A Minimal Set of Refactoring Rules for Object-Z

Tim McComb[1] and Graeme Smith[2]

[1] ARC Centre of Excellence in Bioinformatics
Institute for Molecular Bioscience, The University of Queensland, Australia
[2] School of Information Technology and Electrical Engineering
The University of Queensland, Australia

Abstract. This paper presents a minimal and complete set of structural refactoring rules for the Object-Z specification language that allow for the derivation of arbitrary object-oriented architectures. The rules are equivalence preserving and work in concert with existing class refinement theory, so that any design derived using the rule set can be shown to be equivalent to, or a refinement of, the original specification.

1 Introduction

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. Object-oriented formal specification languages such as Object-Z [14], Alloy [6], and VDM^{++} [7] share these core features with their programming language counterparts. However, the way they are utilised to capture requirements associated with a problem domain is often quite different from the way in which they are used to implement a specific solution to a problem. The result is that an object-oriented specification does not usually directly resemble, in a structural sense, the design of the desired implementation. Here, structure is interpreted as the relationships between classes. Generally, a set of specification classes will describe a system of many more interacting implementation classes.

To bridge this gap between specification and implementation, specification refactoring rules have been proposed [7–9, 11, 2, 5]. These allow the structure of a specification to be incrementally transformed to represent a given design. Goldsack and Lano [7, 8], for example, introduced the ANNEALING rule to VDM^{++}. This rule effectively splits a class's state and operations into two classes — one holding a reference to an instance of the other. It was later adapted to Object-Z by McComb [9] who also introduced the COALESCENCE rule. This second rule merges two classes together to create a new class that simulates both. Together these rules have been shown quite effective for introducing designs [11].

However, these rules both deal with referential structure, and do not cover the other primary forms of object-oriented design structure: inheritance, polymorphism and generics. In this paper, we fill this gap by formalising rules for Object-Z that permit the modification of inheritance hierarchies and allow classes to be parameterised. Furthermore, we show that our set of rules is both minimal and complete for refactoring Object-Z specifications to derive designs.

We begin in Section 2 with an overview of the Object-Z language. We then provide rules to introduce generic class parameters, to introduce polymorphic behaviour, and to introduce inheritance in Sections 3 to 5 respectively. Together with the ANNEALING rule, these three rules have been shown to be *complete* in the sense that any reasonable design can be derived from any specification [10]. The proof is outlined in Section 6 before we conclude in Section 7.

2 Object-Z

Object-Z [14] extends the formal specification language Z [16] with explicit support for the fundamental constructs of object orientation: (generic) classes, objects, inheritance and polymorphism. Here we overview the notation for basic classes and objects. Other notation will be introduced in the following sections.

A class in Object-Z groups together a collection of state variables with their initial conditions and a set of operations which may change their values. The state variables, initialisation predicate, and operations of a class are collectively referred to as its *features*. The interface of the class is specified by a *visibility list* of the form $\upharpoonright(\ldots)$ listing its externally accessible features.

Consider, for example, the following Object-Z class *Stack*.

Stack

$$\upharpoonright(Push, Pop, INIT)$$

$$\begin{array}{|l} LIMIT : \mathbb{N} \\ \hline LIMIT = 1024 \end{array}$$

$$\begin{array}{|l} items : \operatorname{seq} \mathbb{N} \\ \hline \# items \leq LIMIT \end{array} \qquad \begin{array}{|l} \textit{INIT} \\ \hline items = \langle\,\rangle \end{array}$$

Push

$$\begin{array}{|l} \Delta(items) \\ item? : \mathbb{N} \\ status! : \mathbb{Z} \\ \hline \# items < LIMIT \Rightarrow status! >= 0 \land items' = \langle item? \rangle \frown items \\ \# items \geq LIMIT \Rightarrow status! < 0 \land items' = items \end{array}$$

$$Pop \mathrel{\widehat=} [\, \Delta(items)\ item! : \mathbb{N} \mid items = \langle item! \rangle \frown items' \,]$$

The class *Stack* has a state variable *items* of type $\operatorname{seq} \mathbb{N}$ (a sequence where the elements are natural numbers). The possible bindings of values for state variables are constrained initially by the initialisation predicate, and by the invariant predicate contained within the state schema. Hence, the *items* sequence is initially empty.

Each operation is a schema describing the relationship between pre- and post-state variables. A variable decorated with a prime, e.g., x', denotes the post-state value. All post-state variables from the state schema are available to operations, but by default they are equated to their pre-state counterparts (they do not change). Operations may introduce constraints over post-state variables by including them in a *delta-list* of the form $\Delta(\ldots)$. Any variables included in the delta-list have their pre-/post-state equality constraint relaxed.

