THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Refactoring Object-Oriented Specifications: A Process for Deriving Designs

Tim McComb
Graeme Smith

May 2006

Technical Report SSE-2006-01

Division of Systems and Software Engineering Research
School of Information Technology and Electrical Engineering
The University of Queensland
QLD, 4072, Australia

http://www.itee.uq.edu.au/~sse

Refactoring Object-Oriented Specifications: A Process for Deriving Designs

Tim McComb, Graeme Smith

School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, St. Lucia., QLD 4072, Australia.

Abstract

We describe a set of rules for the systematic structural modification of Object-Z specifications to derive designs that are appropriate for implementation in an object-oriented programming language. Such a methodology is important for systems that are sufficiently large to warrant the object-oriented programming approach, but additionally require development under strict verification conditions. We demonstrate that these rules are *complete* in the sense that from any specification that does not contain unbounded recursive constructs, any design may be derived which represents a refinement of the original and also does not contain unbounded recursive constructs.

1 Introduction

Object orientation is the dominant approach to the development of large-scale software systems and, indeed, software in general. While diagrammatic languages like UML [1] are often used to specify object-oriented software, when strict verification of the implementation is required a formal specification language such as Object-Z can be more appropriate. Object-Z [2] adds support for the modularity and reuse constructs offered by object orientation to the ISO standardised formal specification language Z [3].

Object-Z specifications are appropriate for functional descriptions of systems, and convenient for abstracting above design details – the classes and class structure in the specification need not reflect in any way the intended implementation architecture. However, to implement a system specified in Object-Z with an object-oriented programming language a design is required. Such a

Email address: tjm@itee.uq.edu.au (Tim McComb, Graeme Smith).

design needs to specify the behaviour and data of individual classes, and additionally the static structure of the architecture (how the classes are related).

Presently there is little means of moving from an object-oriented functional specification to a practical object-oriented design in a rigorous manner. The semantics of Object-Z lend the language to such a methodology though, as the reference-based treatment of object identity and syntactical inheritance (there need be no subtype relationship) closely resemble object-oriented programming languages like Java [4], C^{++} [5], and Smalltalk [6].

We have likened the process of modifying an Object-Z specification to introduce design elements to that of refactoring [7], which Fowler describes as the application of simple rules for the improvement of the design of existing code. This is because we utilise rules that are similar to the rules used for refactoring; except they apply to abstract specifications, and their motivation is primarily to introduce designs rather than improve existing ones.

Our rules share one important similarity with Fowler's: they must be equivalence transformations in terms of behavioural interpretation. We have demonstrated previously that the application of two rules for refactoring specifications in Object-Z – *annealing* and *coalescence* – can derive a vast array of differing object-oriented designs from abstract specifications [8,9] [1].

The annealing rule effectively splits a class's state and operations into two classes – one holding a reference to an instance of the other. The coalescence rule merges two classes together to create a new class that simulates both. These two rules both deal with referential structure, and do not cover the other primary form of object-oriented design structure: inheritance.

In this paper, we fill this gap by formalising rules that permit the modification of inheritance hierarchies. This incorporates the *introduce inheritance* and *introduce polymorphism* refactoring rules. We also generalise the annealing rule with the rule *introduce instances* to allow for collections of object instantiations to be introduced, thus significantly improving upon the original rule which only allowed for single instantiations, and further we allow classes to be parameterised with the rule *introduce generic parameter*.

As well as providing arguments for the soundness of the above new rules and extensions to previous rules, we argue their *completeness* for the derivation of all possible designs (structurally speaking) which do not contain unbounded recursive constructs in Object-Z. That is, through the use of *introduce generic parameter, introduce inheritance*, *introduce polymorphism*, and *introduce instances*, it is possible to move from any structural design in Object-Z that does not contain unbounded recursion to any other design that similarly does

[1] The *reflection* rule in [9] is a special application of the annealing rule.

not contain unbounded recursion and that represents a valid refinement [10] of the original.

Before proceeding with the rule descriptions (Sections 3 to 7) and the completeness argument (Section 8), we provide an overview of the Object-Z specification language (Section 2). Afterwards, we conclude with a discussion of the overall design process (Section 9), related work (Section 10), and future work in the area (Section 11).

2 Object-Z

Object-Z extends the formal specification language Z [3] with explicit support for the fundamental constructs of object orientation: classes, objects, inheritance and polymorphism. A full description of the language can be found in [2]. In this section, we provide an overview of the features of the language relevant to this paper.

2.1 Classes

A class in Object-Z groups together a collection of state variables with their initial conditions and all operations which may change their values. Such a class may have one or more generic type parameters. For example, the Object-Z class $Stack[X]$ in Figure 1 has a state variable *items* of type $\mathrm{seq}\, X$ (a sequence where the elements are of the generic type T). The *items* sequence is initially empty. The operation *Push* concatenates ($\frown$) the input *item*? to the beginning of the *items* sequence, and the operation *Pop* removes the first *item*! from the beginning of the sequence, the value of which is output.

The state variables, initial state schema *Init*, and operations of a class are collectively referred to as its *features*, where the interface of the class is exposed by the *visibility list* ($\upharpoonright$). The initial state schema and operations of a class include predicates constraining the possible values of the state variables. In operations, a variable decorated with a prime, e.g., x', denotes the post-state value of the variable.

The list of variables following the symbol Δ in an operation is referred to as the delta-list. This list comprises all variables which the operation may change; any variables not listed are unchanged.

An operation has an implicit *guard* determining whether or not it can occur in a given state. For the *Push* operation in $Stack[X]$, this guard is true; it

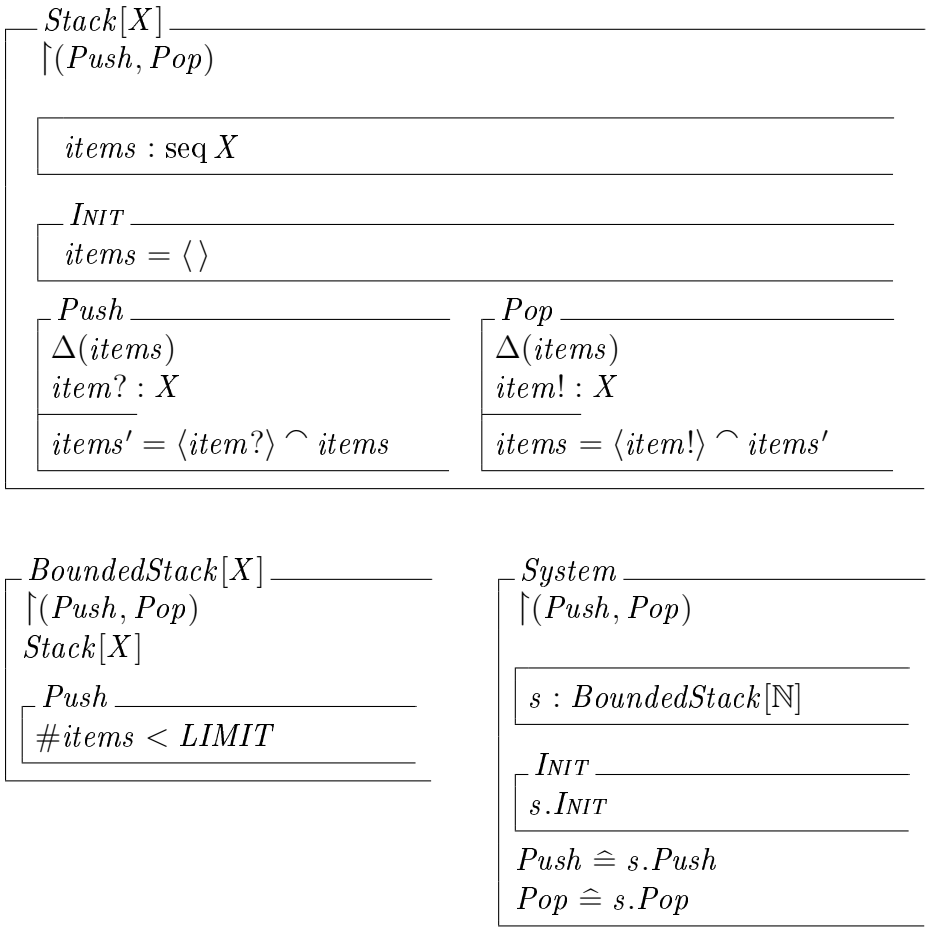

Figure 1. Example Object-Z specification

can occur at any time. However, operation *Pop* requires that *items* contain at least one element, so its implicit guard is $\#items > 0$.

