SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-27

Reasoning about Object-Z specifications

Graeme Smith

September 1995

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

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

Reasoning about Object-Z specifications

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

Abstract

This paper presents a method of reasoning about Object-Z specifications. The approach utilises the modularity inherent in Object-Z specifications to simplify proofs. Properties proved for a class in isolation can be used when that class is either inherited by another class or instantiated as part of a system of interacting objects. Proofs using structural induction and the notion of object integrity are discussed.

1 Introduction

Formal reasoning about specifications enables the specifier to gain confidence in both the correctness of a specification and the behaviour of the specified system. It is also essential in the process of refinement of a specification towards code. To cope with the complexities in specifications of large-scale systems such reasoning needs to be carried out in a modular fashion. This has lead to the development of formal specification languages which support modular structuring techniques.

Object-Z[2, 6, 7] is an extension to the formal specification language Z[1, 10, 5][1] which facilitates specification in an object-oriented style. Enhanced structuring is provided through the addition to Z of *classes*. A class encapsulates types, constants and variables describing the state of particular objects in the system with schemas describing their initial states and the operations which they can undergo. It is not merely a syntactic extension but also defines a type whose instances are references to objects of the class.

Classes in Object-Z can be used to define other classes in two ways: by *inheritance* or *instantiation*. Inheritance allows a class to be defined as an extension or specialisation of another class by a process of incremental modification. In this way, the repetition of structure between classes can be avoided. Instantiation allows classes of objects which refer to other objects to be specified. This enables the specification of systems of interacting objects. Both methods enable the construction of complex systems structured in terms of simpler components.

The purpose of this paper is to investigate and propose guidelines for modular reasoning about Object-Z specifications. It builds on the work on a logic for Object-Z presented in [8] and [9]. The logic is an extension of $\mathcal{W}$[14], a logic advocated for Z and used as the basis for the deductive system in the Z Base Standard[1]. The rules of the logic provide an axiomatic semantics of the language constructs in Object-Z. This paper introduces further rules

[1]This paper assumes a familiarity with Z. Readers unfamiliar with Z may wish to consult [10] or [5].

which facilitate reasoning in a modular fashion. Section 2 introduces the notion of classes in Object-Z and explains the basic concepts of the logic. Section 3 examines the issues underlying reasoning about inheritance in Object-Z and Section 4 looks at instantiation. A case study is used to illustrate the proposed approach to reasoning in both sections. Section 5 looks at issues requiring further research; the problems of aliasing and reasoning about liveness properties.

2 Classes

Classes provide a means of encapsulating state information with its associated operations. By adding structure to specifications they facilitate reasoning in a modular fashion. The properties of an individual class can be deduced in isolation from the system in which it occurs. The properties of the system can then be derived from the properties of its classes.

As an example of a class in Object-Z, consider the following specification of a generic queue whose maximum length is less than or equal to 10.

$Queue[T]$

$Length == 1 \mathrel{..} 10$

$max : Length$

$items : \operatorname{seq} T$
Δ
$size : \mathbb{N}$

$size = \# items$
$size \leqslant max$

$\textit{INIT}$

$items = \langle\, \rangle$

$Join$

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

$items' = items \frown \langle item? \rangle$

$Leave$

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

$items = \langle item! \rangle \frown items'$

The class has a single generic type T corresponding to the type of the items in the queue. It has a local type $Length$ corresponding to the set of permissible queue lengths of which the local constant max is an instance. The value of max can vary for different objects of the class but is unchanged by any class operation.

The state schema of the class declares a *primary* state variable *items* representing the sequence of items in the queue and a *secondary* state variable *size* representing the size of the queue. Primary state variables may only change when they are included in the *delta-list* of an operation. Secondary state variables are always available for change. They may be changed by operations of their class as well as operations of other classes in which their class is instantiated.

The state schema's predicate relates *size* to *items* and ensures that the size of the queue never exceeds *max*. This predicate is implicitly true in the initial state and before and after each operation.

The state variables *items* and *size* are implicitly available in the initial state schema INIT which states that the queue is initially empty. They are also implicitly available in each operation along with the variables $items'$ and $size'$ denoting the value of the state variables *items* and *size* respectively in the post-state of the operation. The operation *Join* allows a new item to be added to the end of a queue (? denotes an input variable) and the operation *Leave* allows an item to be removed from the head of a queue (! denotes an output variable). Both operations change the primary state variable *items* and hence include it in their delta-lists.

2.1 A logic for Object-Z

A logic for reasoning about Object-Z specifications is presented in [8] and [9]. The logic is a Gentzen-style sequence calculus. Axioms and theorems are expressed using *sequents* and further theorems are derived by the application of *inference rules*.

Sequents

Sequents are of the form

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

where A is a class name, d is a list of declarations and Ψ and Φ are lists of predicates. A sequent is *valid* if at least one of the predicates in Φ is true in the local environment of A (i.e. the environment defined by A's local types and constants) enriched by the declarations d and satisfying all predicates in Ψ.