For example, the operation *Push* from the *Stack* class concatenates ($\frown$) the input *item*? to the beginning of the *items* sequence, if the size of *items* is less than the *LIMIT*. Thus, *items* appears in the delta-list of *Push*. An output variable *status*! is set to be greater-than or equal-to zero if *items* can indeed accommodate the new item, otherwise the *status*! binds to a value strictly less-than zero and *items* remains unchanged. The operation *Pop* in this example removes the first item, bound to output variable *item*!, from the beginning of the sequence.

Such a class can be instantiated within another class and its visible features accessed using standard dereferencing notation. For example, consider the following class *System* which references a single object of class *Stack*.

System

$\upharpoonright(Push, Pop, INIT)$

$s : Stack$

INIT

$s.INIT$

$Push \mathrel{\widehat{=}} s.Push$

$Pop \mathrel{\widehat{=}} s.Pop$

Given the declaration $s : Stack$, the notation $s.INIT$ 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.

3 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. Refactoring a specification to add support for generic parameterisation of classes is desirable, as it allows for the derivation of library components, and increases reuse throughout the design as a 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 [1] or *templates* in C^{++} [15].

The rule is illustrated in Figure 1. A class C has a locally defined type L which is defined to be the actual type T. When the rule is applied, the class C is replaced with a class $C[X]$, where the name X is fresh, and the local definition which previously defined L as T is changed to define L as X. All references to C in the specification are replaced with references to $C[T]$, including references for inheritance.

$$\begin{array}{ccc} \text{(references to } C\text{)} & & \text{(references to } C[T]\text{)} \\ \begin{array}{|l} \hline C \\ L == T \\ \vdots \\ \hline \end{array} & \equiv & \begin{array}{|l} \hline C[X] \\ L == X \\ \vdots \\ \hline \end{array} \\ & & \textit{where } X \textit{ is fresh} \end{array}$$

Fig. 1. Introduce generic parameter refactoring

This refactoring rule only introduces one parameter, but repeated application can provide as many parameters as necessary. As new parameters are added, they can be appended to the right of the parameter list. For example, Figure 2 represents the result of the rule being applied again to the right-hand side of Figure 1, introducing a new parameter Y to stand for an actual type S. Exactly where in the list of parameters the new parameter is inserted is arbitrary, as long as the references are updated in a consistent manner.

$$\begin{array}{ccc} \text{(references to } C[T]\text{)} & & \text{(references to } C[T,S]\text{)} \\ \begin{array}{|l} \hline C[X] \\ L == X \\ M == S \\ \vdots \\ \hline \end{array} & \equiv & \begin{array}{|l} \hline C[X, Y] \\ L == X \\ M == Y \\ \vdots \\ \hline \end{array} \\ & & \textit{where } Y \textit{ is fresh} \end{array}$$

Fig. 2. Repeated application of the introduce generic parameter refactoring

The soundness of this rule follows directly from the semantics of generic parameters in Object-Z. A formal proof is presented in [10].

3.1 Example

The *Stack* in Section 2 specifically deals with natural numbers, but through the application of the above refactoring rule we are able to systematically introduce a generic parameter.

First, the *Stack* class must undergo a refinement step to introduce a local definition, such that the class conforms to the precondition required by the refactoring rule. The refinement step (actually, an equivalence transformation) would be proved using the simulation rules for Object-Z [3]. We present just the result of the refinement step here.

$$
\begin{array}{|l}
\hline
\textit{Stack} \\
\upharpoonright(\textit{Push}, \textit{Pop}, \textit{INIT}) \\
L == \mathbb{N} \\
\quad \begin{array}{|l}
\textit{LIMIT} : \mathbb{N} \\
\hline
\textit{LIMIT} = 1024
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{items} : \operatorname{seq} L \\
\hline
\#\textit{items} \leq \textit{LIMIT} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{INIT} \\
\textit{items} = \langle\,\rangle \\
\hline
\end{array} \\
\quad \begin{array}{|l}
\hline
\textit{Push} \\
\Delta(\textit{items}) \\
\textit{item}? : L \\
\textit{status}! : \mathbb{Z} \\
\hline
\#\textit{items} < \textit{LIMIT} \Rightarrow \textit{status}! >= 0 \land \textit{items}' = \langle \textit{item}? \rangle \frown \textit{items} \\
\#\textit{items} \geq \textit{LIMIT} \Rightarrow \textit{status}! < 0 \land \textit{items}' = \textit{items} \\
\hline
\end{array} \\
\textit{Pop} \mathrel{\widehat{=}} [\, \Delta(\textit{items})\ \textit{item}! : L \mid \textit{items} = \langle \textit{item}! \rangle \frown \textit{items}' \,] \\
\hline
\end{array}
$$

