Reasoning and Refinement in Object-Oriented Specification Languages

K. Lano, H. Haughton

Lloyds Register of Shipping, 29 Wellesley Rd., Croydon

Abstract

This paper describes a formal object-oriented specification language, Z^{++}, and identifies proof rules and associated specification structuring and development styles for the facilitation of validation and verification of implementations against specifications in this language. We give inference rules for showing that certain forms of inheritance lead to refinement, and for showing that refinements are preserved by constructs such as *promotion* of an operation from a supplier class to a client class. Extension of these rules to other languages is also discussed.

1 Introduction

Formal specification languages such as Z have been primarily used to *describe* systems, and hence to aid in informal validation of specifications against requirements, and in system design, but not to formally *develop* implementations from specifications, with mathematical justifications for the steps taken. Formal specification languages however provide the *possibility* of formal verification, a precise document against which implementations can be proved correct, if anyone had the tools or dedication to do this proof. Until Spivey's semantics for Z [29], not even this benefit was available through the use of Z.

In response to the recognition of this deficiency, several approaches to supporting proof and refinement in Z or Z-like languages have been in development: the B-tool and the ZedB-tool [26, 28]; the refinement editor, based on Carroll Morgans refinement calculus [24]; and natural-deduction style proof systems [32, 7, 19]. The schema operators of Z have been often identified as a major cause of problems in constructing such systems. Both the B-tool and the refinement calculus dispose of this notation, and the ZedB-tool works by expanding schema expressions built by use of $\wedge$, $\vee$ etc into their explicit forms.

In this paper we will examine other forms of structuring, using *objects* and *classes*. We define proof rules that can be used with classes and the higher-level combination operations on classes, and describe specification styles which can assist in proof and refinement. We can express within the specification language relationships of refinement and inheritance between specifications, and hence, can provide a formal documentation for development, to assist in future maintenance and reuse [17].

It has been claimed that object-oriented design and specification support a rapid prototyping methodology, in which reuse of previously developed classes features heavily. This is valid to the extent that a class serves as a 'black box' which insulates its internal workings from the outside environment, and whose behaviour is specified by *axioms*. In this case knowledge about the internal implementation of the methods of a class is not needed to prove properties about the functionality of the methods:

reverse-engineering or program comprehension need not descend into the code of the class.

The algebraic constraints and the classification structures that can be expressed by object-oriented design may be of assistance in selecting components, but the relationships of refinement and inheritance need to be clearly distinguished: when component D is a refinement of component C, with a documented refinement f, we can use D in place of C with a guarantee of correct behaviour. Forms of inheritance which are not refinements cannot give this guarantee. In addition, it seems to be often the case that the 'units of reuse' for object-oriented systems are *modifications* and enhancements of classes, rather than classes themselves [31]. Hence we have emphasised proving properties about *changes* in specifications in this paper.

In section 2 we will give the basic definitions of refinement and inheritance that we will use, in section 3 we describe rules for showing that some inheritances are refinements, and for constructing other simple forms of refinement; in section 4 we describe laws associated with our co-product construction, and compare this to that adopted by other object-oriented languages. In section 5 we relate our concept of refinement to software development, and provide general rules for constructing refinements of systems from refinements of components. In section 6 we relate refinement to the concepts of refinement used by Z, CSP and Object-Z, and in section 7 we summarise our refinement rules and provide an example of their application.

2 Terminology

We give the definitions that we will use in the rest of the paper. The syntax and the basic functions *types_of_class*, *signature*, etc, of the semantics of Z^{++} are given in [15, 14], and are summarised in the appendix. The invariant *invariant*(C) of a class C will also be denoted by Inv_C.

Definition A *formal language* $\mathcal{L}$ is a tuple $(\mathcal{R}, \mathcal{O}, L, v)$ of relation symbols (of various arities), operation symbols (including constants), logical connectives, and variables. A set $Exp_{\mathcal{L}}$ of valid expressions of the language is defined recursively [23].

Definition A *language morphism* ϕ between formal languages $\mathcal{L}$ and $\mathcal{L}'$, is a map from the non-logical symbols of $\mathcal{L}$ to the non-logical symbols and terms of $\mathcal{L}'$, which extends in the usual way to a function mapping expressions to expressions and formulae to formulae, such that logical connectives are preserved: $\phi(A \wedge B) = \phi(A) \wedge \phi(B)$ and so forth.

Definition *Strict inheritance* of a class C in a class D is defined as follows. Let E be the class defined as D but without C mentioned in its `EXTENDS` list. Then rename attributes of C until the set of names used for attributes, methods and types defined in C are disjoint from those defined in E, to form a class C'. The class D has attributes and methods the union of those of C' and E, algebraic constraints the conjunction of those of C' and E, and invariant the conjunction of those of C' and E. A formal definition is in [14].

Definition *Inheritance* of C in D is defined as above, but without attribute renaming taking place. It can be expressed via strict inheritance and *strengthenings* of a class. It is denoted by $C < D$.

Definition A *strengthening* of a class C is a class C' with attributes, methods and

types as defined in C, but with a logically stronger invariant.

Definition If C is a class, $\pi(C)$ denotes the cross-product of its attribute types. This corresponds to the *state space* of a Z specification, and we will refer to the *state* of C to mean the tuple of attributes of C which ranges over $\pi(C)$.

Definition The (formal) *language* of a class C has constants T, v_i, for each type name and attribute name defined in C, and function symbols of appropriate arities for each method name m_j defined in C. Local constants are subsumed under the attributes. Also, we include the function and predicate symbols of *Zermelo* set theory as used within Z [30]. We view a class as defining a specification, that is, a theory in an appropriate formal language, which must include all the symbols defined within the class. The axioms $\underline{\Gamma}\ C$ of a class C consist of the *Zermelo* axioms, modified for Z [30], together with axioms from each class that is used as a type in C, and axioms which state that the attributes satisfy the invariant of the class, and that the methods transform any admissible state in the way given by their definitions. Each algebraic constraint e gives rise to an axiom $algtrans(e, C)$ in terms of the attributes and method definitions of the class. A precise definition is in [14]. The use of axioms to specify a class behaviour is also a feature of the Fresco language of [31].