2.2 Inheritance

Object-Z supports multiple inheritance. A subclass implicitly includes all of the features of its superclasses, and may also modify or add to these features. As an example, consider the class *BoundedStack*[*X*] in Figure 1 which inherits *Stack*[*X*].

In the class *BoundedStack*[*X*], the state, initial conditions, and operations of *Stack*[*X*] are implicitly conjoined with those declared in *BoundedStack*[*X*]. The class *BoundedStack*[*X*] has an operation *Push*, as does *Stack*[*X*]. The predicate and delta-list of *Push* in *Stack*[*X*] are implicitly part of *Push* in *BoundedStack*[*X*]. Hence, *Push* in *BoundedStack*[*X*] adds an element to the beginning of *items*, as before, but now has the guard $\#items < LIMIT$.

When a class is inherited, it's visibility list is not. Hence, any restrictions on access to features of the class need to be included again when required.

2.3 Objects

Classes in Object-Z may be used as types. Instances of such types are references to objects of the class. For example, in the class *System*, $s : BoundedStack[\mathbb{N}]$ is a reference to an object of class $BoundedStack[\mathbb{N}]$. In all cases of classes with generic parameters, the parameters must be instantiated in order for the class to be used as a type. For the above class definition, $s : BoundedStack[\mathbb{N}]$ declares a reference s to an object of the class $BoundedStack[X]$ with its generic parameter X instantiated with $\mathbb{N}$.

Adopting a reference-based type for classes allows objects with the same class and same values of their state variables to be semantically distinguished. The reference to an object acts as the *identity* of the object distinguishing it from all others.

So that objects can refer to their own identity, all classes C have an implicitly declared constant $self : C$. The value of the constant for a given object is the reference to that object.

Given the declaration $s : BoundedStack[\mathbb{N}]$, the notation $s.\textsc{Init}$ denotes a predicate which is true precisely when the referenced object is in its initial state. Also, the notation $s.Push$ is an operation corresponding to the referenced object undergoing its *Push* operation, and $s.Pop$ is an operation corresponding to the referenced object undergoing its *Pop* operation.

Promoted operations, i.e., those of the form $a.Op$, are often combined with other promoted operations or operations of the class in which they occur. A number of operation composition operators exist to facilitate this including those for conjunction $\wedge$ and choice $[]$ [2].

We can also access the state variables of the referenced object using the notation $a.x$ and $a.y$. Such access to state variables is often avoided in specifications, however, to allow for *data refinement*, i.e., where the state variables and their types are changed as the class specification is refined towards an implementation [10]. It is possible to explicitly restrict access to a class's state variables (or other features) by adding a visibility list to the class [2].

2.4 Polymorphism

Object-Z supports the standard inheritance-based polymorphism of object orientation. To declare a reference to an object of a given class C or any of its subclasses we write $c : \downarrow C$. This notation is only applicable when all subclasses of the given class have the same number of generic parameters as the given class, and at least all of the visible features of the given class.

Object-Z also supports a more general form of polymorphism called *class union* which is independent of inheritance. Class union allows the declaration of a reference to an object of one of an arbitrary set of classes. For example, given classes C and D, the declaration $c : C \cup D$ declares c to be a reference to an object of either class C or class D.

When using either type of polymorphism, only features common to all possible classes of the referenced object may be accessed. This common set of features is referred to as the *polymorphic core*. Due to the restrictions on inheritance-based polymorphism, the polymorphic core in this case is given by the features of the superclass.

3 Refactoring rules

Specification-based design in Object-Z presents a separation of concerns. The internal data and operations of classes can be modified behind their interface (as abstract data types) through a process of class refinement; but this process should be considered independently from modifying the composition of classes to change the structure of the system as a whole. We do not detail any method of individual class refinement in this paper as such a theory has been presented previously [10] – rather, we concentrate upon the latter problem of sculpting the architecture of the system. To this end, some of the refactoring rules make assumptions that certain local (non-structural) refinements have occurred as a precondition to applying the rule, usually with the intent of changing the class into an expected form for the refactoring to take place. Even though this process plays an integral part in the overarching methodology, we refer the reader to the technical treatment of class refinement presented in [10].

Class instantiation, class inheritance, polymorphism, and generics (class parameters or templates) are four object-oriented architectural constructs which are almost universal. They underpin the paradigm and provide the modularity and reuse capabilities. In this section we provide a transformation rule to deal with each construct: one to introduce generic class parameters; one to introduce inheritance; one to introduce polymorphic behaviour; and one

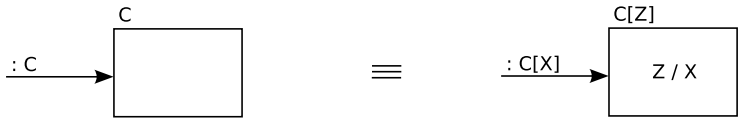

Figure 2. Introduce generic parameter refactoring

to introduce instantiations of new objects. These rules are equivalence transformations in the sense that the behaviour of the resulting specification is equivalent to that of the original. Hence, they can also be applied in reverse to *remove* structure, thus providing a powerful mechanism for altering design structure at the specification level.

Sections 4 – 7 describe each of the rules in turn. Interleaved with the presentation of the high-level rules are some practical rules that are, whilst just being specialisations and compositions of the high-level rules, nevertheless useful and perhaps more resemblant of Fowler's programming language level rules [7].

4 Introduce generic parameter

This section provides a description of the *introduce generic parameter* refactoring rule. Generic parameters in Object-Z allow a type, or a list of types, to be passed as parameters to a class. The parameters rename the types in the class definition to be that of the type passed, so with generic class definition the parameterised types act as placeholders. Refactoring a specification to add support for generic parameterisation of classes is desirable, as it allows for the derivation of library components, and increases the reuse throughout the design as single class may be instantiated many times with different parameters. The parameterised classes in Object-Z could possibly be implemented using the support for *generics* in Java [11] or *templates* in C^{++} [5].

In Object-Z parameterisation is defined through renaming [2]. In Figure 2, a class C (represented by a named box) that holds references to a type X is replaced with a class $C[Z]$, where Z is a fresh variable, and each occurrence of X is renamed to be Z (the notation Z/X is used to signify this renaming). Further, any reference in the specification to C (references to class instances are denoted by the class name preceded by a colon, and a closed-headed arrow) is changed to be to $C[X]$, including references for inheritance.

This refactoring rule only introduces one parameter, but repeated application can provide as many parameters as necessary. As new parameters are added, they are appended to the right of the parameter list. For example, Figure 3 represents the result of the rule being applied again to the right-hand side of Figure 2 making type Y a generic parameter.

Figure 3. Repeated application of introduce generic parameter

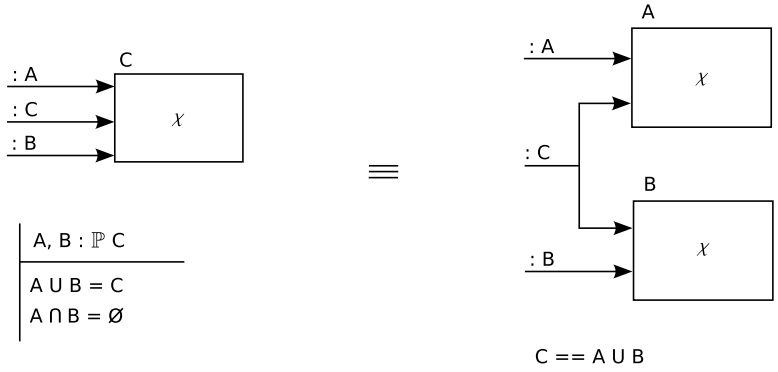

Figure 4. Introduce polymorphism refactoring

4.1 Soundness

X is renamed to Z by the refactoring rule, which is then renamed back to X whenever the class is referenced – preserving the equivalence of the transformation. Since Z is just a placeholder, there is no possibility of introducing type inconsistency.