We are now in a position to apply the refactoring rule, which introduces the parameter X to *Stack* and updates the reference to *Stack* in the *System* class to instantiate the parameter with $\mathbb{N}$.

$$
\begin{array}{|l}
\hline
\textit{System} \\
\upharpoonright(\textit{Push}, \textit{Pop}, \textit{INIT}) \\
\quad \begin{array}{|l}
\hline
s : \textit{Stack}[\mathbb{N}] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{INIT} \\
s.\textit{INIT} \\
\hline
\end{array} \\
\textit{Push} \mathrel{\widehat{=}} s.\textit{Push} \\
\textit{Pop} \mathrel{\widehat{=}} s.\textit{Pop} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Stack}[X]\ }} \\
\upharpoonright(\mathit{Push}, \mathit{Pop}, \mathit{INIT}) \\
L == X \\
\begin{array}{|l}
\mathit{LIMIT} : \mathbb{N} \\ \hline
\mathit{LIMIT} = 1024
\end{array} \\
\begin{array}{|l}
\hline
\mathit{items} : \operatorname{seq} L \\ \hline
\#\mathit{items} \leq \mathit{LIMIT} \\ \hline
\end{array}
\quad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{INIT}\ }} \\
\mathit{items} = \langle\,\rangle \\ \hline
\end{array} \\
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{Push}\ }} \\
\Delta(\mathit{items}) \\
\mathit{item}? : L \\
\mathit{status}! : \mathbb{Z} \\ \hline
\#\mathit{items} < \mathit{LIMIT} \Rightarrow \mathit{status}! >= 0 \land \mathit{items}' = \langle \mathit{item}? \rangle \frown \mathit{items} \\
\#\mathit{items} \geq \mathit{LIMIT} \Rightarrow \mathit{status}! < 0 \land \mathit{items}' = \mathit{items} \\ \hline
\end{array} \\
\mathit{Pop} \mathrel{\widehat=} [\, \Delta(\mathit{items})\ \mathit{item}! : L \mid \mathit{items} = \langle \mathit{item}! \rangle \frown \mathit{items}' \,] \\ \hline
\end{array}
$$

4 Introduce Polymorphism

Not all inheritance hierarchies take advantage of polymorphism in Object-Z, and conversely 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 treating it as a by-product of introducing inheritance.

In Figure 3, the class $C[P_1, \ldots, P_m]$ on the left-hand side has exactly $n+1$ means of referencing it: by $C[P_1, \ldots, P_m]$, or by n axiomatically defined aliases $A_1[P_1, \ldots, P_m]$,…,$A_n[P_1, \ldots, P_m]$ which disjointly partition the references to objects of $C[P_1, \ldots, P_m]$. The parameter lists $[P_1, \ldots, P_m]$ are irrelevant to the application of the rule, except that all classes $A_1, \ldots, A_n$ and C must have the same arity. For brevity of presentation these parameter lists are omitted from the discussion below.

The introduction of polymorphism is normally motivated by the identification of a class (C) that behaves in different ways depending upon the context in which it is used. The Introduce Polymorphism rule requires that the designer identify the contexts where alternate behaviours are expected, and divide the references between $A_1, \ldots, A_n$ accordingly. Since the collection of class aliases $A_1, \ldots, A_n$ are together disjoint, any introduced references to these classes must be proved to be invariantly unequal whenever the references are to different aliases. This can be proved as a data refinement.

