stmt}_{\text{Ⓢ}} \\
\hline
\textit{Execute} \mathrel{\widehat=} \mathbin{⨟}\ i : 1..\#\textit{stmts} \bullet \textit{stmts}(i).\textit{Execute} \\
\hline
\end{array}$$

Note that the operation *Execute*, namely, $⨟\ i : 1..\#stmts \bullet stmts(i).Execute$ is an abbreviation for:

$stmts(1).Execute ⨟ \ldots ⨟ stmts(\#stmts).Execute.$

3.4.4 Modelling Methods

To define a method-call statement, first we need to model a method of a class. A method is an operation which is defined in a class. The internal state of an object of a class can be changed by the execution of a method of the class. A method consists of a name, a formal parameters list (*fpl*), a set of local variables and a body of a sequence of statements.

$$\begin{array}{|l}
\hline
\textit{Method} \\
\hline
\textit{id} : \textit{Id} \\
\textit{fpl} : \operatorname{seq} \textit{VarRef}_{\text{Ⓢ}} \\
\textit{locvars} : \mathbb{P}\, \textit{VarRef}_{\text{Ⓢ}} \\
\textit{body} : \textit{SeqStmt}_{\text{Ⓢ}} \\
\hline
\#\{r : (\textit{VarRef} \cap \textit{scontain}) \bullet r.\textit{id}\} = \#(\textit{VarRef} \cap \textit{scontain}) \\
\hline
\textit{Execute} \mathrel{\widehat=} \textit{body}.\textit{Execute} \\
\hline
\end{array}$$

The class invariant specifies that, in a method, any two variable references having the same identifier must be the same variable reference. The meaning of a method is the execution of the body statements of the method (*body.Execute*). Notice that there is no '©' appended on the type of the attribute *id* so that two distinct methods in two different classes are allowed to have the same name (unlike in the case of class names).

3.4.5 Modelling Method-call Statements

A method-call statement in SOPL is like a procedure call in Pascal. It consists of a receiver (variable reference), a method of the declared class of the receiver and an actual parameter list (*apl*).

$$
\begin{array}{l}
\textit{MethodCall} \\
\hline
recv : InstRef_{\circledS} \\
m : Method \\
apl : \mathrm{seq}\, Exp_{\circledS} \\
\Delta \\
ps : VarRef \rightarrowtail\!\!\!\!\!\rightarrow VarRef \\
\hline
ps = \{(r_1, r_2) : VarRef \times VarRef \mid r_1 \in recv.obj.attris \land \\
\qquad r_2 \in m.body.scontain \land r_1.id = r_2.id\} \\
m \in recv.type.scontain \qquad [\mathsf{inv1}] \\
\#apl = \#m.fpl \qquad [\mathsf{inv2}] \\
\forall\, i : \mathrm{dom}\, apl \bullet m.fpl(i).type = apl(i).type \qquad [\mathsf{inv3}] \\
\qquad \lor\ m.fpl(i).type \in apl(i).type.scontain \\
\hline
PmtSub \mathrel{\widehat=} \bigwedge i : \mathrm{dom}\, apl \bullet apl(i).Output \parallel m.fpl(i).Reattach \\
RelocM \mathrel{\widehat=} \bigwedge (r_1, r_2) : ps \bullet r_1.Output \parallel r_2.Reattach \\
RelocR \mathrel{\widehat=} \bigwedge (r_1, r_2) : ps \bullet r_2.Output \parallel r_1.Reattach \\
Execute \mathrel{\widehat=} [recv.obj \notin NilObj] \bullet\ PmtSub \mathbin{⨟} RelocM \mathbin{⨟} m.Execute \mathbin{⨟} RelocR
\end{array}
$$

The dependent attribute ps pairs variable references of the same name which both appeare as part of the the receiver object's state and are used in the calling method's body statements.