A definition of refinement in terms of *theory morphisms* is given in [14]. A more practical definition of refinement of C by D is given by the following conditions on a pair (ϕ, f):

Definition Let class D have attributes $w : W$, invariant Inv_D, operations m with preconditions $Pre_{D,m} : W \leftrightarrow X$, and definitions $Op_{D,m} : W \times X \leftrightarrow W \times Y$, with a similar convention being followed in class C: $Pre_{C,m} : T \leftrightarrow X$ is the precondition of the method m, $x : X$ and $y : Y$ its input and output parameters, and $Op_{C,m} : T \times X \leftrightarrow T \times Y$ its definition; T is the cross product $\pi(C)$ of the type identifiers of the signature of C. For simplicity, we are considering that the **OWNS** list of a class consists of a single variable declaration $w : W$, with multiple attribute types treated by defining W as $\pi(D)$. The pair (ϕ, f), where ϕ is a language morphism from the language of C to that of D, restricted so that each method name of C is mapped to a method name of D, every attribute name of C is mapped to an expression built only of attribute names of D, every type name is mapped to an expression built only of type names, and for each constant c of C, $\phi(c)$ is an expression built only out of constant names of D, and f, the *underlying function*, is a map from $\{w : \pi(D) \mid Inv_D(w)\}$ to $\{v : \phi(\pi(C)) \mid \phi(Inv_C)(v)\}$ under $\underline{\Gamma}\ D$, is a *refinement* if the conditions 1 - 6 hold under $\underline{\Gamma}\ D$:

1. $\forall\, w : W \bullet Inv_D(w) \Rightarrow w \in dom\, f \wedge \phi(Inv_C)(f(w))$;
2. $\forall\, w : W;\ x : X \bullet \phi(Pre_{C,m})(f(w), x) \Rightarrow Pre_{D,\phi(m)}(w, x)$;
3. $\forall\, w, w' : W;\ x : X;\ y : Y \bullet \phi(Pre_{C,m})(f(w), x) \wedge Op_{D,\phi(m)}(w, x, w', y) \Rightarrow \phi(Op_{C,m})(f(w), x, f(w'), y)$;
4. $\forall\, w : W \bullet f(w) \in \phi(T)$;
5. $\forall\, a : t \bullet \phi(\theta[\overrightarrow{a.m}/\vec{m}\ \overrightarrow{a.v}/\vec{v}])$ for each $\theta \in \underline{\Gamma}\ t$, where $\vec{m}$, $\vec{v}$ are the methods and attributes of t;

6. $\phi(algtrans(\underline{algebraic\_constraints}(C), C))$;

Where 2 and 3 hold for each method m of C, 5 holds for each class t that is used as a type in C, and $\theta[a/v]$ denotes the substitution of v by a in θ.

We denote that D is a refinement of C via (ϕ, f) by: $C \sqsubseteq_{(\phi,f)} D$ or $C \sqsubseteq_{\phi} D$. Normally ϕ will be taken as the identity on the basic set-theoretic operations such as $\#$ or $\cup$, and $\phi(\pi(C))$ as $\pi(C)$. In addition, $\phi(v)$ will be normally syntactically identical to the defining expression of $f(w)$.

Conditions 1 and 4 imply that any element of the state space of D can be translated into an element of the state space of (the interpretation in D) of C. Condition 2 gives the usual condition that the precondition of the interpretation $\phi(m)$ of each method m of C is less restrictive than the precondition of m; condition 3 says that $\phi(m)$ is more deterministic than m when both are defined - again, on the translated state space of C. Condition 5 means that all properties of supplier classes of C still hold under translation in D. Condition 6 means that the translation of the algebraic constraints of C are valid in D.

The conditions 2 and 3 correspond to the conditions of [6] on *EXTENSION* of a class, except that, for simplicity, we are not considering mappings on input types X of methods. We require a more general expression of the conditions since we want $\underline{\Gamma}\ D \vdash \phi(\psi)$ for each $\psi \in \underline{\Gamma}\ C$. At the level of basic types such as integers we also take a more general view, considering that D is a refinement of C if there is a surjective map $f : D \twoheadrightarrow C$, ie, if every member of D can be seen as a member of C.

These definitions capture the idea that D can implement each of the operations of C, in the sense that the interpretation $\phi(m)$ of each method m is a procedural refinement of m relative to the data refinement (or 'abstraction morphism') represented by f. The definition corresponds to the concept of relative interpretation used in set theory, as, for example, in the definition of the inner model L of constructible sets [3].

As a simple example of the refinement concept, consider the transformation of a class $Set[X]$ to a class $Seq[X]$:

```
CLASS Set[X]                          CLASS Seq[X]
OWNS                                  OWNS
 s : ℙ X                                 t : seq X
OPERATIONS                            OPERATIONS
 Add : X →                               Cat : X →
ACTIONS                               ACTIONS
 Add x ==> s' = s ∪ {x}                  Cat x ==> t' = t ⌢ ⟨ x ⟩
END CLASS                             END CLASS
```

via the refinement ϕ defined by:

$$s \mapsto ran\ t$$
$$X \mapsto X,\ \cup \mapsto \cup,\ \mathbb{P} \mapsto \mathbb{P}$$
$$Add \mapsto Cat$$

$f(t) = ran\ t$ also.

Instances of $Seq[X]$ can be regarded as instances of $Set[X]$ for functional purposes, and the same would apply if more complex corresponding operations were included, such as *Remove* or *Member*.

If D is a refinement of C via a language morphism ϕ then the type of D is a subtype of that of C, via a translation operation derived from ϕ:

$$\begin{array}{l} ^{\phi} : \underline{instances}(D) \rightarrow \underline{instances}(C) \\ \quad a \mapsto a^{\phi} \end{array}$$

The assignment $a' = b^{\phi}$, where $a' : C$; $b : D$, is valid in this case.

Definition There are operations of co-product $\underline{+}$ and product $\underline{\times}$ on classes, which satisfy the co-product and product axioms of a category when refinement is used as the categorical arrow in the category whose elements are all classes and types. $C \underline{+} D$ is the 'least common refinement' of C and D, and $C \underline{\times} D$ is the 'greatest common abstraction' of C and D. These can be syntactically defined: $C \underline{+} D$ is defined to be the class which is the same as D but with C added to its **EXTENDS** list, that is, the mutual strict inheritance of both in the empty class. $C \underline{\times} D$ is based on the *union* of C and D as in [8].

Definition A Z^{++} specification S is a sequence of Z^{++} paragraphs, where a paragraph is either a class definition or a Z paragraph, in which class names can be used as type names. Scope for class names follows the same conventions as for schema names in Z.

We will concentrate on the semantics of classes in the following. A specification defines a directed acyclic graph of classes in terms of the inheritance relationship $C \ll D$ "C occurs in D's **EXTENDS** list". We denote by $C <_i D$ the transitive closure of the relation $\ll$. Similarly, we do not allow cycles in the client relation (D is a client of C if C is used as a type in the definition of D): such cycles are unnecessary since we can use the free type constructions of Z to build recursive data-structures such as trees.