$C[P_1, \ldots, P_m]$
$\mathcal{F}$

$[P_1, \ldots, P_m]$
$A_1[P_1, \ldots, P_m] : \mathbb{P}\, C[P_1, \ldots, P_m]$
$\vdots$
$A_n[P_1, \ldots, P_m] : \mathbb{P}\, C[P_1, \ldots, P_m]$
$\langle A_1[P_1, \ldots, P_m], \ldots, A_n[P_1, \ldots, P_m] \rangle$
partitions
$C[P_1, \ldots, P_m]$

$\equiv$

$A_1[P_1, \ldots, P_m]$
$\mathcal{F}$

$\vdots$

$A_n[P_1, \ldots, P_m]$
$\mathcal{F}$

$[P_1, \ldots, P_m]$
$C[P_1, \ldots, P_m] ==$
$A_1[P_1, \ldots, P_m] \cup$
$\ldots \cup A_n[P_1, \ldots, P_m]$

Fig. 3. Introduce polymorphism refactoring

Assuming this identification and partitioning of object references has occurred, the Introduce Polymorphism rule allows for the splitting of the behaviours into separate class definitions (A_1 to A_n on the right-hand side of Figure 3). To execute the refactoring transformation, all of the features of class C are copied verbatim to define the classes A_1 to A_n. The class C is removed from the specification, but C is globally defined to be the class union $A_1 \cup \ldots \cup A_n$ — thus providing for the polymorphism. The identical feature sets of the classes are represented with the symbol $\mathcal{F}$ in Figure 3.

Since C is defined as a class union after the 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 [14]). Such classes must inherit one of $A_1, \ldots, A_n$ instead.

There is an axiomatic definition on the left-hand side that describes the typing relationships between $A_1, \ldots, A_n$ 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 the meaning of $A_1, \ldots, A_n$ (i.e., that they are aliases for class C) but if the rule is applied in reverse (to *coalesce* two or more classes), this axiomatic definition retains the vital information that relates the types.

There does not need to be any distinction between the behaviours of the aliases $A_1, \ldots, A_n$, but without it this would render the application of the rule largely redundant. When there is a distinction, it is expected that the 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_1 or A_2 respectively. The designer then guards these behaviours through the use of the choice operator $[\!]$ and the *self* keyword inside the operation: using $Op \mathrel{\widehat{=}} ([self \in A_1] \wedge \alpha) \mathbin{[\!]} ([self \in A_2] \wedge \beta)$, ensuring that the introduction of the guards does not affect the behavioural interpretation of the specification (e.g., whenever a reference to A_1 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 $A_1, \ldots, A_n$ by realising that in class A_1 (from the example above), $[self \in A_1] \equiv [\text{true}]$ and $[self \in A_2] \equiv [\text{false}]$; and likewise in class A_2, $[self \in A_2] \equiv [\text{true}]$ and $[self \in A_1] \equiv [\text{false}]$. Therefore, in class A_1, *Op* simplifies to α. Similarly, in class A_2, *Op* simplifies to β.

It is particularly important to realise that for instances $a : A_1;\ b : A_2$, it is never the case that $a = b$ because the references are disjoint ($A_1 \cap A_2 = \varnothing$). If A_1 and A_2 objects need to reside in a common data structure, for example a set declaration using the class union $A_1 \cup A_2$, then references to C may be used.

The labels $A_1, \ldots, A_n$ and C are representative only: the rule requires that $A_1, \ldots, A_n$ are fresh names, and generic parameters are carried across ($C[X]$, for example, becomes $A_1[X], \ldots, A_n[X]$). A proof of soundness of the rule is presented in [10].

4.1 Example

The Introduce Polymorphism rule tends to act as a bridge for applying more interesting refactorings: it is best utilised in combination with the other rules and interesting class refinements. So to provide an example for the Introduce Polymorphism refactoring, we are going to progress towards deriving a class $RepStack[X]$ which inherits $Stack[X]$. The class $RepStack[X]$ will define a *reporting* stack, where more helpful information is provided in the *status*! output of the *Push* operation. This requires inheritance, for which a rule will be provided in the Section 5, so we will only make progress toward the final goal at this stage (the example will be completed at the end of the Section 5).

We begin our example where the Introduce Generic Parameter rule ended, with the *System* class and a parameterised $Stack[X]$ class (refer to Section 3.1). In order to satisfy the prerequisite for the rule (being applied in the forward direction), we must introduce some new (generic) class aliases into the specification. As the new identifiers are fresh, this cannot affect the meaning of the specification.

$$\begin{array}{|l}
[X] \\
\hline
RegularStack[X], RepStack[X] : \mathbb{P}\, Stack[X] \\
\hline
\langle RegularStack[X], RepStack[X] \rangle \text{ partitions } Stack[X] \\
\hline
\end{array}$$

Although we have now satisfied the precondition for applying the rule to the $Stack[X]$ class, now is a good time to perform a refinement upon the *System* class to reference $RepStack[X]$ rather than $Stack[X]$. This refinement is correct, as *RepStack*$[X]$ is an alias for *Stack*$[X]$ (and therefore has the same meaning), so we omit a detailed argument as to its correctness and just present the result of the refinement:

System

$\upharpoonright(Push, Pop, INIT)$

$s : RepStack[\mathbb{N}]$

INIT

$s.INIT$

$Push \mathrel{\widehat=} s.Push$
$Pop \mathrel{\widehat=} s.Pop$

The reason we have chosen to perform the refinement at this stage is because it is easier to justify (as *RepStack*$[X]$ is behaviourally equivalent to *Stack*$[X]$), and we know that we wish to have *System* reference *RepStack*$[X]$ at the end of the refactoring process.