The class invariant inv1 specifies that the calling method is defined in the declared class of the receiver (variable reference). The class invariant inv2 specifies that the number of the actual parameters from a method-call statement is equal to the number of formal parameters in the calling method. The class invariant inv3 specifies the conformance rule of parameter substitutions in a method-call.

The operation $PmtSub$ specifies the substitution of the actual (expression) parameters of the method-call to the formal (variable reference) parameters of the method.

The operation $RelocM$ specifies the relocation of the variable references occurring in the method body to point to the objects referenced by the attribute variables of the receiver object. The operation $RelocR$ specifies the reverse of the process captured by the operation $RelocM$. With these process relocations, the execution of the method ($m.Execute$) can effectively update the state of the object.

3.5 Modelling Classes

A class is a template for objects: each object of the class has a state which conforms to the signature of the class and the state of the object can be changed by invoking a method of the class.

In object-oriented programming languages, classes are often used as types. There are two kinds of classes, predefined classes (e.g. integer class) and user-defined classes defined by the programmer. Let

$$\begin{array}{|l}
\textit{IntClass} \\
\hline
\begin{array}{|l}
classname : Id_{©} \\
insts : \mathbb{P}\, IntObj_{©} \\
\hline
\#IntClass = 1 \\
insts = IntObj
\end{array} \\
\hline
\end{array}$$

model the integer class of the language. In the class invariant part of *IntClass*, the predicate $\#IntClass = 1$ specifies that there is only one integer class in SOPL. The second conjunct specifies that all possible integer objects are contained by the integer class.

From the inheritance point of view, there are two kinds of user-defined classes — root classes (classes defined without inheriting another class) and derived classes (classes defined by inheriting another class). In Object-Z, the user-defined classes in a program can be modelled as a class-union:

$$UsrdefClass \mathrel{\widehat{=}} RootClass \cup DerivedClass$$

where *RootClass* and *DerivedClass* will be defined later. Similarly, SOPL classes can also be defined as a class-union:

$$pClass \mathrel{\widehat{=}} IntClass \cup UsrdefClass$$

As a root class and a derived class are quite similar (they all have a class name, a signature, a set of methods and a set of created instances), we can model their common part as the Object-Z class *CommonClass*, and then the *RootClass* and *DerivedClass* can be defined by inheriting *CommonClass*.

$$
\begin{array}{l}
\underline{\textit{CommonClass}} \\
\quad \textit{classname} : \textit{Id}_{\copyright} \\
\quad \textit{sig} : \textit{Id} \nrightarrow \textit{pClass} \\
\quad \textit{methods} : \mathbb{P}\, \textit{Method}_{\copyright} \\
\quad \textit{insts} : \mathbb{P}\, \textit{InstObj}_{\copyright} \\
\quad \Delta \\
\quad \textit{totalsig} : \textit{Id} \nrightarrow \textit{pClass} \\
\quad \textit{totalms} : \mathbb{P}\, \textit{Method} \\
\hline
\quad \forall\, o : \textit{insts} \bullet o.\textit{type} = \textit{self} \\
\quad \forall\, o : \textit{insts} \bullet \{ r : o.\textit{attris} \bullet (r.\textit{id}, r.\textit{type}) \} = \textit{totalsig} \\
\quad \#\textit{totalms} = \#\{ m : \textit{totalms} \bullet m.\textit{id} \} \\
\quad \forall\, m : \textit{totalms} \bullet \{ r : \textit{VarRef} \mid \\
\qquad r \in m.\textit{scontain} \setminus \operatorname{ran} m.\textit{fpl} \setminus m.\textit{locvars} \bullet (r.\textit{id}, r.\textit{type}) \} \subseteq \textit{totalsig} \\
\\
\quad \underline{\textit{Create}} \\
\quad \Delta(\textit{insts}) \\
\quad \textit{obj}! : \textit{pObject} \\
\hline
\quad \exists\, o : \textit{InstObj} \bullet \forall\, r : o.\textit{attris} \bullet r.\textsc{Init} \\
\qquad\qquad o \notin \textit{insts} \land \textit{insts}' = \textit{insts} \cup \{o\} \land \textit{obj}! = o
\end{array}
$$

The object-oriented view of the example class Point (given in Section 3.1) can be illustrated in a graphical form in Figure 3.