As in Eiffel [22], the concepts of class and type are equivalent in Z^{++}. We do not have a concept of 'object identity' (Object-Z does not have this concept either), however semantic behaviour of objects equivalent to that of objects with identities can be specified.

3 Inheritance and Refinement

In general, the syntactic operation of inheritance will not imply the semantic relation of refinement. Conversely, it is also possible for refinements to exist between classes which are not related in the inheritance hierarchy of a specification. Thus, during development, care should be taken to distinguish between inheritances which are refinements, and those which are not, and to document those refinements which do exist.

3.1 Specialisation

A class D is a *specialisation* of a class C if it is a refinement of a strengthening of C:

$$C \underline{\leq} D \equiv \exists C'(C' \sqsubseteq D \wedge C' \textit{ strengthens } C)$$

Specialisations are useful for expressing similarities between object classes, or classification hierarchies (eg: that the class of 2-3 trees is a particular specialisation of the class of trees). They correspond to *DERIVED TYPEs* in [6] - methods are restricted to act on the new restricted sub-state, and hence, are only *relative* refinements (relative to the state space of the specialised class), and not absolute refinements. An instance of such a class cannot be used in place of an instance of the class it specialises, since its methods may be called upon to act on inputs and states for which their behaviour is not consistent with the method of the original class. However, specialisations often occur in practical development.

The following rules are used for determining when a general inheritance is a refinement. In each case the refinement is based on the identity theory morphism, which maps each symbol of the language of C to the corresponding symbol of the language of D.

$$(I1): \quad \frac{\underline{\textit{invariant}}(D) \Leftrightarrow \underline{\textit{invariant}}(C) \quad \underline{\textit{algebraic\_constraints}}(C) \Leftrightarrow \underline{\textit{algebraic\_constraints}}(D) \quad C < D}{C \sqsubseteq_{id} D}$$

If the invariant of D adds no further conditions to that of C, then the inheritance is a refinement. This holds because we consider that all of the previously defined versions of a method are retained in the new class D, in renamed form (referred to by the notation $C.m$), even if D redefines them.

$$(I2): \quad \frac{C < D \quad \underline{\textit{algebraic\_constraints}}(C) \Leftrightarrow \underline{\textit{algebraic\_constraints}}(D) \quad \underline{\textit{invariant}}(C) \Rightarrow \exists u : U \bullet \underline{\textit{invariant}}(D)}{C \sqsubseteq_{(id,f)} D}$$

Where E is the class D with C removed from its **EXTENDS** list, $u : U$ is the declaration of E, $v : V$ of C, and the attribute names in u and v are disjoint. Intuitively, this means that the invariant of D, which can relate the states of E and C, does not constrain the inherited state of C. An example would be the introduction of new attributes in E which are purely auxilliary functions of the attributes of C. This rule holds if we define the abstraction function f as follows:

$$f : U \times V \nrightarrow V$$
$$(u, v) \longmapsto v \quad \text{if } Inv_D(u)$$

$W \nrightarrow V$ denotes the space of partial functions from W to V.

In particular, any strict inheritance is a refinement.

3.2 Simple Refinements

A set of basic but useful transformations can be used to modify and adapt classes, either for reasons of efficiency or clarity, and we list some of these here.

Finite Differencing A RETURNS operation r, which computes a complex expression $E(\vec{v})$ of the attributes of a class C, can be replaced by a new variable v_{n+1}, an operation *ret* that returns the value of v_{n+1}, and by the provision of additional functionality for each method which changes the state, to maintain the invariant $E(\vec{v}) = v_{n+1}$. This is a refinement since the invariant Inv_D of the modified class D is $Inv_C(\vec{v}) \wedge v_{n+1} = E(\vec{v})$, and we can define the data refinement to be:

$$\begin{array}{l} f : T \times T_{n+1} \rightarrow T \\ \quad (\vec{v}, v_{n+1}) \mapsto \vec{v} \end{array}$$

with $dom\, f = \{\vec{v} : T;\ v_{n+1} : T_{n+1} \mid Inv_C(\vec{v}) \wedge v_{n+1} = E(\vec{v})\}$. Algebraic constraints are preserved if we map r to *ret*.

1-1 Renaming A one to one renaming of the attributes and methods of a class is a refinement, via the language morphism that performs this re-naming. This means that we can always assume that two classes have disjoint sets of names for their attributes and methods when we need to form their co-product.

Replacing an Output Variable by a New Attribute If class C contains a method $op\ x\ y\ y_{r+1}\ ==>\ \psi(v, x, v', (y, y_{r+1}))$ then we can transform C into a class with a new attribute variable $v_{n+1} : Y_{r+1}$ where Y_{r+1} is the type of the output parameter y_{r+1} of op, and in which the method op is replaced by the method $op'\ x\ y\ ==>\ \psi'((v, v_{n+1}), x, (v', y_{r+1}), y)$ where $\psi'((v, v_{n+1}), x, (v', y_{r+1}), y) \equiv \psi(v, x, v', (y, y_{r+1}))$. We also include an operation *ret* that simply returns the value of v_{n+1}, and a method op'' which applies op' and then *ret*. The resulting class is a refinement, via the projection abstraction morphism. Algebraic constraints are preserved if we map op to op''.

4 Laws of Products and Co-Products

The following laws provide methods for building refinements from other refinements, and hence, for reducing the work of discovering new refinements. The significant properties of our definitions of product and co-product of classes are:

$$\begin{array}{ll} (CP1): & A \sqsubseteq_f D \wedge B \sqsubseteq_g D \ \Rightarrow\ (A \mathbin{+} B) \sqsubseteq_{f \mathbin{+} g} D \\ (CP2): & A \sqsubseteq_{\rho_1} A \mathbin{+} B \ \wedge\ B \sqsubseteq_{\rho_2} A \mathbin{+} B \end{array}$$

where $f \mathbin{+} g$ can be defined from f and g, and ρ_1, ρ_2 are the canonical embeddings;

$$\begin{array}{ll} (P1): & D \sqsubseteq_f A \ \wedge\ D \sqsubseteq_g B \ \Rightarrow\ D \sqsubseteq_{f \times g} (A \times B) \\ (P2): & (A \times B) \sqsubseteq_{\pi_1} A \ \wedge\ (A \times B) \sqsubseteq_{\pi_2} B \end{array}$$

These properties are not shared by the definitions of multiple inheritance taken by Object-Z [8] or other versions of object-oriented Z, which use conjunction of schemas, as in Z, and which merge variables of the same name. For instance, if we define the classes $C1$ and $C2$:

```
CLASS C1                    CLASS C2
OWNS                        OWNS
  v : ℕ                       v : ℕ
INVARIANT                   INVARIANT
  v ≥ 10                      v < 10
END CLASS                   END CLASS
```

The co-product of $C1$ and $C2$ defined using multiple inheritance under the MooZ [21] or Object-Z definition would be a class with *false* invariant. But these classes are both refined by the class:

```
CLASS C3
OWNS
  v1 : ℕ;
  v2 : ℕ
INVARIANT   v1 ≥ 10 ∧ v2 < 10
END CLASS
```

($C3 = C1 \pm C2$ under our definitions [10, 14]), and there is *no* refinement from the *false* class into this class: all possible implementations have been eliminated by taking a more refined than necessary definition of multiple inheritance.