For example, given the class $Queue[T]$ above and an actual type t, the following is a valid sequent.

$$Queue[t] :: \quad x : Length \mid x < max \vdash x \in 1 \mathrel{..} 9$$

That is, in the environment of class $Queue[t]$, given that $x : Length$ and that $x < max$ we can deduce that $x \in 1 \mathrel{..} 9$.

Inference rules

Inference rules for manipulating sequents are of the form

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

where the *premisses* are zero or more sequents and the *conclusion* is a single sequent. The *name* identifies the rule and the *proviso* is a predicate which must be true for the rule to be applicable. A rule is *sound* if, whenever the proviso is satisfied and the premisses are valid, the conclusion is also valid.

The logic for Object-Z provides both general rules defining constructs in Object-Z and rules specific to a particular specification. Both types of rules are presented in a schematic form which must be instantiated before they are used. To facilitate the expression of rules, meta-functions which derive information from the specification are used. For example, the meta-function η returns a predicate representing the properties of the local environment of a class. For $Queue[T]$ above and an actual type t, we have[2]

$$\eta(Queue[t]) \equiv Length = 1 \mathinner{..} 10 \wedge max \in Length$$

2.2 Rules for classes

General rules

A general rule which uses the meta-function η is *LocalDefs*. It states that when we are reasoning within the local environment of a class, we can assume the predicate representing its properties.

$$\frac{A[t_1, \ldots, t_n] :: \quad \eta(A[t_1, \ldots, t_n]) \vdash}{A[t_1, \ldots, t_n] :: \ \vdash} \quad (\ \mathit{LocalDefs}\ )$$

For notational convenience, rules (such as the one above) are written with declarations and predicates common to the conclusion and those premisses which refer to the same class as the conclusion elided. Thus, *LocalDefs* could be used to prove the sequent from the previous section as follows.

$$\frac{Queue[t] :: \quad x : Length \mid x < max, \eta(Queue[t]) \vdash x \in 1 \mathinner{..} 9}{Queue[t] :: \quad x : Length \mid x < max \vdash x \in 1 \mathinner{..} 9}$$

Since $\eta(Queue[t]) \equiv Length = 1 \mathinner{..} 10 \wedge max \in Length$, the premiss of this rule is obviously valid. (This would be proved in the logic by application of general rules corresponding to the rules of first-order predicate calculus.) Therefore, the conclusion must also be valid.

Specification-specific rules

The specification-specific rules for a class provide definitions of the schemas of the class. For example, given a class $A[X_1, \ldots, X_n]$ with state schema

$$\begin{array}{|l}
d_1 \\
\Delta \\
d_2 \\
\hline
p
\end{array}$$

[2] $\equiv$ is meta-equivalence.

and the actual types $t_1, \ldots, t_n$ we have the following rule.

$$\frac{A[t_1, \ldots, t_n] :: \quad \mathit{S\scriptstyle TATE} = [b \odot d_1;\ b \odot d_2 \mid b \odot p] \vdash}{A[t_1, \ldots, t_n] :: \ \vdash} \quad (\ \mathit{StateDef}\ )\ [\ q\ ]$$

where $q \equiv b = \langle\!|\ X_1 \rightsquigarrow t_1, \ldots, X_n \rightsquigarrow t_n\ |\!\rangle$

The keyword *State* is introduced to refer to the state schema of the class $A[t_1, \ldots, t_n]$. It is equated to the schema formed by replacing all occurrences of generic types X_i in the declarations and predicate of the state schema of $A[X_1, \ldots, X_n]$ by actual type t_i using the substitution operator $\odot$. This rule is intended to be used in a similar fashion to the rule *LocalDefs*, i.e. common declarations and predicates have been elided.

Instantiating *StateDef* for the class $Queue[t]$ gives the following.

$$\frac{Queue[t] :: \quad \mathit{S\scriptstyle TATE} = [items : \mathrm{seq}\, t;\ size : \mathbb{N} \mid size = \# items \wedge size \leqslant max] \vdash}{Queue[t] :: \ \vdash}$$

Similar rules exist for the initial state schema and operations.

2.3 Reasoning about classes

The kinds of properties we would like to prove for classes usually relate to one of the class' schemas. That is, we would like to prove properties about the class' state, initial state or operations. For example, suppose we want to show that $size \leqslant max$ in the state of class $Queue[t]$. The required theorem is expressed as

$$Queue[t] :: \quad \mathit{S\scriptstyle TATE} \vdash size \leqslant max$$

The schema *State* in this sequent is a declaration. To prove the sequent, we apply a number of backward (goal-directed) proof steps. Each step corresponds to a rule (or sequence of rules) of the logic. The first, named $d \mid$ in the Z Base Standard, introduces the predicate of the schema, denoted [*State*], i.e. the expression *State* within schema brackets.

$$Queue[t] :: \quad \mathit{S\scriptstyle TATE} \mid [\mathit{S\scriptstyle TATE}] \vdash size \leqslant max$$

Next, the rule defining the state schema of $Queue[t]$ in the previous section is used to obtain the following.