In *CommonClass*, *sig* denotes the signature of the attributes explicitly declared in a class, while the dependent attribute *totalsig* denotes the total signature of the variables explicitly declared in the class and implicitly inherited; *methods* denotes the methods explicitly defined in the class, while *totalms* denotes all the methods explicitly defined in the class and implicitly inherited (the precise meaning of *totalsig* and *totalms* is defined in the class invariants of *RootClass* and *DerivedClass*); *insts* denotes all the created objects of the class.

The subscript '$\copyright$' appended on the type of *classname* specifies that any two different classes have different names. The subscript '$\copyright$' appended on the type of *methods* indicates that a class contains a distinct set of (locally) explicitly defined methods. The subscript '$\copyright$' appended on the type of *insts* specifies that objects are partitioned into classes.

The class invariants of *CommonClass* specify that

- all objects contained by a class must belong to the class;
- every object must have the same signature as its class;

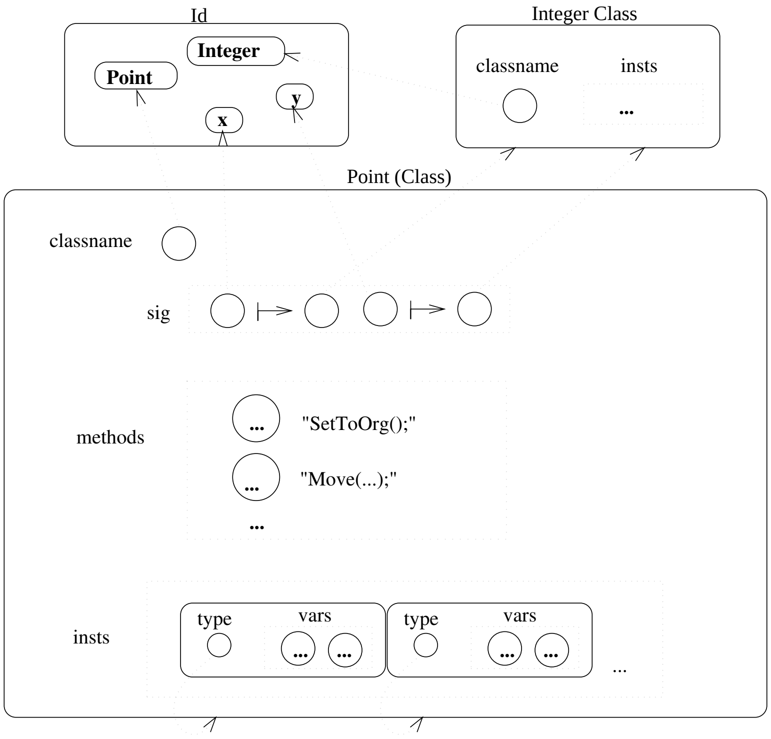

Figure 3: Object Reference Structure of the 'Point' Class.

- every method of a class has a distinct method name; (note that it still allows two different classes to have two different methods with the same name);
- variable references (except formal parameters and local variables) accessed in a method must be declared in the class. Note that the term $m.scontain$ denotes the objects which are directly and indirectly contained (but sharable) by the object referenced by m.

The operation *Create* specifies that a new object can be created by its class and the address (identity) of the object can be output to the environment. Although we are not concerned with the way in which a new object is created, by adding this operation in *CommonClass*, we capture the notion that a class is a factory for its objects. Also, note that the operation *Create* is only defined for modelling a user-defined class, whereas in the case of modelling the integer class, it is not necessary to define such an operation because all integer objects are predefined and do not need to be explicitly created by a create-statement.