The laws $(CP1)$ and $(CP2)$ lead to the useful law that strict inheritance preserves refinement:

$$(I3): \quad \frac{A \sqsubseteq A' \quad C = A \pm B \quad C' = A' \pm B}{C \sqsubseteq C'}$$

5 The Preservation of Refinement

A development history can be viewed as a sequence $\langle s_1, s_2, \ldots \rangle$ of specifications, terminating in an implementation s_n, and with each s_{i+1} being a transformation of s_i. Ideally these transformations would be refinements or specialisations.

Within software maintenance tasks can similarly be classified into several types of transformation [18], such as *corrective*, *adaptive* and *perfective*. Adaptive and perfective maintenance tasks typically involve a user request for a refinement of the implementation s_n (that it should be able to perform more operations, and be able to deal with a wider range of data). Thus we aim to refine a specification s_i at some stage $i < n$, and to translate this refinement automatically into refinements of later stages. Modification of an abstract specification is easier than modification of the imperative code of s_n. In general, the higher the level of abstraction, the fewer details need to be considered, although we cannot go beyond levels at which it is possible to express the required change.

Preservation of refinement by the transformations which constitute the development steps $s_j \longrightarrow s_{j+1}$ for $j \geq i$ is therefore essential. Similarly, we will require that

generic classes preserve refinements of their parameters, in order that specifications constructed from components by such classes are 'improved' by an improvement of their components. This is a highly desirable property in a wide class of situations involving composite systems.

5.1 Specific laws of Refinement Preservation

Some rules on type and operation constructions which imply that they preserve relational refinement $\sqsubseteq$ of their components are the following, expressed in a natural deduction format.

$$(R1): \quad \frac{f, f' : A \nrightarrow B \quad g, g' : B \nrightarrow C \quad f \sqsubseteq f' \quad g \sqsubseteq g'}{f ; g \sqsubseteq f' ; g'}$$

$$(R2): \quad \frac{R, R' : A \leftrightarrow B \quad S, S' : A \leftrightarrow B \quad R \sqsubseteq R' \quad S \sqsubseteq S' \quad dom\, R' \cap dom\, S' = \emptyset}{R \cup S \sqsubseteq R' \cup S'}$$

$$(R3): \quad \frac{R, R' : A \leftrightarrow B \quad S, S' : C \leftrightarrow D \quad R \sqsubseteq R' \quad S \sqsubseteq S'}{R \times S \sqsubseteq R' \times S'}$$

From $R3$ and $R2$ we can deduce that the disjoint union constructor: $(R, S) \mapsto \{0\} \times R \cup \{1\} \times S$ preserves refinement. However, in practice we will often have to use constructions which preserve refinement only under a limited range of circumstances, such as relational composition $\fatsemi$, or intersection.

$(R2)$ allows us to conclude that a method defined using disjoint union of two other methods is refined by the refinement of these methods. This is a common situation in Z specifications. Similarly, $(R3)$ ensures that if we expand the state of a class, and place no new restriction on the old state variables, and extend each method $C.m$ of the old class by a (satisfiable) new conjunct $m_{new}(u)$ entirely in the new variables u, then this new method $D.m$ is a refinement of the old.

5.2 Syntax Directed Construction of Refinements

Ideally, if we have constructed a data refinement $f : W \rightarrow V$, we should be able to use this function in building data refinements between types built from W and V: $con[f] : con[W] \rightarrow con[V]$ for a range of constructors con. In the following, we give a set of rules for constructing 'natural' data refinements; these constructions are 'natural' in the sense of being functors. This is an important property, since it implies that they preserve sequential (functional and relational) composition: $con[f \fatsemi g] = con[f] \fatsemi con[g]$ and identity: $con[id_A] = id_{con[A]}$; often they will also preserve co-products and products.

$$\frac{(\textit{Power set } 1):\ f : W \to V}{\mathbb{P}[f] : \mathbb{P}\ W \to \mathbb{P}\ V \qquad a \longmapsto f\ (\!|a|\!)} \qquad \frac{(\textit{Power set } 2):\ R : W \leftrightarrow V}{\mathbb{P}_2[R] : \mathbb{P}\ W \leftrightarrow \mathbb{P}\ V \qquad a \longmapsto b \quad \text{where } b \subseteq R\ (\!|a|\!)}$$

$$\frac{(\textit{Cartesian product}):\ f : W \to V}{(\times X)[f] : W \times X \to V \times X \qquad (w, x) \longmapsto (f(w), x)} \qquad \frac{(\textit{Function lift}):\ f : W \to V}{(X \nrightarrow)[f] : (X \nrightarrow W) \to (X \nrightarrow V) \qquad t \longmapsto (x \mapsto f(t(x)))}$$

These are functors, and similarly for the product on the right. The notation $T \leftrightarrow S$ specifies the set of relations between T and S.

These constructors satisfy other useful properties:

$$dom(f \times X) = (dom\, f) \times X \qquad dom\ \mathbb{P}[f] = \mathbb{P}(dom\, f)$$
$$ran(f \times X) = (ran\, f) \times X \qquad ran\ \mathbb{P}_2[f] = \mathbb{P}(ran\, f)$$
$$dom\ \mathbb{P}_2[f] = \mathbb{P}(dom\, f)$$

The above lead to the following rules about generic classes:

$$(G1):\ \frac{\begin{array}{l}\text{No methods of } X \text{ are used in } G \\ X \text{ only occurs in the declaration part of } G \text{ as a type } con[X]\end{array}}{A \sqsubseteq_{(\phi,f)} B \ \Rightarrow\ G[A] \sqsubseteq_{G[(\phi,f)]} G[B]}$$

where con is a refinement-preserving constructor, and X is the generic parameter of $G[X]$. This is valid since the refinement ϕ produces a refinement $con[\phi]$ from the type represented by $con[A]$ to the type represented by $con[B]$ (the underlying function $f : \pi(B) \nrightarrow \pi(A)$ lifts to a function $g : con[\pi(B)] \nrightarrow con[\pi(A)]$). $G[\phi]$ maps the attributes of $G[A]$, except those of form $v_i : con_i[X]$, to themselves. Each attribute $v_i : con_i[X]$ is mapped to $g_i(v_i)$.