$$\begin{aligned} Queue[t] :: \quad & \mathit{S\scriptstyle TATE} \mid \mathit{S\scriptstyle TATE} = [items : \mathrm{seq}\, t;\ size : \mathbb{N} \mid size = \# items \wedge size \leqslant max], \\ & [\mathit{S\scriptstyle TATE}] \vdash size \leqslant max \end{aligned}$$

The next step involves replacing the expression *State* in the predicate [*State*] with its definition. This is actually achieved by a sequence of proof steps which form the derived rule *ApplyLeibniz* (see the Appendix).

$$Queue[t] :: \quad \mathit{S\scriptstyle TATE} \mid [[items : \mathrm{seq}\, t;\ size : \mathbb{N} \mid size = \# items \wedge size \leqslant max]] \vdash size \leqslant max$$

Finally, the schema in the predicate part of the sequent is converted to a predicate using the rules $\vdash [|]$, $\vdash [D;\ D]$ and $\vdash [n : s]$ of the Z Base Standard. The resulting sequent is obviously valid.

$$Queue[t] :: \quad \textit{STATE} \mid items \in \operatorname{seq} t \land size \in \mathbb{N} \land size = \# items \land size \leqslant max \vdash size \leqslant max$$

Other class properties such as those proved by structural induction are only relevant when reasoning about instances of the class. This is discussed in Section 4.

3 Inheritance

Inheritance is a mechanism for incrementally modifying a class to produce a new class. The new class is either an extension or specialisation of the original class in the sense that any properties of the schemas of the original class are also properties of the corresponding schemas of the new class.

As an example of inheritance in Object-Z, consider the following specification of a queue which has a counter which records the number of items which have been joined to it.

CQueue[T]

> $Queue[T]$
>
> $count : \mathbb{N}$
>
> *INIT*
> $count = 0$
>
> *Join*
> $\Delta(count)$
> ———
> $count' = count + 1$

The understanding is that the definitions of $Queue[T]$ are *merged* with those in $CQueue[T]$. The type *Length*, the constant *max* and the operation *Leave* of $CQueue[T]$ are equivalent to those defined in $Queue[T]$. If there had been any local types or constants with the same name occurring in both the inherited and inheriting classes they would have been semantically identified. The state schema, initial state schema and common-named operations are effectively conjoined. Hence, the state schema, initial state schema and operation *Join* of $CQueue[T]$ are equivalent to the following.

	INIT	*Join*
$items : \operatorname{seq} T$ $count : \mathbb{N}$ Δ $size : \mathbb{N}$ ——— $size = \# items$ $size \leqslant max$	$items = \langle\,\rangle$ $count = 0$	$\Delta(items, count)$ $item? : T$ ——— $items' = items \frown \langle item? \rangle$ $count' = count + 1$

Object-Z also supports multiple inheritance. Semantically, it is identical to progressively inheriting each of the inherited classes in an arbitrary order.

3.1 Rules for inheritance

General rules

The inherited part of a schema is defined as the conjunction of the same-named schemas in each inherited class. Before the conjunction takes place, each schema must be fully expanded so that it does not include references to other schemas of its class.

This is necessary because it is semantic, and not syntactic, information which is inherited in Object-Z. For example, consider the following classes where d, d_1 and d_2 are declarations and p, p_1, p_2 and q_1 are predicates.

A

- State schema: d | p
- Op_1: d_1 | p_1
- Op_2: $d2$ | p_2
- $Op_3 \mathrel{\widehat=} Op_1 \wedge Op_2$

B

- A
- Op_1: q_1

Class B inherits class A and extends the operation Op_1 with the predicate q_1. The meaning of Op_3 in B is identical to that in A, i.e. Op_3 is the conjunction of A's Op_1 and Op_2, not B's Op_1 and Op_2.

The choice of semantic, as opposed to syntactic, inheritance is necessary for defining modular refinement. That is, we would like to be able to refine a class such as A in isolation and by doing so produce a refinement of the entire specification. One way to refine A would be to replace the definition of Op_3 by the following.

Op_3

d_1
d_2

---

p_1
p_2

If inheritance is semantic, the meaning of B is not changed. However, if inheritance was syntactic the meaning of B would change as it would no longer include the predicate q_1.

The rule defining the inherited state schema (denoted $\uparrow\textit{State}$) is as follows.

$$\frac{\begin{array}{l} A_1 :: \ \vdash \textit{State} = S_1 \\ \ldots \\ A_n :: \ \vdash \textit{State} = S_n \end{array}}{B :: \ \vdash \uparrow\textit{State} = S_1 \wedge \ldots \wedge S_n} \quad (\uparrow\textit{State}) \; [\, p \,]$$

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

The first condition of the proviso uses the meta-function ι which returns the set of classes inherited by a class. Hence, it states that the set of classes inherited by class B is $\{A_1, \ldots, A_n\}$. The other provisos use the meta-function ϕ which returns the free variables of a schema or predicate and the meta-function α which returns the set of types, constants and state variables of a class. These provisos ensure that the schemas S_i are expressed in fully expanded form (i.e. in terms of $\alpha(A_i)$ only and not in terms of references to the state, initial state or operations of A_i).