5 Introduce polymorphism

Not all inheritance hierarchies need be polymorphic in Object-Z, just as not all polymorphism must be confined to inheritance hierarchies. In programming languages like Java it is common to have classes implement *interfaces* that provide a mechanism for polymorphism that is not related to inheritance. This orthogonal treatment of polymorphism both in Object-Z and in some programming languages warrants a rule specifically for its introduction, rather than just having it as a by-product of introducing inheritance. We have chosen to present this rule, *introduce polymorphism*, prior to discussing inheritance as the introduce inheritance rule (Section 6) requires the application of this rule as a prerequisite for constructing hierarchies that are polymorphic.

Figure 4 shows the refactoring rule diagrammatically, where the left-hand side and the right-hand side are equivalent. The class C on the left-hand side has three means of referencing it: by C, or by two aliases A and B which partition the identity space of C.

The introduction of polymorphism is normally motivated by the identification of a class (C) that behaves in different ways depending upon how it is used in context. The rule requires that the designer identify the contexts where the alternate behaviours are expected, and divide the references between A, B, and C accordingly.

The introduce polymorphism rule allows for the splitting of these behaviours into separate classes (A and B on the right-hand side of the figure). To execute the refactoring transformation, all of the features of class C (represented by χ in the figure) are copied verbatim to define both classes A and B. The class C is removed from the specification, but C is globally defined to be the class union of A and B – thus providing for the polymorphism. If a designer wishes to identify and extract more than two behaviours from a single class, then this can be achieved by repeated application of the rule.

Because C becomes a class union after application of the transformation, C cannot be inherited by any other classes in the specification after this refactoring is applied (this is a restriction of the Object-Z language [2]) – such classes must inherit either A or B instead.

There is an axiomatic definition on the left-hand side that describes the typing relationships between A, B and C. The designer must add this to the specification as a precondition to applying the rule. The axiomatic definition is not only important to declare what A and B mean (i.e. that they are aliases for class C) but if the rule is applied in reverse (see Section 5.2), this axiomatic definition retains the vital information that relates the types.

References to A expect one behaviour, references to B expect the other behaviour, and references to C encompass references to A and B (a C is either an A or a B) [2]. It is expected that these different behaviours are explicitly guarded. For example, the designer may wish for an operation Op in C to behave in two different ways captured by the operation schemas α and β, depending upon its context. The identification of these contexts is achieved by instantiating C as A or B respectively. The designer then guards these behaviours through the use of the *self* keyword inside the operation: using $Op \mathrel{\widehat=} ([self \in A] \land \alpha) \,[]\, ([self \in B] \land \beta)$, ensuring that the introduction of the guards does not affect the behavioural interpretation of the specification (e.g., whenever a reference to A is introduced, α would have always been the behaviour of Op prior to the application of the rule).

Although the class definition is copied, the use of these guards becomes crucial after the application of the refactoring. The designer can substantially simplify the class definitions A and B by realising that in class A, $[self \in A] \equiv [\text{true}]$

[2] There need be no distinction between the behaviours of A and B, but this would render the application of the rule largely redundant.

and $[self \in B] \equiv$ [false] are invariant; and likewise in class B, $[self \in B] \equiv$ [true] and $[self \in A] \equiv$ [false] always hold. Therefore, in class A, Op simplifies to α. Similarly, in class B, Op simplifies to β.

It is particularly important to realise that for instances $a : A;\ b : B$, it is never the case that $a = b$ because the identity sets are disjoint ($A \cap B = \varnothing$). If A and B objects need to reside in a common data structure, for example a set declaration using the class union $A \cup B$, then references to C must be used.

The labels A, B, and C are representative only: the rule mandates that A and B are fresh names, and generic parameters are carried across ($C[X]$, for example, becomes $A[X]$ and $B[X]$).

5.1 Soundness

Because the class definition (C) on the left-hand side is copied verbatim to form A and B on the right-hand side, the references to A and B are not affected by the application of the rule. Furthermore, C is not affected since the polymorphic core of $A \cup B$ on the right-hand side contains exactly those features of C on the left-hand side. The class union on the right-hand side is equivalent to the declarations of A and B and the first predicate of the axiomatic definition on the left-hand side. The second predicate $A \cap B = \varnothing$ is maintained by the fact that any two Object-Z classes have disjoint identity sets [2] .

5.2 "Coalescence"

The reversal of the introduce polymorphism rule is particularly interesting, enabling two classes to be merged into one class. This allows the designer to extract common functionality from possibly independent parts of the system and *coalesce* it into a single class. As stipulated by the rule definition, these classes must be identical. However, any classes that are candidates for coalescence can be made identical through the merging of the class definitions (utilising $[self \in \ldots]$ guards), and the addition of state variables so that each state becomes the union of the two original class states.

The previously published rule specifically designed for coalescence [8] has been superseded by the reversal of the introduce-polymorphism rule, as the reversal of this rule is more generally applicable. For instance, the original coalescence rule could not be applied where class union was being utilised over the two classes.

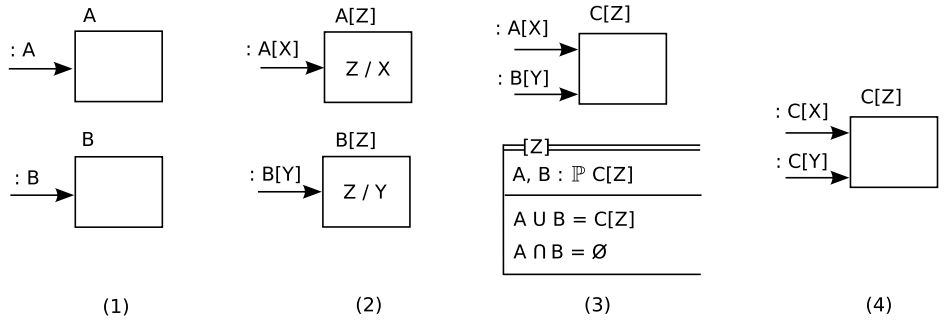

Figure 5. "Generic Coalescence" composition

5.3 "Generic coalescence" composition

The coalescence process can be used to merge classes that are similar but use different types, into a single class with generic parameterisation of types. This can be achieved by combining the reverse application of the introduce polymorphism rule with the introduce generic parameter rule. Figure 5 illustrates this process on two classes, A and B, that are the same or similar except for the use of different types: the specification at step (2) is derived from that at step (1) by the application of the introduce generic parameter rule (twice), which is then transformed into the specification at step (3) through coalescence. Since class union is not utilised in this situation, $A[X]$ and $B[Y]$ may be transformed into $C[X]$ and $C[Y]$ respectively, thus forming the result in step (4). The axiomatic definition defining A and B is removed, as it serves no purpose (A and B are no longer referred to in the specification).

6 Introduce inheritance

Reuse of data constructs and operations in classes is achieved through inheritance in the object-oriented programming (and specification) paradigm. The *introduce inheritance* rule offers a means through which to build an inheritance hierarchy from existing classes.

The introduce inheritance rule creates an inheritance relationship between any two classes in the specification, as long as the addition of the relationship does not result in a circular dependency.

Figure 6 illustrates the application of the rule to two classes A and B with features α and β respectively. It also illustrates the use of the short hand polymorphism notation $\downarrow A == A \cup B$.

The rule is most effectively applied to link together classes that contain common features in order to maximise the potential for reuse, but the classes

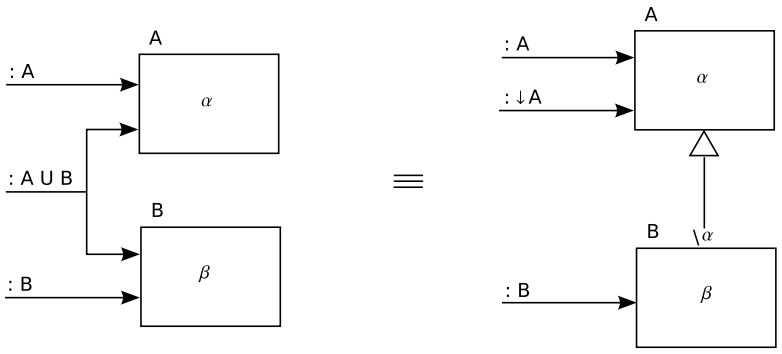

Figure 6. Introduce inheritance refactoring

need not share any features at all. This is because the introduce inheritance rule not only adds the inheritance relationship (indicated in Figure 6 by the open arrow-head) but also hides every feature of the superclass (the notation '$\backslash\alpha$' indicates that all features α of the superclass A are hidden). The combination of the inheritance and the hiding effectively renders the refactoring rule an equivalence transformation, as it does not change the meaning of the specification.