We will seek to extend the above result to cases in which the methods of X are used in contexts which preserve their refinement. We denote the fact that the methods of D preserves refinement of a named set S of methods by: $\underline{methods}(D)\ \underline{valid\_usages}\ S$.

$$(G2):\ \frac{\begin{array}{l}\underline{methods}(G)\ \underline{valid\_usages}\ S \\ S \subseteq \underline{methods}(A) \cap \underline{methods}(B) \\ X \text{ only occurs in the declaration part of } G \text{ as a type } con[X] \\ \underline{\Gamma}\ G[B] \vdash \phi(algtrans(\underline{algebraic\_constraints}(G[A]), A))\end{array}}{A \sqsubseteq_{\phi} B \ \Rightarrow\ G[A] \sqsubseteq_{G[\phi]} G[B]}$$

where S is the set of methods used from the parameter class within G, other restrictions are as above. From $(R2)$ we know in particular that a global method of the form

$$op\ x\ y \quad ==> \quad E(v, x) \wedge op_L(v, x, v', y) \vee \neg\, E(v, x) \wedge \psi(v, x, y, v')$$

preserves procedural refinements of op_L to op'_L.

($G3$): Another special case is where op_G uses op_L via a framing or promotion [19] construct:

$$op_G\ x1\ y1\ ==>\ U(v, x1, v_L, x) \wedge op_L(v_L, x, v'_L, y) \wedge E(v'_L, x1, v, y, v', y1)$$

in which we regard the global state $v : con[L]$ as being built from the local state L via a refinement-preserving constructor, $U : con[L] \times X1 \rightarrow (L \times X)$ selects a local state from the global state, and $E : L \times X1 \times con[L] \times Y \rightarrow (con[L] \times Y1)$ assembles a new global state and result from the new local state and old global state, and the local result. For general data-refinements $f : W \rightarrow V$, we have that $con[f] : con[W] \rightarrow con[V]$ preserves function composition, and can be lifted to a data-refinement, provided that

$$\begin{array}{l} U(con[f](v), x1, f(v_L), x) \equiv U(v, x1, v_L, x) \\ E(f(v'_L), x1, con[f](v), y, con[f](v'), y1) \equiv E(v'_L, x1, v, y, v', y1) \\ op_{L2}(v_L, x, v'_L, y) \wedge op_{L2}(v_L, x, v''_L, y') \wedge \\ \qquad E(v'_L, x1, v, y, v', y1) \Rightarrow E(v''_L, x1, v, y', v', y1) \end{array}$$

under $Inv_G(con[f](v))$, and $Inv_G(con[f](v)) \Rightarrow Inv_G(v)$. A specific example of an application of this rule is given later.

5.3 Algebraic Constraints

We have been assuming a particular interpretation ((ii) below) of algebraic constraints. However there are other reasonable definitions of the meaning of the statement $\forall a : C \bullet t_1(a) = t_2(a)$, where C is a class type, and t_1, t_2 are terms built from method symbols of C: (i) "All deterministic implementations (refinements) of the methods satisfy this equation"; (ii) "The methods (as relations) satisfy this equation"; (iii) "Any object or data resulting from $t_1(a)$ is observationally equivalent to any object or data resulting from $t_2(a)$". Observational equivalence for the terms of a class means that no sequence of further methods applied to the terms can distinguish them: in particular, the values of returnable functions applied to them are the same. (ii) is the most consistent with the conventional style of Z, in which schemas are relations, and abstract 'algebraic' constraints can be placed upon them by using the schema composition $\fatsemi$.

6 Relationship with Other Concepts of Refinement

Development of a calculus of refinement, based on weakest precondition semantics for programs and specifications (in a single wide-spectrum language), has been carried out by a number of groups [24, 2, 25]. Our notation for class operations corresponds quite well to the Z refinement calculus specification forms, and our semantic base is also similar [4].

In a class C, with global invariant $Inv_C(v)$ and declaration $v : T$, we can express an operation definition

$$op : X \rightarrow Y$$

$$op\ x\ y\ ==>\ \psi(w, x, w', y)$$

where w is a sublist of v, as the specification statement

$$op \mathrel{\widehat{=}} \quad w : [Inv_C(v) \wedge v \in T \wedge w_0 = w \ \wedge \\ \exists v' : T;\ y : Y \bullet Inv_C(v') \wedge \psi(v, x, v', y), \\ Inv_C(v) \wedge v \in T \wedge \psi(w_0, x, w, y)]$$

where the w_0 are a list of new variables.

The semantics of such a statement is given by its predicate transformer. This is:

$$\| (C.op\ x\ y) \| \phi \ = \ Inv_C(v) \wedge v \in T \wedge \\ \exists v' : T;\ y : Y \bullet Inv_C(v') \wedge \psi(v, x, v', y) \ \wedge \\ w_0 = w \wedge \forall w \bullet (Inv_C(v) \wedge v \in T \\ \wedge \psi(w_0, x, w, y) \Rightarrow \phi)$$

Refinement between two specifications is procedural refinement:

$$P \sqsubseteq P' \ \Leftrightarrow \ \forall \phi(\|P\|\phi \Rightarrow \|P'\|\phi)$$

which, viewing predicates as sets of result states: $\phi : \mathbf{P}(V \times Y)$, is equivalent to the concept of procedural refinement expressed via relations. Data refinement is viewed in the same framework. A general data refinement *rep* from P to P' is a predicate transformer which has

$$rep \circ P \sqsubseteq P' \circ rep$$

The restrictions that *rep* be monotonic and preserves arbitrary disjunctions are needed [24].

The *traces-failures* model of CSP [12] can also be related to our semantics, with each class C being assigned an alphabet $\alpha C = \bigcup\{m : \underline{methods}(C) \bullet \{m\} \times \underline{input\_type}(m)\}$ of all methods and possible inputs for these methods on C, and failures set F_C which consists of those (tr, X) such that $tr : seq\, \alpha C$, $X : \mathbf{P}\, \alpha C$, with a sequence $a : seq\, C$ of objects such that $a_1 = (m_1\ a_0)\ x_1$ some $a_0 : C$, $a_i = (m_i\ a_{i-1})\ x_i$ for each $i \leq \#tr$, $(m_i, x_i) = tr_i$. All elements $(m, x) \in X$ must have $\neg\ Pre_m(a_{\#tr}.v, x)$ in C.

Then, if D strengthens C, $dom\, F_D \subseteq dom\, F_C$ and $tr \in dom\, F_D \Rightarrow F_C(\!|tr|\!) \subseteq F_D(\!|tr|\!)$; if D strictly inherits C, then D is a traces-failures refinement of C, that is, $F_D \subseteq F_C$.