Similar rules exist for the inherited initial state schema and operations.

Specification-specific rules

A specification-specific rule defining the state schema of a class was given in Section 2.2. For classes which inherit other classes, this rule is extended as follows.

$$\frac{A[t_1, \ldots, t_n] :: \quad \textsc{State} = [\uparrow\textsc{State};\ b \odot d_1;\ b \odot d_2 \mid b \odot p] \vdash}{A[t_1, \ldots, t_n] :: \ \vdash} \quad (\ \mathit{StateDef}\ )\ [\ q\ ]$$

$$\text{where } q \equiv b = \langle\!|\ X_1 \leadsto t_1, \ldots, X_n \leadsto t_n\ |\!\rangle$$

Instantiating this rule for $CQueue[t]$ gives the following.

$$\frac{CQueue[t] :: \quad \textsc{State} = [\uparrow\textsc{State};\ count : \mathbb{N}] \vdash}{CQueue[t] :: \ \vdash}$$

Similar rules exist for the initial state schema and operations.

3.2 Reasoning about inheritance

Suppose we wished to prove that $size \leqslant max$ in the state of class $CQueue[t]$. That is,

$$CQueue[t] :: \quad \textsc{State} \vdash size \leqslant max$$

One way to do this would be to use the rule $\uparrow\textsc{State}$ and the rules defining the state schemas of $Queue[t]$ and $CQueue[t]$. However, since inheritance preserves properties of the schemas of the original class a more modular approach is to use a rule such as the following.

$$\frac{A :: \quad \textsc{State} \vdash p}{B :: \quad \textsc{State} \vdash p} \quad (\ \mathit{InheritStateProperty}\ )\ [\ A \in \iota(B) \wedge \phi(p) \subseteq \alpha(A)\ ]$$

That is, any property provable from the state schema of A is provable from the state schema of B provided B inherits A and the property is expressed in terms of the types, constants and state variables of A. This is true because inheritance involves only the addition of declarations and predicates and not their removal.

By backward application of this rule to the above property we have

$$Queue[t] :: \quad \text{S}{\small TATE} \vdash size \leqslant max$$

which was proved in Section 2.3. Thus, using *InheritStateProperty*, we can prove the desired property of $CQueue[t]$ directly from a property proved for $Queue[t]$.

Similarly, we introduce rules for the initial state schema and operations.

$$\frac{A :: \quad \text{I}{\small NIT} \vdash p}{B :: \quad \text{I}{\small NIT} \vdash p} \quad (\ \mathit{InheritInitProperty}\ )\ [\ A \in \iota(B) \wedge \phi(p) \subseteq \alpha(A)\ ]$$

$$\frac{A :: \quad Op \vdash p}{B :: \quad Op \vdash p} \quad (\ \mathit{InheritOpProperty}\ )\ [\ q\ ]$$

where $q \equiv A \in \iota(B)$
$\{x_1, \ldots, x_n\} = \alpha_1(A) \cup \alpha_2(A)$
$\phi(p) \subseteq \alpha(A) \cup \{x_1', \ldots, x_n'\}$

The meta-functions α_1 and α_2 return the primary and secondary variables of a class respectively. The properties of an operation which are preserved by inheritance may be expressed in terms of primed form of these variables.

It is important to note that properties of a class derived by structural induction do not necessarily carry over to a class which inherits it. For example, the property $count \geqslant \#items$ can be proved for $CQueue[T]$ by structural induction (see Section 4.2). This property is not true, however, for all classes which inherit $CQueue[T]$. For example, it is not true for the following class of queues whose counters can be reset.

$RCQueue[T]$

$CQueue[T]$

$Reset$

$\Delta(count)$

$count' = 0$

4 Instantiation

Instances of a class can be used as state variables. Such instances refer to objects which possess all properties of the class including those derived by methods such as structural induction. Furthermore, such objects can only be changed by the operations of the class. This latter property is known as *object integrity*.

As an example of instantiation in Object-Z, consider the following specification of a message distributor. The distributor sends messages along one of two routes depending on the amount of traffic already sent on those routes. MSG is a global type corresponding to the set of all messages.

$$
\begin{array}{|l}
\hline
\textit{MsgDistributor} \\
\begin{array}{|l}
\hline
route1, route2 : CQueue[MSG] \\
\hline
route1 \neq route2 \\
\hline
\end{array} \\
\\
\begin{array}{|l}
\hline
\textsc{Init} \\
route1.\textsc{Init} \wedge route2.\textsc{Init} \\
\hline
\end{array} \\
\\
Join1 \mathrel{\widehat=} [route1.count - route2.count < 10] \wedge route1.Join \\
Join2 \mathrel{\widehat=} [route2.count - route1.count < 10] \wedge route2.Join \\
Leave1 \mathrel{\widehat=} route1.Leave \\
Leave2 \mathrel{\widehat=} route2.Leave \\
\hline
\end{array}
$$

The class has two state variables, $route1$ and $route2$, denoting references to distinct objects of the class $CQueue[MSG]$. The dot notation is used to initialise and apply operations to the referenced objects. The initial state schema states that initially, each referenced object satisfies its initial state, i.e. each of the queues is empty.