Now, *RootClass* and *DerivedClass* can be defined by inheriting *CommonClass*:

$$
\begin{array}{|l}
\textit{RootClass} \\
\hline
\textit{CommonClass} \\
\begin{array}{|l}
\hline
totalsig = sig \\
totalms = methods \\
\hline
\end{array} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\textit{DerivedClass} \\
\hline
\textit{CommonClass} \\
\begin{array}{|l}
\hline
parclass : UsrdefClass_{\circledS} \\
\hline
totalsig = sig \cup parclass.totalsig \\
totalms = methods \cup parclass.totalms \\
\hline
\end{array} \\
\hline
\end{array}
$$

In *DerivedClass*, *parclass* denotes the parent class of a derived class. The subscript '$_{\circledS}$' (sharable containment)[9] appended on the type of *parclass* precisely specifies that the single inheritance structure is acyclic (i.e. a class cannot directly or indirectly inherit itself). The attribute *methods* (inherited from *CommonClass*) denotes the locally defined additional methods.

3.6 Programs

A program is similar to a method of a class. It consists of a set of declared variable references and a sequence of statements to manipulate them.

$$
\begin{array}{|l}
\textit{Program} \\
\hline
\begin{array}{|l}
\hline
vars : \mathbb{P}\ VarRef \\
body : SeqStmt \\
\hline
\forall r_1, r_2 : vars \bullet r_1 \neq r_2 \Rightarrow r_1.id \neq r_2.id \\
body.scontain \cap VarRef = vars \\
\hline
\end{array}
\quad
\begin{array}{|l}
\textsc{Init} \\
\hline
\forall r : vars \bullet r.\textsc{Init} \\
\hline
\end{array} \\
Execute \mathrel{\widehat{=}} body.Execute \\
\hline
\end{array}
$$

The first class invariant specifies that any two declared variables have different names in a program. The second class invariant specifies that every occurrence of variable references in the body of the program must be declared in *vars*.

4 The Ease of Extendibility

One aim of using an object-oriented approach in the language model is to achieve the extendibility of the language model. Suppose that SOPL is extended to have the following language constructs:

- predefined *Boolean* class, predefined *Boolean* objects and *Boolean* variable references;

- less-than expression;
- while statement.

Then, in our model it is necessary only to:

(1) add the following classes:

BoolObj

NilObj

$type : BoolClass$
$val : \mathbb{B}$

$\#\{b : BoolObj \bullet b.val\} = \#\mathbb{B} = \#BoolObj$

BoolClass

$classname : Id_{©}$
$insts : \mathbb{P}\ BoolObj_{©}$

$\#BoolClass = 1$
$insts = BoolObj$

BoolRef

CommonRef

$id : Id$
$type : BoolClass$

$obj.type = type$

BoolComponentExp

ComponentExp

$obj \in BoolObj$

LessExp

$e_1, e_2 : IntExp_{Ⓢ}$
$type : BoolClass$
Δ
$obj : BoolObj$

$obj.val = (e_1.obj.val < e_2.obj.val)$

Output

$obj! : pObject$

$obj! = obj$

WhileStmt

$test : BoolExp_{Ⓢ}$
$body : SeqStmt_{Ⓢ}$

$$Execute \mathrel{\widehat=} ([test.obj.val] \land body.Execute) \mathbin{⨾} Execute \;[]\; [\neg\ test.obj.val]$$

(2) extend the class-unions *pClass*, *pObject* and *VarRef* to include the classes *BoolClass*, *BoolObj* and *BoolRef* respectively; add a new class-union

$$BoolExp \mathrel{\widehat=} BoolObj \cup BoolRef \cup LessExp \cup BoolComponentExp$$

to the model; then extend the class-unions *Exp* and *Stmt* to include the classes *BoolExp* and *WhileStmt* respectively.

All classes defined previously in Section 3 remain unchanged.

