15

Type Checking Classes in Object-Z to Promote Quality of Specifications

J. Chen and B. Durnota

Department of Software Development, Monash University, PO Box 197,
Caulfield East 3145, Australia

Abstract

Formal specification of software requirements has been recognised as an essential ingredient to improve the quality of delivered code. When integrated with the structuring mechanisms of object-orientation, formal specifications can provide a powerful tool for the software developer. However, the formal specification of particularly large systems needs to be supported by tools which assist the specifier or reader of the specification either to understand and/or to reason about a specification. The simplest form of reasoning is that of type checking. This paper develops a simple but useful set of rules for type checking the object-oriented formal specification language Object-Z. Although type checkers exist for Z, at present none exists for Object-Z. The lack of a type checker for Object-Z is a hinderence to a wider and more consistent use of this specification language as an integral component in developing high-quality software products.

Keyword Codes: D.2.1; D.2.2; D.1.5
Keywords: Requirements/Specifications; Tools and Techniques; Object-oriented Programming

1. INTRODUCTION

The construction of software conforming to a client's requirements has been a major challenge of software engineering. Formal specification, as an important part of formal methods, has been an active area in recent years, firstly within the academic and research community and now attracting serious attention from industry. One of the most successful formal specification languages is Z, developed by the Programming Research Group at Oxford and is being used by many industry companies [1].

Object-Z is an object-oriented extension of Z developed in order to overcome some of the limitations of Z [2,5], especially with regards to the specification of large systems. Although Object-Z is largely based on Z, there are important differences such as the notion of *class*, *class type* and *class inheritance*.

However, the formal specification of particularly large systems needs to be supported by tools which assist the specifier or reader of the specification either to understand and/or to reason about a specification. Reasoning can range from the easier, more syntactic level of

M. Lee et al. (eds.), *Software Quality and Productivity*

type checking, through to a wider semantic analysis (such as consistency checking), and ultimately to theorem proving.

The work described in this paper concerns the type checking issue of Object-Z specifications. Although type checkers exist for Z, at present none exists for Object-Z. This has hindered a wider and more consistent use of Object-Z.

Type checking Z specifications has been previously investigated and several type checkers have been developed [6]. From several recent industry surveys and reports of using Z, type checkers are regarded as an important tool in the construction of formal specifications [4, 9]. Especially for specifications of large size or complex or involved team work, type checking is a simple and effective way to improve the quality of specifications.

In this paper, we focus on type checking the components of Object-Z related to defining or referring to class types. We will briefly discuss other aspects of type checking Object-Z, which can be based on the work done to Z. A detail report on type checking Object-Z is under preparation[3].

Formal specifications provide a means to precisely describe the requirements of software. Therefore using formal specification could help in achieving better quality of software. However such a benefit cannot be taken as granted without suitable tool support as various stages of specification construction. One of the important quality criteria is that the specification itself be consistent. We regard the consistency as having different degrees. These degrees of consistency range from type consistency, selected aspect consistency and total consistency. Realistically, most software developments would require the type and selected aspect consistency. In some cases, especially in safety-critical software development, the total consistency may be required. We believe our work described in this paper is an important step towards a tool set for supporting quality assurance of formal specification.

The rest of this paper is organised as follows. In Section 2 we give a brief description of class types in Object-Z and other background knowledge needed in this paper. Section 3 discusses typing rules for class type checking in Object-Z. Finally we discuss further work in Section 4.

2. CLASS TYPES IN OBJECT-Z

Z is a model-based specification language based on typed set theory and with structuring constructs called schemas [8]. Object-Z is an object-oriented extension to Z developed to allow the structuring of large specifications [2, 5, 7].

A Z specification consists of a number of *state schemas*, *initial state schemas* and *operation schemas*. The values of variables defined in a state schema define a state of the system we are modelling. An operation schema relates input to output variables, as well as changes to state variables before invocation of the operation schema to that after invocation. The *predicate* part of a schema imposes constraints amongst values of variables (*invariants, preconditions, postconditions*).

Object-Z bundles all the operations pertaining to a given state schema into a *class*. Classes are templates for *object* instances of that class. Classes may be defined via (potentially multiple) *inheritance*, and variables may reference objects -- this is called *instantiation*. A class definition parameterized with generic types has the following form:

ClassName [*generic parameters*]
visibility list
inherited classes
type definitions
constant definitions
state schema
initial state schema
operation schemas
history invariant

An example of an Object-Z class definition is the following one of a bounded stack of items of type *T*:

Stack[*T*]

$max : \mathbb{N}$

$max \leq 100$

$items : \text{seq } T$

$\#items \leq max$

INIT

$items = \langle\rangle$

Push

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

$\#items < max$
$items' = \langle item? \rangle \frown items$

Pop

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

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

The constant *max* is the bound on the stack. The state variable *items* models the stack as a sequence of items of type *T*. Initially, the stack has no items. The *Push* operation prepends an input *item?* to *items* -- only if there is room left in *items*. The Δ-list gives the state variables that are changed by the operation -- *items* in this case. The *Pop* operation outputs the head *item!* from *items*, and then equates the new value of *items* to the tail -- provided *items* is not empty.