The operation $Join1$ applies the operation $Join$ to the object referenced by $route1$ provided that the count of messages already sent on $route1$ doesn't already exceed the count of messages sent on $route2$ by 10 or more. The operation $Leave1$ applies the operation $Leave$ to the object referenced by $route1$. The operations $Join2$ and $Leave2$ are interpreted in a similar fashion.

4.1 Rules for instantiation

To express the properties of an object in terms of the properties of its class, a meta-substitution operator $\circledast$ is introduced. Given an object reference a, the notation $a \circledast p$ is equivalent to the predicate p with all occurrences of x, where x is a type, constant or variable (in either primed or unprimed form) of a's class, replaced by $a.x$. For example, $route1 \circledast (size \leqslant max)$ is equivalent to $route1.size \leqslant route1.max$.

The meta-substitution operator is used in general rules such as the following which states that an object is a member of a class A when its types, constants and variables satisfy the properties of A's local environment and the predicate of A's state schema (in both primed and unprimed form).

$$
\frac{\begin{array}{l} \vdash a \in A \\ A :: \ \vdash \textsc{State} = S \end{array}}{\vdash a \circledast (\eta(A) \wedge \Delta S)} \quad (\ \textit{ClassMembership}\ )\ [\ \phi(S) \subseteq \alpha(A)\ ]
$$

Similar rules hold for the initial state schema and operations of a class providing definitions of $a.\textsc{Init}$ and $a.Op$. See [8] for details.

4.2 Reasoning about instantiation

As with inheritance, we can introduce rules more suitable for modular reasoning than those already defined in the logic. For example, to prove the sequent

$$MsgDistributor :: \quad \textsc{State} \vdash route1.size \leqslant route1.max$$

is valid we could use the following rule.

$$\frac{\begin{array}{l}\vdash a \in A \\ A :: \quad \textsc{State} \vdash p\end{array}}{\vdash a \circledast p} \quad (\ PromoteStateProperty\ )\ [\ \phi(p) \subseteq \alpha(A)\ ]$$

In this rule, as well as common declarations and predicates being elided, the common class of the conclusion and first premiss is elided. An instantiation of the rule to prove the desired property of $MsgDistributor$ is as follows.

$$\frac{\begin{array}{l}MsgDistributor :: \quad \textsc{State} \vdash route1 \in CQueue[MSG] \\ CQueue[MSG] :: \quad \textsc{State} \vdash size \leqslant max\end{array}}{MsgDistributor :: \quad \textsc{State} \vdash route1.size \leqslant route1.max}$$

Similarly, to prove properties involving the initialisation and application of operations to objects, the following rules are introduced.

$$\frac{\begin{array}{l}\vdash a \in A \\ A :: \quad \textsc{Init} \vdash p\end{array}}{\vdash a.\textsc{Init} \Rightarrow a \circledast p} \quad (\ PromoteInitProperty\ )\ [\ \phi(p) \subseteq \alpha(A)\ ]$$

$$\frac{\begin{array}{l}\vdash a \in A \\ A :: \quad [Op \mid x_1' = x_1 \wedge \ldots \wedge x_m' = x_m] \vdash p\end{array}}{a.Op \vdash a \circledast p} \quad (\ PromoteOpProperty\ )\ [\ q\ ]$$

$$\text{where } q \equiv \begin{array}{l}\{x_1, \ldots, x_m\} = \alpha_1(A) \setminus \delta(A, Op) \\ \{x_1, \ldots, x_n\} = \alpha_1(A) \cup \alpha_2(A) \\ \phi(p) \subseteq \alpha(A) \cup \{x_1', \ldots, x_n'\}\end{array}$$

The first condition of the proviso of $PromoteOpProperty$ uses the meta-function δ which returns the variables in the delta-list of an operation. The rule uses an extended form of the operation Op with a predicate stating that all primary variables of the class A not in Op's delta-list are unchanged.

Induction

Properties of an object not involving secondary variables of the object's class can be proved by structural induction by showing that

- they are true in the initial state of the object's class and

- if they are true in the pre-state of an operation of the object's class then they are also true in the post-state of the operation.

This is captured by the following rule.

$$\frac{\begin{array}{l} \vdash a \in A \\ A :: \quad \textit{I}\textsc{nit} \vdash p \\ A :: \quad [Op_1 \mid x1'_1 = x1_1 \wedge \ldots \wedge x1'_{m1} = x1_{m1}] \\ \qquad\qquad \vdash p \Rightarrow \pi(\langle\!| \, x_1 \leadsto x'_1, \ldots, x_m \leadsto x'_m \, |\!\rangle \odot p) \\ \ldots \\ A :: \quad [Op_n \mid xn'_1 = xn_1 \wedge \ldots \wedge xn'_{mn} = xn_{mn}] \\ \qquad\qquad \vdash p \Rightarrow \pi(\langle\!| \, x_1 \leadsto x'_1, \ldots, x_m \leadsto x'_m \, |\!\rangle \odot p) \end{array}}{\vdash a \circledast p} \quad (\ \textit{Induction}\ )\ [\ q\ ]$$

$$\text{where } q \equiv \begin{array}[t]{l} \Omega(A) = \{Op_1, \ldots, Op_n\} \\ \{x_1, \ldots, x_m\} = \alpha_1(A) \\ \{x1_1, \ldots, x1_{m1}\} = \alpha_1(A) \setminus \delta(A, Op_1) \\ \ldots \\ \{xn_1, \ldots, xn_{mn}\} = \alpha_1(A) \setminus \delta(A, Op_n) \\ \phi(p) \subseteq \alpha(A) \setminus \alpha_2(A) \end{array}$$