Similarly, we can relate our concept of refinement to that of [9] for Object-Z. Strict inheritance will imply *observational* compatibility: if $C \ll D$ then the behaviours (the possible sequences of operation applications) of C are a subset of the behaviours of D. Refinement of methods will imply, as in [9], *operational* compatibility.

7 A Refinement Calculus for Classes

We break down the steps required for proving refinements into applications of the following rules, thus reducing the burden of proof required from the developer. Our rules for class refinement are as follows:

- (1) Addition of a new attribute;
- (2) Addition of a set s of new attributes, invariants, and methods, such that rules ($I1$), ($I2$) or ($I3$) are valid;
- (3) Any 'simple refinement' listed in section 3.2 is a valid transformation;
- (4) Replacing the type of an attribute $v : V$ by a refinement W of V, provided that the restrictions of ($G1$), ($G2$) or ($G3$) hold. (We can view any type T as being a class type, simply by encapsulating the operations that are applied to its members into a class cT with an attribute *contents* : T).
- (5) Replacing a method m by a more refined method m', provided that the restrictions of ($R1$), ($R2$) or ($R3$) are true for the constructs which use m within the class;
- (6) Forming the co-product of the class with another class;
- (7) Adding additional algebraic constraints on the methods of a class.

7.1 Examples

We give an example of the use of these rules. If we have classes:

```
CLASS A
OWNS
 w : W
OPERATIONS
 Add : X →
ACTIONS
 Add ....
END CLASS
```

```
CLASS GA
OWNS
 u : Label ⇸ A
OPERATIONS
 AddG : Label X →
ACTIONS
 AddG l x  ==>  l ∈ dom u ∧
                u' = u ⊕ {l ↦ (Add u(l)) x}
END CLASS
```

Where [*Label*] is a set of labels, and GA may have other operations, we can apply the rule (4) above, with ($G3$), to establish that any refinement (ϕ, f) of A to a class B, of the form

```
CLASS B
OWNS v : V
OPERATIONS
 Add : X →
ACTIONS
 Add ....
END CLASS
```

produces a refined class if we substitute B for A in GA to give GB. We can express GA as $G[A]$ and GB as $G[B]$ by parameterising these classes, and the function lift

data refinement is taken: $con[f] : (Label \nrightarrow \pi(B)) \rightarrow (Label \nrightarrow \pi(A))$. G has no invariant, and the operations U and E in the $AddG$ method are:

$$\begin{aligned} U(u,(l,x),v_L,x) &\equiv v_L = u(l) \wedge l \in dom\, u \\ E(v'_L,(l,x),u,u') &\equiv u' = u \oplus \{l \mapsto v'_L\} \end{aligned}$$

and these satisfy the required conditions.

8 Conclusion

The results we have given support a transformational approach to object-oriented specification in the Z^{++} language. Other object-oriented Z extensions, such as Object-Z, share some common aspects with our language, such as the definition of class union or product, and our rules can be lifted to these languages in these cases. Practical software development, of a machine learning system [11], of a software maintenance toolset [16], and of database systems [15, 17], has been carried out using the language, and it has been used to represent large reverse-engineered data-processing applications as part of the REDO project [13, 5].

References

[1] A. Alencar, J. Goguen: *OOZE: An Object-Oriented Z Environment.* In: P. America (ed.): ECOOP '91 Proceedings, Springer-Verlag LNCS, Vol 512, 1991.

[2] R.J.R Back, *A Calculus of Refinements for Program Derivation*, Acta Informatica 25, 593-624 (1988).

[3] J.L. Bell, **Boolean-valued models and independence proofs in set theory**, Oxford Logic Guides 12, Clarendon Press, Oxford 1977.

[4] P. Breuer, K. Lano, *From Code to Z Specifications.* In: J. Nicholls (ed.): Z User Meeting 1989, in Springer-Verlag Workshops in Computer Science, 1990.

[5] P. Breuer, K. Lano, *Creating Specifications from Code: Reverse-Engineering Techniques*, Journal of Software Maintenance, September 1991.

[6] E. Cusack, *Object-Oriented Modelling in Z.* In: P. America (ed.): ECOOP '91 Proceedings, Springer-Verlag Lecture Notes in Computer Science, Vol 512, 1991.

[7] A. Diller, **Z: An Introduction to Formal Methods**, Wiley, 1991.

[8] D. Duke, R. Duke, P. King, G. A. Rose, G. Smith, *Object-Z: An Object-Oriented Extension to Z*, technical report 91-1, Software Verification Research Centre, The University of Queensland.

[9] R. Duke, P. King, G. Smith, *Formalising Behavioural Compatibility for Reactive Object-Oriented Systems.* In: P. America (ed.): ECOOP '91 Proceedings, Springer-Verlag Lecture Notes in Computer Science, Vol. 512, 1991.

[10] H. Haughton, K. Lano, *An Algebraic Semantics for the Specification Language Z^{++}*, AMAST '91 Conference, Iowa. To appear in Springer-Verlag Workshops in Computer Science.

[11] H. Haughton, K. Lano, *Using Formal Methods in Artificial Intelligence*, IJCAI '91 Workshop on Software Engineering for Knowledge-Based Systems, Sydney, August 1991.

[12] C.A.R. Hoare, **Communicating Sequential Processes**, Prentice Hall 1985.

[13] K. Lano, H. Haughton, P. Breuer, *Reverse-Engineering of Library Case Study*, REDO Document 2487-TN-PRG-1064, April 1991.

[14] K. Lano, H. Haughton, *Axioms for Object-Oriented Extensions to Z*, ZOOM Workshop, Oxford University Programming Research Group, 1991.

[15] K. Lano, *Z^{++}, an Object-Oriented Extension to Z*. In: J. Nicholls (ed.): Z User Workshop, Oxford 1990, Springer-Verlag Workshops in Computing, 151-172, 1991.

[16] K. Lano, *The Design of the Verification Toolset*, REDO Project Document 2487-TN-PRG-1068, Oxford University Programming Research Group, 1991.

[17] K. Lano, *Integrating Development and Maintenance in an Object-Oriented Environment*, REDO Document 2487-TN-PRG-1050, Oxford University Programming Research Group, 1991.

[18] S. Leonard, J. Pardoe, S. Wade, *Software Maintenance - Cinderella is Still not Getting to the Ball*. In: **BCS/IEE Conference on Software Engineering 1988**, IEE London, 104-106, 1988.

[19] P. Lupton, *Promoting Forward Simulation*. In: J. Nicholls (ed.): Z User Workshop, Oxford 1990, Springer-Verlag Workshops in Computing, 1991.