4.1 Method Redefinition in Class Inheritance

Now suppose we allow method-redefinition in class inheritance; then the *DerivedClass* can be defined as:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DerivedClass}} \\
\hline
\textit{CommonClass} \\
\begin{array}{|l}
\hline
parclass : UsrdefClass_{\circledS} \\
\Delta \\
oldms, newms : \mathbb{P}\, Method \\
\hline
oldms = \{ m_1 : parclass.totalms \mid \exists\, m_2 : methods \bullet m_1.id = m_2.id \} \\
newms = \{ m_1 : methods \mid \exists\, m_2 : parclass.totalms \bullet m_1.id = m_2.id \} \\
totalsig = sig \cup parclass.totalsig \\
totalms = methods \cup parclass.totalms - oldms \\
\forall\, m_1 : oldms, m_2 : newms \bullet m_1.id = m_2.id \Rightarrow \\
\qquad\qquad \# m_1.fpl = \# m_2.fpl \land \forall\, i : \operatorname{dom} m_2.fpl \bullet \\
\qquad\qquad\qquad m_2.fpl(i).id = m_1.fpl(i).id \\
\qquad\qquad\qquad (m_2.fpl(i).type = m_1.fpl(i).type \\
\qquad\qquad\qquad\quad \lor\ m_2.fpl(i).type \in m_1.fpl(i).type.scontain) \\
\hline
\end{array} \\
\hline
\end{array}
$$

The attribute *methods* (inherited from *CommonClass*) specifies the additional methods and redefined methods of a derived class. The dependent attribute *newms* specifies a set of redefined methods and *oldms* specifies a set of overwritten methods of the parent class (the precise meaning of *newms* and *oldms* is defined by the first two conjunctions in the class invariants of the *DerivedClass*). The last conjunction of the class invariant specifies the conformance rule of method redefinition.

It should be noted that when specifying the semantics of a programming language, the meta language also needs to have a clear semantics. The semantics of Object-Z has been developed in [12, 16, 25][7]. The formal semantics of the Object-Z operators, such as '$\land$', '$\parallel$', ' $[]$ ' and '$\fatsemi$' (including the recursive operations) are formally defined in [24, 25].

[7]The semantics foundations for object identity[14] has been developed in [16], where the work is related to the Z base standard project [3]. Axiomatic semantics of Object-Z has been developed in [25] by extending the logic $\mathcal{W}$[29].

5 Conclusion

In this paper, we have used the Object-Z notation to achieve an object-oriented approach to the specification of the denotational semantics of a simple object-oriented programming language. The key idea of this approach is to model the language constructs of OOPLs using an object-oriented view. The advantages of applying such an approach to model a programming language is that the abstract syntax and static and dynamic semantics for each component of a programming language structure are grouped together and captured in a class construct; hence the language model is concise and easily extended.

As the core features of OOPLs have been modelled in this paper, we believe that it is feasible to extend this model to capture fully the denotational semantics of commercial object-oriented programming languages, such as Eiffel and C++. We have also successfully applied this approach to specify the denotational semantics of block-structured sequential programming languages[7, 10]. Whether the approach is suitable for modelling the semantics of other programming language paradigms, such as declarative and parallel programming languages, remains to be explored.

This paper also demonstrates the expressiveness of the Object-Z notation when it is used to specify the semantics of a programming language. Various object-oriented concepts in object-oriented programming are precisely captured by the various features (e.g. the object containment notion) of the Object-Z notation. For instance, the acyclic structure of class inheritance relationship is precisely specified by the object containment notation '$_{\circledS}$'. Furthermore, this semantic model of SOPL captures various object-oriented concepts. For example, when a class is modelled, the specification precisely captures the combined ideas that a class is a template, a collection and a factory of objects. This paper in effect does more than define the semantics of OOPLs; it also presents an object-oriented approach to the formalisation of object-oriented concepts themselves.