The meta-function Ω returns the set of operations of a class. The meta-function α_2 returns the set of secondary variables of a class. To state that a predicate is true in the post-state of an operation, we replace all primary and secondary variables with their primed counterparts and then apply the meta-function π to the resulting predicate. This meta-function replaces occurrences of $a.x$ by $a.x'$ where x is a variable of a's class (see [9]).

As an example of the application of this operator. Consider proving that

$$\textit{MsgDistributor} :: \quad \textit{S}\textsc{tate} \vdash \textit{route1.count} \geqslant \#\textit{route1.items}$$

The rule would be instantiated as follows.

$$\frac{\begin{array}{ll} \textit{MsgDistributor} :: & \textit{S}\textsc{tate} \vdash \textit{route1} \in \textit{CQueue}[\textit{MSG}] \\ \textit{CQueue}[\textit{MSG}] :: & \textit{I}\textsc{nit} \vdash \textit{count} \geqslant \#\textit{items} \\ \textit{CQueue}[\textit{MSG}] :: & \textit{Join} \vdash \textit{count} \geqslant \#\textit{items} \Rightarrow \textit{count}' \geqslant \#\textit{items}' \\ \textit{CQueue}[\textit{MSG}] :: & [\textit{Leave} \mid \textit{count}' = \textit{count}] \vdash \textit{count} \geqslant \#\textit{items} \Rightarrow \textit{count}' \geqslant \#\textit{items}' \end{array}}{\textit{MsgDistributor} :: \quad \textit{S}\textsc{tate} \vdash \textit{route1.count} \geqslant \#\textit{route1.items}}$$

Each of the premisses can be shown to be valid, therefore the conclusion must also be valid.

Integrity

Objects can only undergo operations defined in their class. This property, known as object integrity, is useful when nothing else is known about the change an object undergoes during an operation. For example, in the class *MsgDistributor*, the operation *Join*1 describes explicitly what happens to the object referenced by *route*1 but says nothing about the object referenced by *route*2. The fact that *route*2 is not in the operation's delta-list means that the reference is unchanged, i.e. it still refers to the same object after the operation, but the object itself may have undergone some internal change of its variables. This is possible because another reference to the object referenced by $route2$ may exist in the system. This is known as *aliasing* and is discussed in Section 5.

The rule for object integrity needs to take into account the fact that a given object may undergo, not just an operation of its class, but a sequence of operations of its class. This is because Object-Z has a sequential composition operator $\fatsemi$ which enables us to write operations of the form $a.Op_1 \fatsemi a.Op_2$.

The integrity properties we can deduce about an object, therefore, are properties not involving secondary variables of the object's class which are true over any sequence of operations. For example, we could prove

$$MsgDistributor :: \quad \mathit{STATE} \vdash route1.count' \geqslant route1.count$$

That is, the $count$ variable of the object referenced by $route1$ is only ever increased. This is true since all operations of $CQueue[MSG]$ only increase $count$ or leave it unchanged and, given $count'' : \mathbb{N}$, $count' \geqslant count \wedge count'' \geqslant count' \Rightarrow count'' \geqslant count$. The latter condition ensures that the property is also true for any sequence of operations of $CQueue[MSG]$. The property is therefore proved as follows.

$$\frac{\begin{array}{ll} MsgDistributor :: & \mathit{STATE} \vdash route1 \in CQueue \\ CQueue[MSG] :: & Join \vdash count' \geqslant count \\ CQueue[MSG] :: & [Leave \mid count' = count] \vdash count' \geqslant count \\ CQueue[MSG] :: & [\Delta \mathit{STATE};\ count'' : \mathbb{N}] \vdash count' \geqslant count \wedge count'' \geqslant count' \Rightarrow \\ & \qquad\qquad\qquad\qquad\qquad\qquad count'' \geqslant count \end{array}}{MsgDistributor :: \quad \mathit{STATE} \vdash route1.count' \geqslant route1.count}$$

A schematic form of this rule is given below. To form the condition ensuring the predicate p holds over any sequence of operations, it is necessary to have another predicate p_1 in which all expressions of the form $a.x'$ have been replaced by a new variable u. Such expressions, referring to post-state values, cannot be renamed using the substitution operator $\odot$.