We now apply the Introduce Polymorphism rule to the *Stack*$[X]$ class. The yields two class paragraph definitions *RegularStack*$[X]$ and *RepStack*$[X]$ which exactly match (apart from the class name) the definition of *Stack*$[X]$ prior to the rule being applied. The definition of *Stack*$[X]$, in turn, becomes:

$[X]$

$$Stack[X] == RegularStack[X] \cup RepStack[X]$$

Our intention is to have *RepStack*$[X]$ inherit *RegularStack*$[X]$, which will become possible with the Introduce Inheritance refactoring rule presented in Section 5.

5 Introduce Inheritance

Reuse of data constructs and operations in classes is achieved through inheritance in the object-oriented paradigm (both with programming and specification). The Introduce Inheritance rule offers a means by 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 4 illustrates the application of the rule to two classes A and B with features $\mathcal{F}$ and $\mathcal{G}$ respectively.

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 need not share any features at all. This is because the Introduce Inheritance rule not only adds the inheritance relationship (indicated in Figure 4 by the inclusion of A in B) but also hides every feature of the superclass by assigning them a fresh name (the notation '$\mathcal{H}/\mathcal{F}$' indicates that all features $\mathcal{F}$ of the superclass A are hidden by assigning fresh names $\mathcal{H}$ for the features). The combination of

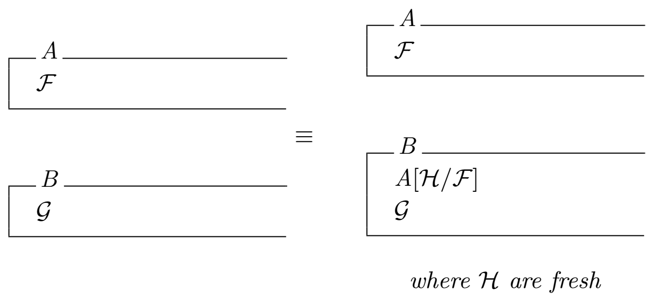

Fig. 4. Introduce inheritance refactoring

inheritance and hiding makes the refactoring rule an equivalence transformation, so long as we assume that the inheritance-based polymorphism operator $\downarrow$ is not used in the specification. $\downarrow C$ is an abbreviation for the class union of C with all of its subclasses. All occurences of $\downarrow$ in the specification must be replaced to specifically enumerate the inheritance hierarchy as a class union (that is, the $\downarrow$ must be replaced by its definition).

Some classes have an implicit visibility-list, whereby it is understood that every feature of the class is externally visible [14]. In such circumstances, the rule is still equivalence preserving as the interface of B is only widened, and the fresh features cannot possibly have been referenced as they did not previously exist.

To use the features inherited from the superclass, the designer must make refinements local to the subclass to reference the features in $\mathcal{H}$. Note that the introduced reference to the superclass must parameterise the superclass if it has generic parameters. These parameter instantiations of the superclass reference may include formal parameters of the subclass.

The reversal of the Introduce Inheritance rule removes the inheritance relationship under the condition that every feature of the superclass concerned is not referenced ("fresh"). As inheritance in Object-Z is syntax-based [14], this precondition can be satisfied by copying any referenced feature definitions from the superclass into the subclass. A soundness proof for the rule is presented in [10].

5.1 Example

The example of the previous section (4.1) was left unfinished owing to the absence of the Introduce Inheritance rule described above; we are now in a position to complete it.