To reuse features of the superclass the designer must make further alterations local to the subclass to unhide, and perhaps rename, the features inherited from the superclass [3]. If the features of the superclass revealed by the designer to the subclass are indeed shared features, then these local changes will be equivalence transformations – but the onus is upon the designer to appropriately select and justify the reuse. Since inheritance in Object-Z is syntax-based, this justification is often a pattern-matching exercise rather than a theorem proving one.

The reversal of the introduce inheritance rule removes the inheritance relationship under the assumption that every feature of the superclass concerned is hidden. This precondition can be satisfied in any case by copying the feature definitions from the superclass (that are not already hidden) into the subclass.

6.1 Soundness

The rule converts the reference to the class union $A \cup B$ on the left-hand side to $\downarrow A$ on the right-hand side when the inheritance relationship is introduced. By definition, $\downarrow A == A \cup B$ [2] in this case, so this is an equivalence transformation.

The existence of the inheritance relationship on the right-hand side does not

[3] The subclass specifies the hiding and renaming of inherited features in Object-Z [2].

change the meaning of the superclass A on the left-hand side, as no features are added, altered, or removed. Because every feature of the superclass A is explicitly hidden from the subclass B on the right-hand side, B is also equivalent for the same reason: no features are added, altered, or removed.

6.2 "Extract superclass" composition

Extracting a superclass from an existing class in a design is a common refactoring [7,12] and it is one example of a transformation that can be derived by combining the rules we have presented so far.

A designer may utilise guards to distinguish the general behaviour α from the specific behaviour β of a class. Then, through the sequential composition of the introduce polymorphism rule with the introduce inheritance rule, a superclass can be extracted. Features of the new superclass that the designer wishes to utilise (rather than redefine) in the subclass can then be removed from the subclass and 'unhidden' in the inheritance relationship. Since the introduce polymorphism rule copies the class definition, every feature of the superclass may be unhidden without concern over maintaining the equivalence of the transformation.

7 Introduce instances

All of the rules presented so far have dealt with the static structure of classes but not with the instantiation of classes to reference individual objects. The *introduce instances* rule addresses this deficit by introducing to a class a set of instances of itself. When used in conjunction with the introduce polymorphism rule (Section 5), the designer can yield elaborate structures of class instantiations between different class definitions.

Like the other rules, the introduce instances rule is an equivalence preserving transformation in terms of behavioural interpretation. Previously in the literature [8,9] the *annealing* rule was presented that performed a similar function, however this rule was limited to the extraction of only one instance to a new class, and consequently did not address the challenges of managing object construction and disposal. The introduce instances rule is a generalisation of the annealing rule that allows for the introduction of an arbitrary number of instances.

A class is identified – in this case A (refer to Figure 7) – which manages a set containing data elements β for which the designer intends to eventually move

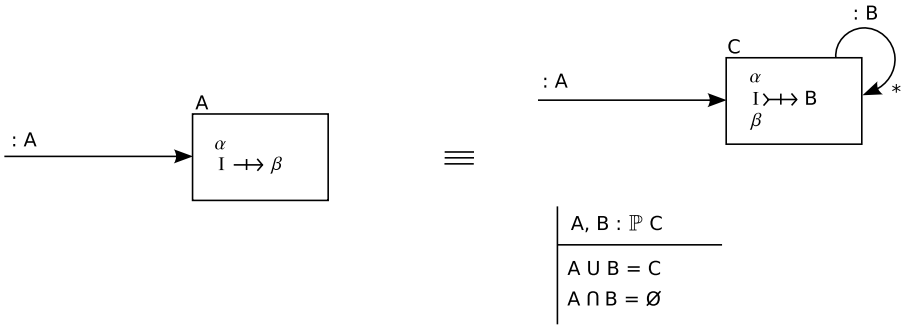

Figure 7. Introduce instances refactoring

into individual objects of class type B. The rest of the state of A is represented by α. The notation $I \nrightarrow \beta$ denotes the precondition that the designer must reference the set of data elements using an indexing function in the state (this function may need to be introduced as a precursor to applying the rule, and will be further explained in the following section).

The rule operates by adding instances to the class A which are instances of A itself (so A is self-referencing), but with the class name B introduced to separate external references to A from the internal (B) references to A. The β data elements are replaced to become instances of 'B', and all operations that act upon $I \nrightarrow \beta$ are split up and/or modified to accommodate the change. Precisely how the operations in A are affected by the rule depends upon their operation on the state – this is explained in Section 7.2.

The state of A is augmented to include a single β, and the operations are potentially split up and then *guarded* depending upon whether they apply to a single β or whether they apply to the rest of the state α and $I \nrightarrow B$. The name of the class A is changed (in this case to become C), and these alternatives then represent different behaviours of C, which can be separated into distinct classes at a later stage using the introduce polymorphism rule (see Section 7.4).

Because the state schema of A is changed, it is important to ensure that the indexing function is not visible. Operations in subclasses of A that have access to data in β must be considered in addition to operations local to A when the rule is applied.

The determination of how the operations (or parts of operations) need to be guarded is achieved through the use of *classifiers*. The definition and interpretation of these classifiers is detailed in Section 7.2, but immediately following is a description of the indexing function which is used to introduce and manage the object instantiations.

7.1 Indexing function

The introduce instances rule always operates upon sets of data elements, even if only one instance is to be extracted (this particular case will be discussed in Section 7.4). A non-visible indexing function must be introduced to manage this set before it can be extracted into a set of instantiations, because we must distinguish the addition and removal of elements from the set from the modification of elements within the set. With just a set, all operations are of the former category, but with an indexing function over the set this is not necessarily the case. Naturally, the domain of the indexing function relates to the identities of objects that are being introduced by the rule.

The identity function takes the form of *data* in class A of Figure 8. $IDENTITY$ and $DATA$ can be defined arbitrarily. However $DATA$ is usually a cross product of the state variable types that are going to constitute the variables in the extracted class – this data type acting as an envelope for the entirety of the state variables in the new class. Operations need to act upon the *data* function in a controlled manner, and may take the form of a *local* or a *delegate* operation, where local operations affect the domain (and thus also the range) of the *data* function, and delegate operations affect only the range. We consider only these two possibilities because any operation that acts upon *data* can always be considered as either one of these or a combination of both.

The introduce instances rule replaces the elements in the range of *data* with references to instances of the class A, such that *data* becomes a partial injection. The state of A is extended to hold a single $DATA$ element in a variable named *state*. Since $DATA$ acts as an envelope for the variables that the designer wishes to constitute the new state variable, further data refinements after the refactoring operation can extract and name the independent variables.

Guards are introduced into the initialisation predicate to discriminate between the two uses of the class C: A and B.

7.2 Operation classifiers

Operation classifiers are predicates that are used to determine whether an operation should be guarded as $[self \in A]$ or $[self \in B]$ when the rule is applied. Every operation in the class must be classified to determine if and how it must be altered to accommodate the changes to the specification. As mentioned in the beginning of Section 7, the operations are labelled as *local* or *delegate* respectively by the classifiers. If an operation predicate implies the classifier predicate, the operation then satisfies that classifier. In the next two