$$\frac{\begin{array}{l} \vdash a \in A \\ A :: \ \Delta \mathit{STATE} \vdash a_1 \in A_1 \wedge \ldots \wedge a_n \in A_n \\ A_1 :: \ \mathit{STATE} \vdash x_1 \in T_1 \\ \ldots \\ A_n :: \ \mathit{STATE} \vdash x_n \in T_n \\ A :: \ \mathit{STATE} \vdash y_1 \in T_{n+1} \wedge \ldots \wedge y_m \in T_{n+m} \\ A :: \ [Op_1 \mid y1'_1 = y1_1 \wedge \ldots \wedge y1'_{m1} = y1_{m1}] \vdash p \\ \ldots \\ A :: \ [Op_k \mid yk'_1 = yk_1 \wedge \ldots \wedge yk'_{mk} = yk_{mk}] \vdash p \\ u_1 : T_1;\ \ldots;\ u_n : T_n \mid u_1 = a_1.x'_1 \wedge \ldots \wedge u_n = a_n.x'_n \vdash p \Leftrightarrow p_1 \\ A :: \ [\Delta \mathit{STATE};\ u_1 : T_1;\ \ldots;\ u_n : T_n;\ z_1 : T_{n+1};\ \ldots;\ z_m : T_{n+m}] \vdash \\ \qquad\qquad p \wedge \pi(\langle\!|\ \lambda_1\ |\!\rangle \odot p_1) \Rightarrow \langle\!|\ \lambda_2\ |\!\rangle \odot p_1 \end{array}}{\vdash a \circledast p} \quad (\ Integrity\ )\ [\ q\ ]$$

$$
\begin{array}{ll}
\text{where } & q \equiv \rho(p) \rhd \mathit{WORD}' = \{(a_1, x_1), \ldots, (a_n, x_n)\} \\
& \rho(p_1) \rhd \mathit{WORD}' = \varnothing \\
& \Omega(A) = \{Op_1, \ldots, Op_k\} \\
& \alpha_1(A) = \{y_1, \ldots, y_m\} \\
& \{y1_1, \ldots, y1_{m1}\} = \alpha_1(A) \setminus \delta(A, Op_1) \\
& \ldots \\
& \{yk_1, \ldots, yk_{mk}\} = \alpha_1(A) \setminus \delta(A, Op_k) \\
& \phi(p) \subseteq \alpha(A) \setminus \alpha_2(A) \cup \{y_1', \ldots, y_m'\} \\
& \lambda_1 = y_1' \rightsquigarrow z_1, \ldots, y_m' \rightsquigarrow z_m, y_1 \rightsquigarrow y_1', \ldots, y_m \rightsquigarrow y_m' \\
& \lambda_2 = y_1' \rightsquigarrow z_1, \ldots, y_m' \rightsquigarrow z_m \\
& \{z_1, \ldots, z_m\} \cup \{u_1, \ldots, u_n\} = \varnothing \\
& \{z_1, \ldots, z_m, u_1, \ldots, u_n\} \cap \alpha(A) = \varnothing
\end{array}
$$

The meta-function ρ returns the set of expression/identifier pairs (a, x) corresponding to the expressions $a.x$ occurring in a predicate (see [9]). The proviso $\rho(p_1) \rhd \mathit{WORD}' = \varnothing$ ensures that p_1 has no expressions $a_i.x_i'$. That is, all occurrences of these expressions in p have been replaced in p_1 by $u_i.x_i$.

5 Future work

Aliasing

Aliasing occurs as a result of two or more object references referring to the same object. These references can occur either in the same class or in different classes which make up the system specification. The latter case restricts modular reasoning by reducing what we can deduce about a particular class in isolation. For example, consider the following where C is a class with operation Op.

A

$x, y : C$

$OpA \mathrel{\widehat=} x.Op$

B

$a : A$
$z : C$

$OpB \mathrel{\widehat=} z.Op \land a.OpA$

If, in class B, $a.y$ and z refer to the same object then, when OpB is performed, effectively both $a.x$ and $a.y$ undergo operation Op. Hence, when reasoning about the operation OpA in class A, we can't rely on the object referenced by y to remain unchanged. In fact, the most we can deduce is that if it does change, it undergoes a sequence of operations of its class.

If we wished to be more precise, we could reason using an exhaustive case analysis covering all the possible combinations of equalities and inequalities between object references in the specification. Since this would be impractical in even medium-sized specifications due to the large number of object references which could exist, a means of limiting and controlling aliasing is seen as desirable. Such methods have been proposed by Hogg[4] and Wills[13].

Hogg introduces the concept of *islands* which are groups of objects which are only accessible via a *bridge* object. Hence, objects in a particular island cannot be aliased with objects in other islands. Aliasing, however, can still occur within islands.

Wills presents an approach for reasoning about aliasing. He argues that, in practical systems, reasoning is complicated by the existence of paths of references to objects not significantly related to the object being reasoned about. A notion of object *demesnes* which group the objects involved in the representation of a given object is proposed.

Techniques such as these will need to be adopted for Object-Z if precise reasoning is to be performed in the presence of aliasing. This problem has been addressed in [3].