The two classes, $\mathit{RegularStack}[X]$ and $\mathit{RepStack}[X]$, are identical. We wish to break this symmetry, however, by encapsulating the general stack behaviour in the $\mathit{RegularStack}[X]$ class and extending that behaviour in the $\mathit{RepStack}[X]$ class via inheritance. The refactoring rule can be applied immediately to the specification at the end of Section 4.1 to yield an altered definition for *RepStack*$[X]$:

$$
\begin{array}{|l}
\hline
\textit{RepStack}[X] \\
\upharpoonright(\textit{Push}, \textit{Pop}, \textit{INIT}) \\
\textit{RegularStack}[X][\textit{SuperL}/L, \textit{SuperLIMIT}/\textit{LIMIT}, \textit{SuperPush}/\textit{Push}, \\
\qquad\qquad \textit{SuperPop}/\textit{Pop}, \textit{SuperItems}/\textit{items}, \textit{SuperInit}/\textit{INIT}] \\
L == X \\
\begin{array}{|l}
\textit{LIMIT} : \mathbb{N} \\
\hline
\textit{LIMIT} = 1024
\end{array} \\
\begin{array}{|l}
\hline
\textit{items} : \operatorname{seq} L \\
\hline
\#\textit{items} \leq \textit{LIMIT} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{INIT} \\
\textit{items} = \langle\, \rangle \\
\hline
\end{array} \\
\begin{array}{|l}
\hline
\textit{Push} \\
\Delta(\textit{items}) \\
\textit{item}? : L \\
\textit{status}! : \mathbb{Z} \\
\hline
\#\textit{items} < \textit{LIMIT} \Rightarrow \textit{status}! >= 0 \land \textit{items}' = \langle \textit{item}? \rangle \frown \textit{items} \\
\#\textit{items} \geq \textit{LIMIT} \Rightarrow \textit{status}! < 0 \land \textit{items}' = \textit{items} \\
\hline
\end{array} \\
\textit{Pop} \mathrel{\widehat=} [\, \Delta(\textit{items})\ \textit{item}! : L \mid \textit{items} = \langle \textit{item}! \rangle \frown \textit{items}' \,] \\
\hline
\end{array}
$$

RepStack$[X]$ is then capable of being refined to utilise the definitions inherited by *RegularStack*$[X]$. As the features are identical, the refinement step is trivial:

$$
\begin{array}{|l}
\hline
\textit{RepStack}[X] \\
\upharpoonright(\textit{Push}, \textit{Pop}, \textit{INIT}) \\
\textit{RegularStack}[X] \\
\hline
\end{array}
$$

The desired definition of *RepStack*$[X]$ can be derived through refinement from the above definition, by strengthening the postcondition of the *Push* operation (by reducing non-determinism over the output variable *status*!).

$$
\begin{array}{|l}
\hline
\textit{RepStack}[X] \\
\upharpoonright(\textit{Push}, \textit{Pop}, \textit{INIT}) \\
\textit{RegularStack}[X] \\
\begin{array}{|l}
\hline
\textit{Push} \\
\#\textit{items} < \textit{LIMIT} \Rightarrow \textit{status}! = \#\textit{items}' \\
\#\textit{items} \geq \textit{LIMIT} \Rightarrow \textit{status}! = -\textit{LIMIT} \\
\hline
\end{array} \\
\hline
\end{array}
$$

6 Completeness

In the previous sections we introduced three rules for structural refactoring of Object-Z specifications: Introduce Generic Parameter, Introduce Inheritance and Introduce Polymorphism. In [9], McComb also introduced Annealing, which partitions a class state by introducing an instance to a fresh class definition. These four rules are minimal in the sense that they each operate upon one of the four orthogonal aspects of the object-oriented paradigm: no rule (or sequence of rules) can perform the function of any other rule.

We present an argument that the combination of these four rules with compositional class refinement [12] yields a complete method for design in Object-Z. That is, we demonstrate that it is possible to move from any structural design in Object-Z to any other design that represents a valid refinement of the original. However, we rely on some assumptions which we believe to be reasonable concerning the "system" class, i.e., the class describing the whole system (*System* in the example). First, we assume that the system class is not parameterised. In specifications where parameters exist over the system class, we expect these parameters to be instead expressed as given types in the specification. We also assume that the system class does not directly expose state variables via its visibility-list (although other classes may do this). This is because we cannot reason about the use of such variables, as the context of the system class is unknown, and this constrains possibilities for compositional class refinement. We expect accessor operations to be utilised in these circumstances.