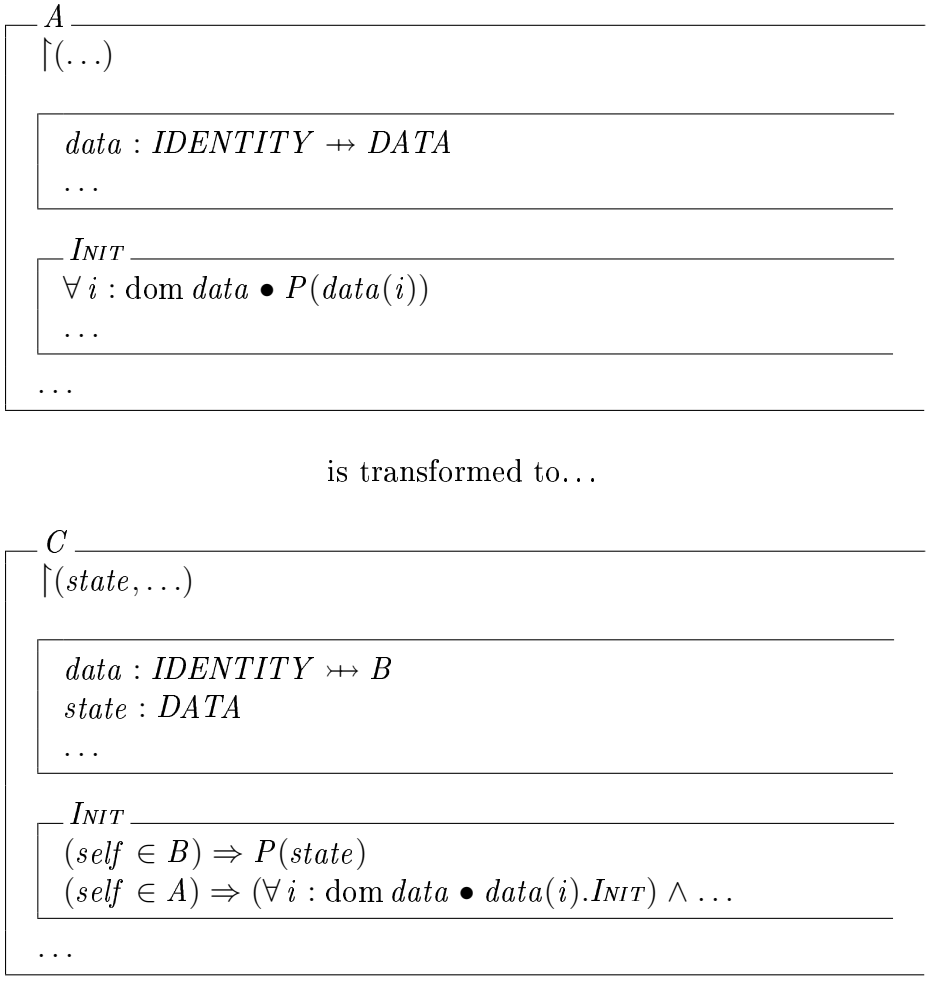

Figure 8. State transformation with introduce instances rule

sections we present the local and delegate operation classifiers, and following that we discuss the situation where an operation satisfies neither classifier and needs to be split.

7.2.1 Local operations

If the property below holds over the operation predicate P, it is considered local and must be guarded with $[self \in A]$.

$$P \Rightarrow data \cup data' \in (\_ \nrightarrow \_)$$

The notation $f \in (\_ \pfun \_)$ is used to specify the constraint that f is a partial function. This classifier stipulates that the union of the pre- and post-states of the data variable must form a partial function. Because we know that $data$ and $data'$ are partial functions independently, this classifier is true unless the predicate P specifies the alteration of the range value of an existing domain element in $data$ to form $data'$ (without removing the tuple). This is always the case for operations that do not affect $data$, as when $data$ is not in the delta-list of an operation, $data' = data$, so this reduces to $P \Rightarrow \text{true}$, which is true. Any range elements of $data$ mentioned in an operation classified as local need to dereference the object upon application of the rule, that is $data(i)$ becomes $data(i).state$.

7.2.2 Delegate operations

If an operation predicate P satisfies the following classifier, it is considered a delegate operation and should be guarded with $[self \in B]$.

$$P \Rightarrow \mathbf{deltalist} = \{data\} \land \operatorname{dom} data = \operatorname{dom} data' \land \operatorname{ran} data \neq \operatorname{ran} data'$$

where **deltalist** is a meta-level variable referring to the delta-list of an operation.

Intuitively, this identifies those operations that exclusively affect $data$, and further only modify a range element in $data$ without altering the domain. These operations should be invoked on the appropriate objects of class B.

The objects that the delegate operation needs to be invoked upon are selected by identifying which domain values have their associated range values altered. The operation must be rewritten by the designer into a list of terms, separated by operation composition operators. In each term, a guard G_i narrows the applicability of a predicate P_i to a subset of domain elements, where P_i specifies the relationship between the pre- and post-states of a $DATA$ element. The following template illustrates this notion, where $P_i(data'(id))$ is a predicate over the pre-state of the class and $data'(id)$ when the guard $G_i(id)$ holds on a domain element id. The asterisk ($\star$) is just a placeholder representing any of the Object-Z operation composition operators, and the notation $(\forall\, d \mid p \bullet q)$ is equivalent to $(\forall\, d \cdot p \Rightarrow q)$.

This template can be applied to any operation that satisfies the classifier, because $data'$ is the only free variable and further only its range can be modified (with respect to its pre-state). Since id covers the entire domain of $data'$ and each predicate has access to the full state, any aspect of the range of $data$ can be specified. Note that if a guard always selects exactly one object, by the one-point rule the universal quantifier is superfluous and can be dropped.

A

$\lceil(\ldots)$

$data : IDENTITY \pfun DATA$

$\ldots$

$\vdots$

$Op \mathrel{\widehat=} \left[\Delta(data) \mid (\forall\, id : \mathrm{dom}\ data \mid G_1(id) \bullet P_1(data'(id)))\right] \star$
$\quad \ldots \star$
$\quad \left[\Delta(data) \mid (\forall\, id : \mathrm{dom}\ data \mid G_n(id) \bullet P_n(data'(id)))\right]$

is transformed to...

C

$\lceil(state, Delgate_1 \,..\, Delegate_n, \ldots)$

$data : IDENTITY \rightarrowtail B$
$state : DATA$
$\ldots$

$\vdots$

$Op \mathrel{\widehat=} \left[self \in A\right] \land$
$\quad (\forall\, id : \mathrm{dom}\ data \mid G_1(id) \bullet data(id).Delegate_1 \star$
$\quad \ldots \star$
$\quad \forall\, id : \mathrm{dom}\ data \mid G_n(id) \bullet data(id).Delegate_n)$
$Delegate_1 \mathrel{\widehat=}$
$\quad \left[self \in B\right] \land \left[\Delta(state) \mid P_1(state/data(id), state'/data'(id))\right]$
$\ldots$
$Delegate_n \mathrel{\widehat=}$
$\quad \left[self \in B\right] \land \left[\Delta(state) \mid P_n(state/data(id), state'/data'(id))\right]$

$A, B : \mathbb{P}\, C$

$A \cup B = C$
$A \cap B = \varnothing$

Figure 9. Operation transformation with introduce instances rule

When the refactoring rule is applied, each predicate P_i becomes a new operation named $Delegate_i$, these operations being invoked depending upon the satisfaction of the guards. The template in Figure 9 illustrates how delegate operations are transformed. In the class C, $state/data(id)$ signifies that occurrences of references to $data(id)$ become references to the variable $state$.

7.2.3 Unclassified operations (so far)

Unclassified operations need to be split into multiple separate operations where each can be independently classified, which may be achieved in Object-Z by the promotion of logical operators to operation composition operators. We now discuss the only two general classes of operations that are unclassified, and further discuss *how* they must be split.

To derive the two cases, we first realise that an operation can never be classified as both a delegate and a local operation, as the conjunction of those classifiers is contradictory (there does not exist a P that could possibly satisfy both). This is because the partial functions $data$ and $data'$ must have common domains but differing range members to satisfy the delegate classifier, so there must exist at least one member in the common domain that maps to different range members in the respective functions. Thus the union of $data$ and $data'$ cannot itself be a partial function, which is a requirement of the local classifier.