[20] J. A. McDermid, P. J. Whysall, *Object Oriented Specification and Refinement*. In: Proceedings of the 4th Refinement Workshop, Cambridge, Springer-Verlag Workshops in Computing, 1991.

[21] S. Meira, A.L.C. Cavalcanti, *Modular Object-oriented Z Specifications*. In: J. Nicholls (ed.): Z User Workshop, Oxford 1990, Springer-Verlag Workshops in Computing, 1991.

[22] B. Meyer, *Tools for the New Culture: Lessons from the Design of the Eiffel Libraries*, Communications of the ACM, September 1990, Vol 33, No. 9.

[23] D. Monk, **Mathematical Logic**, North Holland, 1979.

[24] C. Morgan, K. Robinson, P. Gardiner, *On The Refinement Calculus*, PRG Monograph PRG-70, 1988, Oxford University Programming Research Group.

[25] J. M. Morris, *A theoretical basis for stepwise refinement and the programming calculus*, Science of Computer Programming, 9(3), 298-306, December 1987.

[26] D. Neilson, *Machine Support for Z: the zedB tool*, Z User Meeting 1990. In: J. Nicholls (ed.): Z User Workshop, Oxford 1990, Springer-Verlag Workshops in Computing, 1991.

[27] C. Stanley-Smith, A. Cahill, *UNIFORM: A Language Geared To System Description and Transformation*, University of Limerick, 1990.

[28] I. Sørensen, *The B Method*, VDM '91, Noodwijkerhout, Netherlands, 1991.

[29] M. Spivey, **Understanding Z**, CUP, Cambridge, 1988.

[30] M. Spivey, **The Z Notation : A Reference Manual**, Prentice Hall 1989.

[31] A. Wills, *Capsules and Types in Fresco*. In: P. America (ed.): ECOOP '91 Proceedings, Springer Verlag Lecture Notes in Computer Science, Vol 512, 1991.

[32] J. Woodcock, S. Brien, *W: A Logic for Z*, Oxford University Programming Research Group, 1991.

9 Appendix: Syntax and Semantic Functions

Class Syntax

The BNF description of a Z^{++} class declaration is:

Object_Class ::= **CLASS** *Identifier TypeParameters* [**EXTENDS** *Imported*]
[**TYPES** *Types*] [**FUNCTIONS** *Axdefs*]
[**OWNS** *Locals*]
[**RETURNS** *Optypes*]
[**OPERATIONS** *Optypes*]
[**INVARIANT** *Predicate*]
[**ACTIONS** *Acts*]
[**CONSTRAINTS** *Constraints*]
END CLASS

TypeParameters ::= [*Parlist*]
| ϵ

Parlist ::= *Identifier* [, *Parlist*]
| *Identifier* $\ll$ *Identifier* [, *Parlist*]

Imported ::= *Idlist*

Types ::= *Type_Declarations*

Locals ::= *Identifier* : *Type* ; *Locals*
| *Identifier* : *Type*

Optypes ::= *Identifier* : *Idlist* $\rightarrow$ *Idlist* ; *Optypes*
| *Identifier* : *Idlist* $\rightarrow$ *Idlist*

$$
\begin{array}{lll}
\mathit{Acts} & ::= & [\mathit{Expression}\ \&]\ \mathit{Identifier}\ \mathit{Idlist}\ \texttt{==>}\ \mathit{Code}\ ;\ \mathit{Acts} \\
 & | & [\mathit{Expression}\ \&]\ \mathit{Identifier}\ \mathit{Idlist}\ \texttt{-->}\ \mathit{Code} \\
\mathit{Constraints} & ::= & \mathit{Equation} \\
 & | & \mathit{Equation}\ ;\ \mathit{Constraints}
\end{array}
$$

The *TypeParameters* are a list (possibly empty) of *generic* type parameters used in the class definition. A parameter X can be required to be a descendent of a class A via the notation $A \ll X$ here. The **EXTENDS** list is the set of previously defined classes that we are incorporating into this class. The *Types* are type declarations of type identifiers used in declarations of the local variables of the object. The *Local* variable declarations are attribute declarations, in the style of variable declarations in Z. The **OPERATIONS** list declares the types of the operations, as functions from a sequence of input domains to an output domain. The **RETURNS** list of operations defines the output type of those attributes and functions of the objects internal state that are externally visible; these are operations with no side-effect on the state. The **INVARIANT** gives a predicate that specifies the properties of the internal state, in terms of the local variables of the object. This predicate is guaranteed to be true of the state of an object class instance between executions of the operations of the object instance.

The **ACTIONS** list gives the definitions of the various operations that can be performed on instances of the object; for instance we would write: $READ\ x$ $\texttt{==>}\ \ q' = tail\ q\ \wedge\ x = head\ q$ in a specification of queues with contents q.

The input parameters are listed before the output parameters in the action definitions. *Code* includes Z predicates and procedural UNIFORM [27] code, both have a precise semantics as predicate transformers [4]. Alternative forms of notation, closer to Z or C++ are also available.

Semantic Functions on a Class

In the formal semantics for Z^{++} [14], we have represented a class as a particular type in Z. The type of classes is defined using the following syntactic components:

$$[\mathit{Ident}, \mathit{TypeDecs}, \mathit{Oplist}, \mathit{Actslist}]$$
$$\mathit{Idlist} == seq\ \mathit{Ident}$$
$$\mathit{Sig} == (\mathit{Ident} \nrightarrow \mathit{Ident})$$

$$
\begin{array}{lll}
\mathit{Exp} & ::= & \mathit{ident}\langle\!\langle \mathit{Ident} \rangle\!\rangle \\
 & | & \mathit{num}\langle\!\langle \mathbb{Z} \rangle\!\rangle \\
 & | & \mathit{fapp}\langle\!\langle \mathit{Ident} \times seq\ \mathit{Exp} \rangle\!\rangle \\
 & | & \mathit{frst}\langle\!\langle \mathit{Exp} \rangle\!\rangle \ \ | \ \ \mathit{scnd}\langle\!\langle \mathit{Exp} \rangle\!\rangle
\end{array}
$$

$$
\begin{aligned}
Pred ::= & \ \underline{true} \mid \underline{false} \\
& \mid eq\langle\langle Exp \times Exp \rangle\rangle \\
& \mid and\langle\langle Pred \times Pred \rangle\rangle \\
& \mid forall\langle\langle Sig \times Pred \rangle\rangle \\
& \mid exists\langle\langle Sig \times Pred \rangle\rangle \\
& \mid implies\langle\langle Pred \times Pred \rangle\rangle \\
& \mid in\langle\langle Exp \times Exp \rangle\rangle
\end{aligned}
$$