Recall that each of the rules is an equivalence transformation that acts on the structure of a specification. We take advantage of this to demonstrate completeness by applying a commuting argument based on reduction. The reduction argument is as follows: if a sequence of rule applications exists that can reduce any arbitrary structure down to a canonical form, then it follows that the original structure can be *constructed* from that canonical form by the reverse application of the rules. By a commuting argument it follows that if a refinement ordering exists over the reduced form, then this ordering exists over arbitrary specifications, as all specifications are reducible to this form. Consequently, from any specification structure it is possible to construct another specification, with different structure, that is a valid refinement.

$$\begin{tikzcd}
(\mathit{System}, C_1, \ldots, C_n) \arrow[r, "\equiv"] \arrow[d, dashed, "\sqsubseteq"'] & (\mathit{System}, \mathit{Delegate}) \arrow[d, "\sqsubseteq"] \\
(\mathit{System}', D_1, \ldots, D_m) & (\mathit{System}', \mathit{Delegate}') \arrow[l, "\equiv"']
\end{tikzcd}$$

Fig. 5. Commuting argument for design reduction and construction with refinement

Figure 5 illustrates this schematically. A designer wishing to refactor a specification $(\mathit{System}, C_1, \ldots, C_n)$ to a specification $(\mathit{System}', D_1, \ldots, D_m)$ that is a refinement of or equivalent to $(\mathit{System}, C_1, \ldots, C_n)$ may do so by reducing $(\mathit{System}, C_1, \ldots, C_n)$ to two classes $(\mathit{System}, \mathit{Delegate})$ (our reduced form); refining *System* and/or *Delegate* to derive $(\mathit{System}', \mathit{Delegate}')$; and then constructing $(\mathit{System}', D_1, \ldots, D_m)$ from $(\mathit{System}', \mathit{Delegate}')$.

We progress in five stages. The first stage applies an equivalence preserving refinement step to all classes that inherit features of other classes. This allows us to establish the precondition for applying the Introduce Inheritance rule in reverse. Through this process we effectively remove all inheritance relationships in the specification. The second stage introduces a *Delegate* class that is key to our reduced form. The third stage applies the Introduce Generic Parameter rule in reverse. The precondition for applying this rule reversal is satisfied by forward applications of the Introduce Polymorphism rule. Through this process we show that all generic parameterisation can be removed from the specification. In the fourth stage, we apply the Introduce Polymorphism rule in reverse to collapse the entire specification (except for the *System* class) into the *Delegate* class. Again, we use equivalence preserving class refinements to satisfy the preconditions for applying this rule backwards. The last stage presents the argument that *System* and *Delegate* can be compositionally decoupled [12], and all possible specification refinements can be expressed as class refinements over these classes.

We refer to these stages respectively as: inheritance hierarchy flattening, interface isolation, parameterisation reduction, class paragraph definition reduction, and specification refinement. Below we expand on these stages to provide a high-level completeness argument; a formal proof can be found in [10].

1. Inheritance hierarchy flattening
   (a) Through equivalence preserving class refinements, references to features of superclasses can be removed.
   (b) Given subclasses do not reference features of their superclasses, the precondition for applying Introduce Inheritance in reverse is satisfied for all inheritance relationships.
   (c) Repeated reverse application of Introduce Inheritance removes all inheritance relationships.
2. Interface isolation
   (a) Since there are no inheritance relationships and the system class has no parameters, a fresh class definition $\mathit{Delegate}_1$ can be created via application of the Annealing rule on the system class [9]. Annealing partitions the state of a class into two parts: S and T, where T is the part that is moved into the newly created component class. In this case, the system class state is partitioned such that T represents the entirety of the state. That is, the entire state of the system class is moved to the new component class $\mathit{Delegate}_1$.
3. Parameterisation reduction (only necessary if classes with generic parameters exist)