An unclassified operation P must therefore adhere to neither classifier:

$$
\begin{aligned}
& P \Rightarrow \neg\, (\mathbf{deltalist} = \{data\} \land \operatorname{dom} data = \operatorname{dom} data' \land \\
& \qquad \operatorname{ran} data \neq \operatorname{ran} data') \land \neg\, (data \cup data' \in (\_ \pfun \_)) \\
& \equiv (\text{de morgan}) \\
& P \Rightarrow (\mathbf{deltalist} \neq \{data\} \lor \operatorname{dom} data \neq \operatorname{dom} data' \lor \\
& \qquad \operatorname{ran} data = \operatorname{ran} data') \land data \cup data' \notin (\_ \pfun \_)
\end{aligned}
$$

If we rewrite this single classifier as a disjunctive series of implications, we derive three potentially overlapping classifiers:

(1) $P \Rightarrow \mathbf{deltalist} \neq \{data\} \land data \cup data' \notin (\_ \pfun \_)$
(2) $P \Rightarrow \operatorname{dom} data \neq \operatorname{dom} data' \land data \cup data' \notin (\_ \pfun \_)$
(3) $P \Rightarrow \operatorname{ran} data = \operatorname{ran} data' \land data \cup data' \notin (\_ \pfun \_)$

By observing the third classifier in this list, we can infer that the domains of $data$ and $data'$ are not equal in this case, because otherwise $data = data'$ and hence $data \cup data' \in (\_ \pfun \_)$ – a contradiction. We extend Classifier (3) to reflect this fact:

(3) $P \Rightarrow \operatorname{ran} data = \operatorname{ran} data' \land \operatorname{dom} data \neq \operatorname{dom} data' \land data \cup data' \notin (\_ \pfun \_)$

Hence, any P that satisfies Classifier (3) also satisfies Classifier (2), as Classifier (3) is a stronger predicate. Thus we remove Classifier (3) as it is redundant.

Both remaining classifiers require the operation to be split. We shall discuss Classifier (1) and Classifier (2) individually, and then discuss the case when both classifiers are satisfied. We refer to Classifiers (1) and (2) as the *delta-list partitioning* and *domain partitioning* classifiers respectively.

7.2.3.1 Delta-list partitioning classifier

The classifier for delta-list partitioning is:

$$P \Rightarrow \textbf{deltalist} \neq \{data\} \land data \cup data' \notin (\_ \nrightarrow \_)$$

In this case P is an operation which needs to be a delegate, but it cannot because other variables are referenced in the delta-list besides *data*. The solution to this is to split the operation, which is generally achieved by promoting logical operators to operation composition operators and introducing communicating variables [2] where necessary. We will now present an argument that an operation which satisfies this classifier may be split apart in the general case.

Given that $data \cup data' \notin (\_ \nrightarrow \_)$, it follows that $data \neq data'$, so *data* must be a member of the delta-list. Because $\Delta \neq \{data\}$:

$$P \Rightarrow \textbf{deltalist} \supset \{data\} \land data \cup data' \notin (\_ \nrightarrow \_)$$

We define L to be $\Delta \setminus \{data\}$ and because of the proper superset relation, we know $L \neq \varnothing$. L represents the local variables that the operation changes excluding the *data* variable. Part of the operation needs to be delegated (that which changes the range of the *data* variable), and part must remain local (that which refers to the post-state of any variable in L). Clearly these concerns do not overlap as they apply to different variables, so such operations can be partitioned by the variables they change into one part that satisfies the local classifier, and one part that satisfies the delegated classifier.

7.2.3.2 Domain partitioning classifier

The classifier for domain partitioning is:

$$P \Rightarrow \mathrm{dom}\, data \neq \mathrm{dom}\, data' \land data \cup data' \notin (\_ \nrightarrow \_)$$

In this case part of the operation needs to be delegated (that which changes a range member such that $data \cup data' \notin (\_ \nrightarrow \_)$), and part must remain local (that which adds or removes mappings such that $\mathrm{dom}\, data \neq \mathrm{dom}\, data'$). In a similar fashion to the delta-list partitioning classifier, we present an argument that an operation which satisfies this classifier may be split apart in the general case.

For the domains to differ, members are either added, removed or both. We refer to the subset of the domain that identifies these introduced/removed members as δ, which is defined using symmetric difference $\delta = \mathrm{dom}\, data \triangle \mathrm{dom}\, data'$ (symmetric difference is defined as $S \triangle T = (S \cup T) \setminus (S \cap T)$)

Because $data \cup data' \notin (\_ \pfun \_)$, members in the domain of both $data$ and $data'$ must change mapping from that in $data$ to different range members in $data'$. The domain members that change mapping are represented by ρ. Because the members must exist in both domains, the following property holds over ρ: $\rho \subset \mathrm{dom}\, data \cap \mathrm{dom}\, data'$

It follows that δ and ρ must be disjoint $\delta \cap \rho = \varnothing$. This indicates that operations that adhere to this classifier may be split with respect to the domain members they act upon in $data$: one operation will satisfy the local classifier, one operation will satisfy the delegate classifier, where no domain member appears in both operations.

7.2.3.3 Simultaneous delta-list and domain partitioning Where both classifiers are satisfied by a predicate, the operation can be split as per the delta-list, and then by the domain. That is, the local variables in the delta-list (excluding $data$) may to be moved into a separate operation, and then the original operation that includes $data$ needs to be split again such that the activity over the domain of $data$ is separated between that which is local and that which must be split into a delegate operation.

7.3 Soundness

The indexing function of A is introduced as a precondition to applying the rule and must not be in the visibility list, thus the changes to the state schema of A do not affect the interface of the class, except to extend it with β [4].

Below, as a notational convenience, we will refer to the class C where $self \in A$ as C_A, and the class C where $self \in B$ as C_B. We also use the notation $A.data$, etc, to denote the variable $data$ of class A.

Because this rule changes the structure of the state, we require some mapping between the state schemas of classes A and C_A to show behavioural equivalence. Since we move the elements in the range of $data$ in A to each become a variable $state$ in an instance of C_B, we therefore require that $A.data(i) = C_B.data(i).state$.

Given this mapping, the initialisation schema and all local operations in A and C_A are equivalent. This can be seen by expanding the definitions in C_A according to the definition of the dot notation in Object-Z [2]. For delegate operations, we also need the condition that there is no possibility of interference between instances of C_B. That is, before the application of the rule, changes to

[4] Widening of the interface in Object-Z is allowable under refinement [10].

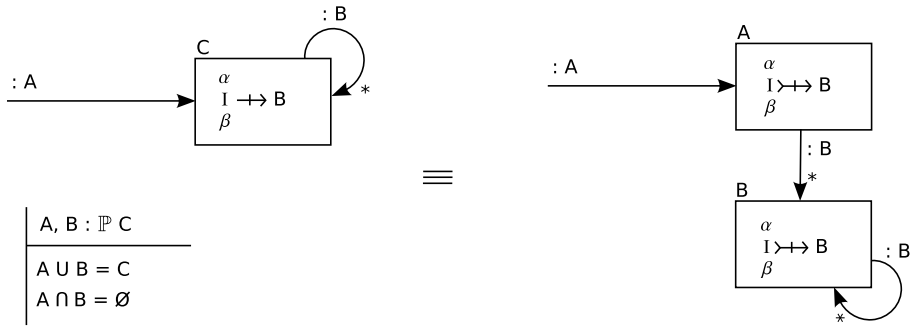

Figure 10. Introduce polymorphism rule applied for "Generalised Annealing"

$A.data(i)$ could not affect $A.data(j)$ when $i \neq j$. The partial injection ensures that this remains true for C_A after the application of the rule (since no object aliasing is possible). Since all operations can be classified as either local or delegate, there are no other types of operations that need to be considered.

7.4 "Generalised annealing" composition

The combination of introduce instances and introduce polymorphism yields instantiation relationships from a class to a new class (see Figure 10). The previously published annealing rule [8,9] is a special case of this process. The annealing rule extracts a set of variables from the state of a class and introduces a single instance to a new class which held those extracted variables in its state. This is still a convenient specialisation in itself as it strongly resembles the well established Extract Class refactoring [7] – often the introduction of more than one instantiation is not required. But the composition of the introduce instances refactoring with introduce polymorphism can create an arbitrary number of instantiations of the new class.

After the two rules are applied, we expect the classes would be substantially simplified to remove the unused parts of the states of A and B that are duplicated as a result of the introduce polymorphism rule.

8 Completeness

The four rules we have presented are minimal in the sense that they each operate upon orthogonal aspects of the object-oriented paradigm: no rule (or sequence of rules) can perform the function of any other rule. However, the question as to whether these rules are complete, such that they can be utilised in conjunction with individual class refinement to derive *any* design will be discussed in this section.