Acknowledgements

The authors would like to thank Steven Atkinson, Graeme Smith, Andrew Hussey and Anthony MacDonald for their useful comments and suggestions.

References

[1] P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83(2):152–

205, 1989.

[2] D. Bjørner and C. B. Jones. *Formal Specification and Software Development.* International Series in Computer Science. Prentice-Hall, 1982.

[3] S. Brien, T. King, J. Nicholls, J. Woodcock, and J. Wordsworth. Z Base Standard — Version 1.0. Technical report, Programming Research Group, Oxford Univ., Nov 1992.

[4] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, International Symposium (LNCS 173)*, pages 51–67. Springer-Verlag, 1984.

[5] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 433–443, 1989.

[6] J. S. Dong. Living with Free Type and Class Union. To appear in *The 1995 Asia-Pacific Software Engineering Conference (APSEC'95) Brisbane, December 1995.*

[7] J. S. Dong. *Formal Object Modelling Techniques and Denotational Semantics Studies.* PhD thesis, The University of Queensland, Oct 1995. Submitted.

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

[9] J. S. Dong and R. Duke. The Geometry of Object Containment. *Object-Oriented Systems*, 2(1):41–63, Chapman & Hall, March 1995.

[10] J. S. Dong, R. Duke, and G. Rose. An Object-Oriented Approach to the Semantics of Programming Languages. *Australian Computer Science Communications*, 16(1):767–775, January 1994.

[11] J. S. Dong, G. Rose, and R. Duke. The Role of Secondary Attributes in Formal Object Modelling. To appear in *The IEEE International Conference on Engineering Complex Computer Systems (ICECCS'95)* Florida, November 1995.

[12] D. Duke and R. Duke. Towards a semantics for Object-Z. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM'90: VDM and Z!*, volume 428 of *Lect. Notes in Comput. Sci.*, pages 242–262. Springer-Verlag, 1990.

[13] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 5*, pages 465–483. Prentice-Hall, 1991.

[14] R. Duke and G. Rose. Modelling object identity. In *Proc. 16th Australian Comput. Sci. Conf. (ACSC-16)*, pages 93–100, February 1993.

[15] R. Duke, G. Rose, and G. Smith. Object-Z: a Specification Language Advocated for the Description of Standards. To appear in a special issue of *Computer Standards and Interfaces* on Formal Methods and Standards, September 1995.

[16] A. Griffiths and G. Rose. A Semantic Foundation for Object Identity in Formal Specification. To appear in the *Object-Oriented Systems (OOS)*, 1995.

[17] Andreas V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, pages 548–568. Springer-Verlag (*LNCS* 526), 1991.

[18] C. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice-Hall, 1986.

[19] Samuel Kamin. Inheritance in Smalltalk–80: A denotational definition. In *Fifteenth Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[20] P. D. Mosses. Denotional semantics. In *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 577–631. MIT Press, 1990.

[21] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1994.

[22] Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[23] 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.

[24] G. Smith. A Logic for Object-Z (Additional Rules). Technical Report 95-26, Software Verification Research Centre, Dept. of Computer Science, Univ. of Queensland, Australia, 1995.

[25] G. Smith. Extending W for Object-Z. In J. P. Bowen and M. G. Hinchey, editors, *Proceedings of the 9th Annual Z-User Meeting*, pages 276–295. Springer-Verlag, September 1995.

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

[27] S. Stepney. *High integrity compilation: A case study.* Prentice-Hall, 1993.

[28] Mario Wolczko. Semantics of Smalltalk-80. In *Proc. European Conference on Object-Oriented Programming*, pages 108–120. Springer-Verlag (*LNCS* 276), 1987.

[29] J. C. P. Woodcock and S. M. Brien. W: A logic for Z. In J. E. Nicholls, editor, *the Sixth Annual Z User Meeting, York, UK.*, Workshops in Computing, pages 77–96. Springer-Verlag, 1992.