   (a) For an arbitrary class definition that has generic parameters, all concrete types used to instantiate a parameter of that class can be enumerated.
   (b) For some class $C[P_1, \ldots, P_n]$ with a generic parameter P_i $(1 \leq i \leq n)$, an equivalence step can introduce a set of alias class names $\mathit{AliasT}_1[P_1, \ldots, P_n]$, ..., $\mathit{AliasT}_m[P_1, \ldots, P_n]$ (axiomatically defined): one member for each different concrete type $T_1, \ldots, T_m$ used to instantiate P_i.
   (c) Since there are no inheritance relationships, the precondition for INTRODUCE POLYMORPHISM is satisfied for any class C and the alias classes $\mathit{AliasT}_1[P_1, \ldots, P_n]$, ..., $\mathit{AliasT}_m[P_1, \ldots, P_n]$ from the previous step.
   (d) Application of INTRODUCE POLYMORPHISM to each class C and its aliases establishes, in each case, the precondition for the reverse application of INTRODUCE GENERIC PARAMETER to each alias class for the chosen parameter P_i. An equivalence preserving data refinement to introduce a local definition of the form $L == X$ may be necessary.
   (e) Reverse application of INTRODUCE GENERIC PARAMETER to all alias classes strictly reduces the number of generic parameters (by exactly 1).
   (f) Repeating steps 3(a) through 3(e) removes all generic parameterisation for all classes, given that the system class is not parameterised.
4. Class paragraph definition reduction
   (a) Let $n \leftarrow 1$.
   (b) For any class definition C where C is not the system class or $\mathit{Delegate}_n$, both C and $\mathit{Delegate}_n$ can undergo an equivalence preserving class refinement such that each definition has identical features.
   (c) The precondition for the reverse application of INTRODUCE POLYMORPHISM is satisfied for a fresh class $\mathit{Delegate}_{n+1}$, which is defined to be the class union of C and $\mathit{Delegate}_n$.
   (d) Reverse application of INTRODUCE POLYMORPHISM to C and $\mathit{Delegate}_n$ yields a strict reduction in the number of classes.
   (e) Let $n \leftarrow n + 1$.
   (f) Repeating steps 4(b) through 4(e) eventually yields the reduced form, i.e., the only classes are the system class and $\mathit{Delegate}_n$.
5. Specification refinement
   (a) The $\mathit{Delegate}_n$ class can be compositionally decoupled from the system class, such that both may be refined, assuming the system class does not expose state variables in its interface [12]. Refinements to the system class and $\mathit{Delegate}_n$ constitute all possible specification refinements.

7 Conclusion

We have emphasised the completeness of the approach for manipulation of architectural *structure* (that is, high-level design), in an object-oriented sense. Unless a light-weight approach is employed, whereby individual class specifications — having undergone sufficient data refinement — are implemented (and tested) informally, there is also potential for future work in providing a rigorous method for implementing specifications of individual classes in an object-oriented programming language. As Object-Z is not wide-spectrum, this would require an extension to the language, or a mapping of our method to another object-oriented specification language that supports low-level programming constructs.

In ongoing work, Ruhroth et al. are investigating refactoring transformations of Object-Z specifications in the presence of CSP [13], particularly with respect to those refactoring transformations we refer to as non-structural. Other work by Estler et al. incorporates model-checking technologies for the verification of refactorings in Object-Z without CSP [4]. The automation of such verifications are important for the practicality and relevance of our approach to current software engineering practice, and can hopefully be extended to our structural rules.

Acknowledgements

We acknowledge the support of Australian Research Council (ARC) Discovery Grant DP0558408.

References

1. Java 2 Platform Standard Edition 5.0, `http://java.sun.com/j2se/1.5.0/guide/`.
2. P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornelio. Algebraic Reasoning for Object-Oriented Programming. *Sci. Comput. Program.*, 52(1-3):53–100, 2004.
3. J. Derrick and E. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* FACIT Series. Springer-Verlag, 2001.
4. H.-C. Estler, T. Ruhroth, and H. Wehrheim. Modelchecking correctness of refactorings – some experiments. *ENTCS*, 187:3–17, 2007.
5. R. Gheyi and P. Borba. Refactoring Alloy specifications. *ENTCS*, 95:227–243, 2004.
6. D. Jackson. Alloy: a lightweight object modelling notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.
7. K. Lano. *Formal Object-oriented Development.* Springer-Verlag, 1995.
8. K. Lano and S. Goldsack. Refinement of Distributed Object Systems. In E. Najm and J.-B. Stefani, editors, *Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 99–114. Chapman and Hall, March 1996.
9. T. McComb. Refactoring Object-Z Specifications. In M. Wermelinger and T. Margaria-Steffen, editors, *FASE '04: Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 69–83. Springer-Verlag, 2004.
10. T. McComb. *Formal Derivation of Object-Oriented Designs.* PhD thesis, The University of Queensland, 2007.
11. T. McComb and G. Smith. Architectural Design in Object-Z. In P. Strooper, editor, *ASWEC '04: Australian Software Engineering Conference*, pages 77–86. IEEE Computer Society Press, 2004.
12. T. McComb and G. Smith. Compositional class refinement in Object-Z. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 205–220. Springer, 2006.

13. T. Ruhroth and H. Wehrheim. Refactoring object-oriented specifications with data and processes. In M. M. Bonsangue and E. B. Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *LNCS*, pages 236–251. Springer, 2007.
14. G. Smith. *The Object-Z Specification Language.* Kluwer, 2000.
15. B. Stroustrup. *The C^{++} Programming Language, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
16. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement.* Prentice Hall International Series in Computer Science, 1996.