Liveness

The logic for Object-Z presented in [8] enables reasoning about the safety properties of specifications. These properties assert which states and state changes *may* occur. Properties which assert that a particular state or state change *must* occur are referred to as liveness properties. Examples of liveness properties include guaranteed occurrence of operations, fairness and termination.

The specification of liveness properties in Object-Z is enabled by temporal logic predicates on classes. These predicates are expressed in terms of the operators $\Box$ (always), $\Diamond$ (eventually) and $\bigcirc$ (next). To reason about such properties, the logic for Object-Z needs to be extended to include rules for these operators. This could be done as in the Appendix of Turski and Miabaum[12] which describes the extension of predicate logic to include the modal operators $\Box$ and $\Diamond$. $\Box$ is introduced by the distribution axiom and rule of necessitation

$$\frac{}{\vdash \Box(p \Rightarrow q) \Rightarrow (\Box p \Rightarrow \Box q)} \;(\ \mathit{Distribution}\ ) \qquad \frac{\vdash p}{\vdash \Box p} \;(\ \mathit{Necessitation}\ )$$

and $\Diamond$ is defined in terms of $\Box$ as follows.

$$\vdash \Box p \Leftrightarrow \neg \Diamond \neg\, p$$

The $\bigcirc$ operator is used to describe the next state of an object. It could be defined in terms of priming as follows.

$$\frac{\vdash \pi(\langle\!|\ x_1 \leadsto x_1', \ldots, x_n \leadsto x_n'\ |\!\rangle \odot p)}{\vdash \bigcirc p} \;(\ \bigcirc\ )\ [\ \{x_1, \ldots, x_n\} = \alpha_1(A) \cup \alpha_2(A)\ ]$$

Other rules for these operators could then be defined as appropriate.

6 Conclusion

This paper has presented the basis of a practical proof system for Object-Z. It extends the logic of Object-Z presented in [8] and [9] with rules for modular reasoning. The rules enable properties proved for a class in isolation to be used when proving properties of other classes which either inherit or instantiate the original class. Rules for proofs by structural induction and for capturing the notion of object integrity were presented. Future work needs to address the problems of aliasing between objects and reasoning about liveness properties.

Acknowledgements

The author would like to thank Roger Duke, Wendy Johnston and Gordon Rose for many helpful discussions. This work is supported by the Australian Research Council (ARC).

References

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

[2] R. Duke, G. Rose, and G. Smith. Object-Z: a specification language advocated for the description of standards. Technical Report 94-45, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1994.

[3] A. Griffiths. Object identity in Object-Z. Honours Report, Department of Computer Science, University of Queensland, 1993.

[4] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91)*, volume 26, pages 271–285. ACM SIGPLAN Notices, 1991.

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

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

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

[8] G. Smith. Extending $\mathcal{W}$ for Object-Z. In J. Bowen and M. Hinchey, editors, *9th International Conference of Z Users*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[9] G. Smith. A logic for Object-Z (additional rules). Technical Report 95-26, Software Verification Research Centre, Department of Computer Science, University of Queensland, Australia, 1995.

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

[11] I. Toyn and J. Hall. Proving conjectures using CADi$\mathbb{Z}$. Department of Computer Science, University of York, 1994.

[12] W.M. Turski and T.S.E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[13] A. Wills. *Formal Methods Applied to Object-Oriented Programming.* PhD thesis, University of Manchester, 1992.

[14] J.C.P. Woodcock and S.M. Brien. $\mathcal{W}$: A logic for Z. In J.E. Nicholls, editor, *Z User Workshop*, Workshops in Computing, pages 77–98. Springer-Verlag, 1992.

Appendix

The *Leibniz* rule enables substitution of an expression t for an expression s in a predicate p provided s and t are equal.

$$\frac{s = t, \langle\!|\ x \leadsto t\ |\!\rangle \odot p \vdash}{s = t, \langle\!|\ x \leadsto s\ |\!\rangle \odot p \vdash} \quad (\ \mathit{Leibniz}\ )$$

This is not always applicable so we have a derived rule *ApplyLeibniz* ($p(s)$ denotes a predicate expressed in terms of an expression s).

$$\frac{p(t) \vdash}{s = t, p(s) \vdash} \quad (\ \mathit{ApplyLeibniz}\ )$$

The following derivation of this rule is based on the approach in [11]. (1= defn of $\odot$, 2=*Leibniz*)

$$\dfrac{\dfrac{\dfrac{\dfrac{p(t) \vdash}{s = t, p(s), p(t) \vdash}\ (\ \mathit{thin}\ )}{s = t, p(s), \langle\!|\ x \leadsto t\ |\!\rangle \odot p(x) \vdash}\ (\ 1\ )}{s = t, p(s), \langle\!|\ x \leadsto s\ |\!\rangle \odot p(x) \vdash}\ (\ 2\ ) \qquad \dfrac{\dfrac{}{s = t, p(s) \vdash p(s)}\ (\ \mathit{assumption}\ )}{s = t, p(s) \vdash \langle\!|\ x \leadsto s\ |\!\rangle \odot p(x)}\ (\ \text{defn of } \odot\ )}{s = t, p(s) \vdash}\ (\ \mathit{cut}\ )$$