Sig is the type of class signatures, associations of variable names with type identifiers, these types having been defined elsewhere in the class or in inherited classes.

$$Parlist ::= \underline{constrained} \ \langle\langle \ Ident \times Ident \ \rangle\rangle \mid \underline{unconstrained} \ \langle\langle \ Ident \ \rangle\rangle$$

These correspond to the two types of generic parameterisation, constrained and unconstrained.

$$
\begin{aligned}
Extends &::= \underline{extends} \ \langle\langle \ Idlist \ \rangle\rangle \\
Types &::= \underline{types} \ \langle\langle \ TypeDecs \ \rangle\rangle \\
Owns &::= \underline{owns} \ \langle\langle \ Sig \ \rangle\rangle \\
Parameters &::= \underline{parameters} \ \langle\langle \ Parlist \ \rangle\rangle \\
Invariant &::= \underline{invariant} \ \langle\langle Exp \rangle\rangle \\
Optypes &::= \underline{operations} \ \langle\langle \ Oplist \ \rangle\rangle \\
OpActions &::= \underline{actions} \ \langle\langle Actslist \rangle\rangle \\
Constraints &::= \underline{constraints} \ \langle\langle Eqnlist \rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
Class &::= \emptyset \\
&\mid \underline{class} \ \langle\langle \ Ident \times Extends \times Types \times Owns \\
&\qquad \times Optypes \times Invariant \times OpActions \times Eqs \ \rangle\rangle \\
GenClass &::= \underline{generic} \ \langle\langle \ Class \times Parameters \rangle\rangle
\end{aligned}
$$

The type of class morphisms (refinements) is $[Ref]$, *Ref* and *Class* are linked by the basic axioms of a category [14]:

$$\_ \bigcirc \_ : Ref \times Ref \nrightarrow Ref$$

$$
\begin{aligned}
& Dom : Ref \rightarrow Class \\
& Ran : Ref \rightarrow Class \\
& id : Class \rightarrow Ref \\
\hline
& \forall f, g : Ref \bullet (\exists h : Ref \bullet f \bigcirc g = h) \Leftrightarrow (Dom \ g = Ran \ f) \\
& \forall f, g, h : Ref \bullet f \bigcirc (g \bigcirc h) = (f \bigcirc g) \bigcirc h \\
& \forall f : Ref \bullet f \bigcirc id \ (Ran \ f) = f \ \wedge \ id \ (Dom \ f) \bigcirc f = f
\end{aligned}
$$

In Z^{++} we write $C \sqsubseteq_f D$ to denote that $Dom \ f = C$, $Ran \ f = D$. There are types of class *instances* and *methods*:

$$
\begin{aligned}
& [Instances, Method] \\
& State == Ident \nrightarrow Exp
\end{aligned}
$$

Functions exist which return the components of a class and an instance:

$$
\begin{array}{l}
\underline{name\_of\_class} : Class \rightarrowtail Ident \\
\underline{extends\_list} : Class \rightarrow \mathbb{F}\ Ident \\
\underline{types\_of\_class} : Class \rightarrow State \\
\underline{signature} : Class \rightarrow Sig \\
\underline{instances} : Class \rightarrow \mathbb{F}\ Instances \\
\underline{class\_invariant} : Class \rightarrow Exp \\
\underline{methods} : Class \rightarrow \mathbb{F}\ Method \\
\underline{actslist} : Class \rightarrow Actslist \\
\underline{name\_of} : Instances \rightarrowtail Ident \\
\underline{parameter\_of1} : GenClass \rightarrow Ident \\
\underline{parameter\_of2} : GenClass \nrightarrow Ident \\
\underline{algebraic\_constraints} : Class \rightarrow \mathbb{F}\ Exp \\
\hline
(\forall\ C : Class;\ name : Ident;\ extends : Idlist; \\
\quad td : TypeDecs;\ locals : Sig;\ parameters : Parlist; \\
\quad inv : Exp;\ oplist : Oplist;\ acts : Actslist \mid \\
\qquad C = \underline{class}\ (name,\ \underline{extends}\ extends, \\
\qquad \underline{types}\ td,\ \underline{owns}\ locals,\ \underline{operations}\ oplist, \\
\qquad \underline{invariant}\ inv,\ \underline{actions}\ acts,\ \underline{constraints}\ cons) \bullet \\
\qquad\qquad \underline{name\_of\_class}(C) = name\ \wedge \\
\qquad\qquad \underline{types\_of\_class}(C) = defined\_in(td)\ \wedge \\
\qquad\qquad \underline{signature}(C) = locals\ \wedge \\
\qquad\qquad \underline{class\_invariant}(C) = inv\ \wedge \\
\qquad\qquad \underline{actslist}(C) = acts\ \wedge \\
\qquad\qquad \underline{algebraic\_constraints}(C) = cons\ \wedge \\
\qquad\qquad \underline{extends\_list}(C) = ran\ extends) \\
(\forall\ C : Class;\ i, j : Ident;\ G : GenClass \mid \\
\quad G = \underline{generic}\ (C, \underline{parameters}\ (\underline{constrained}\ (i, j))) \bullet \\
\qquad\qquad \underline{parameter\_of1}\ G = i\ \wedge\ \underline{parameter\_of2}\ G = j) \\
(\forall\ C : Class;\ i : Ident;\ G : GenClass \mid \\
\quad G = \underline{generic}\ (C, \underline{parameters}\ (\underline{unconstrained}\ i)) \bullet \\
\qquad\qquad \underline{parameter\_of1}\ G = i)
\end{array}
$$

Algebraic constraints are of the form of equations with implicit universal quantifiers over input parameters and objects. Define a *method term* for a class C to be either

1. id_C, the identity operation on C, or
2. $m\ x$ where m is an initialisation operation of C, and x is a list of terms of appropriate types for the inputs of m, or
3. $\tau\ ;\ (m\ x)$, where τ is a method term, and m is a method of C, and x is a list of terms of appropriate types for the inputs of m.
4. $\tau.1$, $\tau.2$, where τ is a method term.

An algebraic constraint is then an equation $\tau = \rho$ between two method terms of the same class, or an equation $\tau = c$ where c is an ordinary Z expression, not involving method terms. $\tau.1$ denotes the object returned by τ, and $\tau.2$ the tuple of output values returned by τ.

As an example:

$$id_{Stack} \ ; \ Push \ x \ ; \ Top \ = \ x$$
$$id_{Stack} \ ; \ Push \ x \ ; \ Pop \ = \ id_{Stack}$$

defines a *Stack* class. An equivalent notation using function calls to object variables is also available.