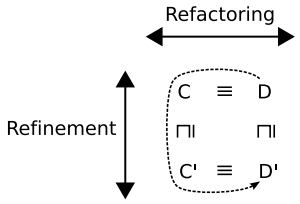

Figure 11. Design reduction and construction

Since each rule is an equivalence transformation, we seek to demonstrate their completeness by applying them in reverse to reason about which structures they can reduce. If a sequence of rule applications exists that can reduce a design down to a single class, then this design can be constructed from a single class by the forward application of the rules. By using a commuting argument (see Figure 11), we show that if a designer wishes to refactor a design D to a design D' that is a refinement of or equivalent to D, and both designs D and D' can be reduced to a single class C and C' respectively, then D' can be derived from D through the application of the refactoring rules and class refinement [10]. Provided we can demonstrate that any possible design can be reduced to a single class, this entails that any design can be refactored to any other design.

It is not possible to argue this in the general case with the four rules presented in this paper. The reason for this is an artifact of the introduce instances rule: it is impossible to create systems with *unbounded recursion*.

Unbounded recursion refers to a design where a class, either directly or indirectly, instantiates itself such that nested instantiations may extend recursively to an arbitrary (unbounded) depth. It is possible to create a class that instantiates itself by applying the introduce instances rule, but the process of introducing instances explicitly partitions the behaviour of the class (each operation is classified, and the classifiers do not overlap). This creates a single level of 'recursion', and repeated application of this process provides for further nesting, but this is always bounded as the state and functionality of the class cannot be split ad infinitum. A design which specifies a potentially infinitely long linked-list data structure, for example, cannot be derived through refactoring as it would require the rules to be applied an infinite number of times.

In this section we present an argument for the rules' completeness, but defer the treatment of constructing and reducing recursive specifications in Object-Z to Smith [13,14] where a theory of fixed-points is applied to reason about the translation between recursive and non-recursive specifications. The possibility of using refactoring rules to derive such designs is an area of future research.

We progress in four stages, relating to each of the four rules presented in this paper. Initially we remove the generic parameterisation of classes, then

we remove every inheritance relationship. We then collapse the referential structure and polymorphism down to a single class which references only itself – the total number of object instantiations in the specification being preserved. In the final stage, we remove this self-reference and reduce the total number of object instantiations down to one. The following sections deal with each reduction process in turn.

8.1 Generic parameterisation reduction

The removal of all occurrences of generic class parameterisation in a specification can be achieved by the introduction of new classes for each and every way the class parameters are instantiated. Renaming is performed, with respect to the context (parameters), for each introduced class – the explicit renaming removing the need for parameterisation. This process exactly corresponds to the reversal of the 'generic coalescence' process discussed in Section 5.3.

8.2 Inheritance structure reduction

Once all generic parameterisation of classes has been removed, inheritance relationships must also be taken out of the specification. This is achieved through the reverse application of the introduce inheritance refactoring. However, a precondition to applying this rule in reverse is that every feature of the superclass is hidden from the subclass – clearly this is not going to be the case for an arbitrary Object-Z specification. It is a straightforward, syntactic process to satisfy this precondition in any inheritance relationship by copying down definitions from the superclass that are not hidden from the subclass, and renaming such operations in the same manner as they are renamed by the subclass (if they are renamed at all).

Once this 'flattening' of definitions has been performed on every inheritance relationship in the specification, the reversal of the introduce inheritance rule can be applied to remove every inheritance relationship. It is worthwhile noting that, as per the definition of the introduce inheritance rule, this process converts all inheritance-based polymorphism ($\downarrow$) to class union.

8.3 Polymorphism and class structure reduction

The only design structure that remains in the specification after the last three steps is the instantiation relationships between classes, which may include polymorphism through the use of the class union operator.

The coalescence process (the reversal of the introduce polymorphism rule) effectively takes two classes and creates a single new class that performs the functionality of both. It is this refactoring that we base our reduction strategy for polymorphism and class structure. As a precondition, the classes need to be identical before this rule can be applied. However, any two classes may be made identical by guarding all of the operations depending upon the class that they originally belong to (see Section 5) and then copying their definitions between the two classes; renaming where necessary to avoid conflicts. The state variables may also be copied between the classes to form a common state that contains both state schemas. Again, some renaming may be necessary to ensure that there are no naming conflicts.

This process of making local changes to each class to capture the features of the other class will change the *polymorphic core* (see Section 2) if either class is involved in a class union declaration. Since the polymorphic core of a set of classes is the intersection of the features of the classes participating in the union, and the alterations required to satisfy the precondition of applying coalescence only add features to classes, the polymorphic core is only encompassing potentially *more* features. This does not affect the behaviour of the specification, because these added features could not possibly be referenced (as they were not previously in the polymorphic core).

Given that coalescence can be applied to any two classes in the specification, it follows that the refactoring can be repeatedly applied until there is only one remaining class in the specification. Although only a single class remains, there is no change whatsoever to the total number of object instantiations in the specification – the single remaining class simulates every class in the original design, and instantiates itself for every object instantiation that existed before the reduction process began.

Hence, all that remains is a single class containing references to itself. Because of the coalescence rule, these self-references are actually to disjoint subsets of the remaining class's identity space (expressed through the axiomatic relationships). These subsets correspond to the different classes in the original design, which were themselves all disjoint subsets of the global identity space [2] – this process has made the semantics relating to disjointness of class identity sets explicit.

8.4 Instantiation reduction

We remove the references from the single remaining class to itself through the reverse application of the introduce instances rule.

Because the coalescence process introduces guards based upon which class the

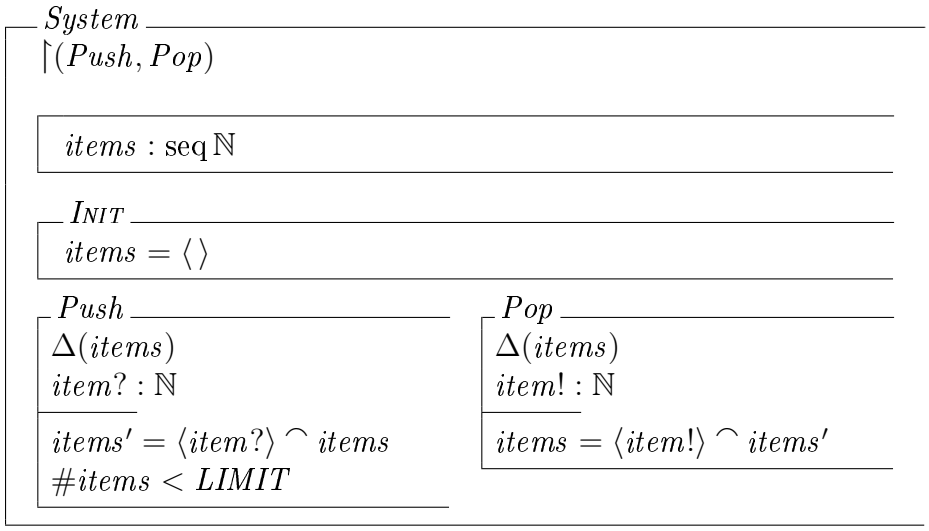

Figure 12. Original example Object-Z specification

behaviour and data belonged to, the single remaining class is compartmentalised in a very specific way – there is no overlap of guards. It is a precondition of applying the introduce instances rule (in reverse) that the features are strictly compartmentalised by these guards (see Section 7). This compartmentalisation is what we mean by 'bounded recursion'. Thus assuming that no unbounded recursion exists in the specification, the self-references can be removed by the reverse application of the introduce instances rule. The situations where this is not possible are a result of a class instantiating itself, either directly or indirectly, where a feature f of the class has the ability to reference the same feature f through the self-instantiation. The most common example of this phenomenon are specifications which include an operation which references itself recursively through another object. A full treatment of this kind of unbounded recursion, including a method for translating between recursive and non-recursive specifications, is presented in [13].

Apart from recursion, we need to consider cardinality. The semantics of Object-Z dictate that the total set of object identities is countably infinite [2, p28], so the introduction of an indexing function by the reversal of the introduce instances refactoring rule is not restrictive.

9 Design process

The process of deriving a design from an abstract specification is as much a product of the designer's imagination as it is a consequence of the refactoring rules. The example specification illustrated in Figure 1 was derived from the

specification in Figure 12. For brevity of presentation, we cannot show every step, but instead hope to provide the reader with a high-level impression of the practical application of the rules by listing the sequence in which they were applied. Apart from the local class refinement steps along the way, this sequence was: generalised annealing composition (to tease apart a separate class dedicated to the stack); introduce generic parameter rule (to make the stack class contain generic items); and lastly extract superclass composition (to form the stack generalisation of the bounded stack).

10 Related work

There has been much work on the formalisation of refactoring rules at both higher and lower levels of abstraction than Object-Z offers, with accompanying arguments for soundness and completeness.

At a higher level of abstraction than Object-Z, early work in the area of transforming object-oriented architectures using graph rewriting rules was presented by Bergstein [15]. This treatment considers two concerns: construction classes/relationships (instantiations) and alternation classes/relationships (inheritance). The refactoring rules we present differ from this work as our transformations are not necessarily object preserving (the introduce instances rule allows for new objects of classes to be created via the transformations). Also, we address two additional concerns – generic parameterisation and polymorphism, and describe all rules in the context of a full specification language.

At a lower level of abstraction than Object-Z, much work has been presented for refactoring at the programming language level. Mens et al. [16] also utilised a graph-based approach with sophisticated dependency analysis (using syntax and static semantics) to model and formalise refactorings at this level, but much of the preliminary work on refactoring originated in William Opdyke's PhD dissertation [17]. He describes refactoring with behaviour preservation based upon preconditions, which Fowler [7] built upon (and popularised) to form a methodology to improve the design of existing code.

Kerievsky [18] has presented a set of heuristics for refactoring towards design patterns [19] which compliments our approach, illustrating the effectiveness of using refactoring as a technique to introduce design architecture. Previously in Object-Z, the annealing and coalescence transformations have been utilised to derive design patterns [9] such as Observer, Proxy, and MVC (Model-View-Controller). Kerievsky's work reinforces the practicality of this approach.

Borba et al. [12] base a set of algebraic laws for refactoring on a weakest precondition semantics in the ROOL language – which is an object-oriented programming language similar to Java, but with a copy semantics. These refactoring rules are proved to be both sound and complete, but since it is

applied to a programming language it is not helpful for deriving designs from specifications which is our primary motivation.

However, later work by Gheyi and Borba [20] presented a set of rules for refactoring the Alloy specification language. To a large extent this represents a complementary approach, but their laws do not intuitively map to the object-oriented paradigm – with our approach the designer uses the rule and language construct that fits with the aspect of object-orientation that they wish to capitalise upon.

Goldsack and Lano [21,22] described an annealing rule (and coined the term) for the VDM^{++} specification language which is analogous to the annealing rule previously presented for Object-Z [8] (refer to Section 7.4). The introduce instances rule extends this annealing concept significantly by generalising the rule to allow for the creation of an arbitrary set of instantiations.

Kim and Carrington formalise a meta-model of the Unified Modelling Language [1] (UML) in Object-Z [23] and translate between Object-Z and UML using the meta-modelling approach [24]. They have recently proposed [25] the use of Object-Z as verification and validation tool for Model-Driven Architecture [26] (MDA) in UML. This is a practical target for our theory. Refactoring approaches that are based in UML have been presented in the past [27,28,29] but some do not preserve semantics, and UML is reasonably difficult to transform in a consistent manner because of its multiple diagrammatic representations. It is this weakness that forms a strong motivation for Kim and Carrington's work, as an Object-Z view of a UML model can be transformed such that consistency and semantics are always maintained.

11 Conclusion and future work

We have presented four rules for the purpose of refactoring an object-oriented specification with the aim of deriving a design, and argued that the rules are complete for architectures that do not contain unbounded recursion. The work is motivated by the need in some circumstances for a rigorous approach to software design that is tractable from an abstract specification, and relates to the current trend towards Model-Driven Architecture [26] in the software engineering discipline.

The rules are intuitive in the sense that they are directly related to the core aspects of the object-oriented paradigm: generics, polymorphism, inheritance, and objects. We believe that the well-established programming language refactoring rules [7] are intuitive (not just behaviour preserving), which has contributed to their popularity and success; the rules we have presented follow

that philosophy.

Apart from describing refactoring rules for introducing (and removing) unbounded recursion, future work in this area is directed towards the study of rule compositions to achieve sophisticated architectures, and the possible description of design patterns at the specification level through sequences of refactoring rule applications.

References

[1] G. Booch, I. Jacobson, J. Rambaugh, The Unified Modelling Language User Guide, Addison-Wesley, 1999.

[2] G. Smith, The Object-Z Specification Language, Kluwer, 2000.

[3] ISO/IEC 13568, Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics, first edition 2002-07-01 (2002).

[4] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[5] B. Stroustrup, The C^{++} Programming Language, Third Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] A. Goldberg, D. Robson, SmallTalk-80 The Language and its Implementation, Addison-Wesley, 1983.

[7] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison–Wesley, 1999.

[8] T. McComb, Refactoring Object-Z Specifications, in: M. Wermelinger, T. Margaria-Steffen (Eds.), FASE '04: Fundamental Approaches to Software Engineering, Vol. 2984 of LNCS, Springer-Verlag, 2004, pp. 69–83.

[9] T. McComb, G. Smith, Architectural Design in Object-Z, in: P. Strooper (Ed.), ASWEC '04: Australian Software Engineering Conference, IEEE Computer Society Press, 2004, pp. 77–86.

[10] J. Derrick, E. Boiten, Refinement in Z and Object-Z: Foundations and Advanced Applications, FACIT Series, Springer, 2001.

[11] Java 2 Platform Standard Edition 5.0
`http://java.sun.com/j2se/1.5.0/guide/`.

[12] P. Borba, A. Sampaio, A. Cavalcanti, M. Cornelio, Algebraic Reasoning for Object-Oriented Programming, Sci. Comput. Program. 52 (1-3) (2004) 53–100.

[13] G. Smith, Recursive Schema Definitions in Object-Z, in: ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B, Vol. 1878 of LNCS, Springer-Verlag, 2000, pp. 42–58.

[14] G. Smith, Introducing Reference Semantics via Refinement, in: ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods, Vol. 2495 of LNCS, Springer-Verlag, 2002, pp. 588–599.

[15] P. Bergstein, Object-preserving class transformations, in: Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications, 1991, pp. 299–313.

[16] T. Mens, S. Demeyer, D. Janssens, Formalising behaviour preserving program transformations, in: ICGT '02: Proceedings of the First International Conference on Graph Transformation, Vol. 2505 of LNCS, Springer-Verlag, 2002, pp. 286–301.

[17] W. F. Opdyke, Refactoring Object-Oriented Frameworks, Ph.D. thesis, Computer Science Department, Urbana-Champaign, IL, USA (May 1992).

[18] J. Kerievsky, Refactoring To Patterns, Addison-Wesley, 2004.

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[20] R. Gheyi, P. Borba, Refactoring Alloy specifications, Electronic Notes in Theoretical Computer Science 95 (2004) 227–243.

[21] K. Lano, Formal Object-oriented Development, Springer Verlag, 1995.

[22] K. Lano, S. Goldsack, Refinement of Distributed Object Systems, in: Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems, Chapman and Hall, 1996.

[23] S.-K. Kim, D. Carrington, A formal denotational semantics of UML in Object-Z, Special Issue of the Journal of l'Objet 7 (1) (2001) 323–362.

[24] S.-K. Kim, D. Carrington, An MDA Approach towards Integrating Formal and Informal Modelling Languages, in: Formal Methods (FM) 2005, Vol. 3582 of LNCS, Springer-Verlag, 2005, pp. 448–464.

[25] S.-K. Kim, D. Carrington, A Formal V&V Framework for UML Models Based on Model Transformation Techniques, submitted for publication (2005).

[26] Model Driven Architecture (MDA), Object Management Group (OMG), `http://www.omg.org/mda`.

[27] G. Sunyé, D. Pollet, Y. L. Traon, J.-M. Jézéquel, Refactoring UML models, in: UML, 2001, pp. 134–148.

[28] A. S. Evans, Reasoning with UML class diagrams, in: WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques, IEEE Computer Society, Washington, DC, USA, 1998, p. 102.

[29] M. Gogolla, M. Richters, Equivalence rules for UML class diagrams, in: J. Bézivin, P.-A. Muller (Eds.), The Unified Modeling Language, UML'98 - Beyond the Notation, 1998, pp. 87–96.