2.1 Instantiation

Any object may have any other object as its constituent. This is done by a variable referencing either an individual object or an aggregate. Consider our generic stack in which the generic type *T* has been replaced by N and some of the variables are renamed, thus giving a stack of natural numbers:

NatStack == *Stack*[N][*nats/items, nat?/item?,nat!/item!*]

Class *StackPair* contains two individually named references s_1 and s_2 to stacks of natural numbers:

StackPair

$s_1, s_2 : NatStack$

$s_1 \neq s_2$

$\#s_1.nats \leq \#s_2.nats$

INIT

$s_1.INIT \wedge s_2.INIT$

$Push_1 \mathrel{\widehat=} s_1.Push$
$Push_2 \mathrel{\widehat=} s_2.Push$
[*other operations*]

$Push_1$

$s_1, s_2 : NatStack$
$s_1.max : \mathbb{N}$
$s_1.nats, s_1.nat' : \text{seq}\,\mathbb{N}$
$nat? : \mathbb{N}$

$s_1 \neq s_2$
$\#s_1.nats \leq s_2.nats$
$\#s_1.nats' \leq s_2.nats'$
$\#s_1.max \leq 100$
$\#s_1.nats \leq s_2.max$
$\#s_1.nats < s_2.max$
$s_1.nats' = \langle nat? \rangle \frown s_1.nats$

Since it is intended the references s_1 and s_2 do not change, they are declared as constants. We have also required that the size of the first stack does not exceed the size of the second stack. The dot notation '.' is used to reference features of an object, such as it's state variables. The expanded definition of operation $Push_1$ is given alongside.

One sometimes wants to reference an object of a particular class or any of its derivative subclasses. This is achieved using the $\downarrow$ prefix. Thus:

stack: $\downarrow$ *Stack*[N]

declares *stack* as a reference to a stack of natural numbers, or to any subclass of stack, where the actual natural number type N has been substituted for the generic type parameter.

IndexedStack[*T*]

Stack[*T*]

$index : \mathbb{N}_1$

$items \neq \langle\rangle \Rightarrow index \in \text{dom}\, items$

SetIndex

$\Delta(index)$
$n? : \mathbb{N}_1$

$n? \in \text{dom}\, items$
$index' = n?$

Push

$\Delta(index)$

$items \neq \langle\rangle \Rightarrow index' = index + 1$

Pop

$\Delta(index)$

$index = 1 \wedge items' \neq \langle\rangle \Rightarrow$
$index' = 1$
$index \neq 1 \Rightarrow index' = index - 1$

2.2. Inheritance

Inheritance allows classes to be incrementally specified by reusing previous class definitions. Type and constant definitions and schemas of the inherited classes and those declared explicitly in the derived classes are merged. Schemas with the same name are conjoined. renaming can be used to avoid unintended name clashes during merging in the inherited class. As an example, an indexed stack can be defined by inheriting from the generic stack.

3. TYPING RULES FOR CLASS TYPES AND OBJECT REFERENCES

In this section, we discuss class type checking issues with respect to each of the uses of class types in Object-Z as described above. We first consider typing rules for the signature parts of classes in the case of inheritance and instantiation respectively. Then we discuss typing rules to expressions of Object-Z with class or class feature references.

3.1. Typing Rules for Class Signatures

3.1.1. Inheritance

When we define a new class in Object-Z, the new class may inherit features from existing classes. Semantically, the features of the new class is the collection formed by merging the features of inherited classes and newly defined features of the current class.

Class type conflicts may occur from this merge. For example, if there is an operation *op* defined in one of the inherited classes and a new operation with the same name *op* is defined in the current class, this should be regarded as a conflict of calls (type) components. A typing rule corresponding to this is as follows:

Let *op* be an operation defined in an inherited class C_1 of a class C. We say that *op* in C is *type correct* if there is no other operation with the same name defined in C, except by explicit redefinition. (Redefinition must conform to the typing rule below.)

We may indeed intend to use the same name of operation but with a new definition in the current class. A redefinition facility can be used in such situations. From a typing point of view, the redefinition of the operation *op* should conform to the following rules:

Let *op* be an operation defined in a class C. Suppose that *op* is redefined in an inherited class C_1 of C. We say that the redefinition of *op* in C_1 is *type correct* if the signature of the new *op* in C_1 is the same as that of C.

In case an operation is explicitly removed in the definition of a new class, no specific typing rules are required with respect to inheritance of classes in defining a new class. However, there are rules on the use of such classes with respect to instantiation of classes.

Object-Z allows multiple inheritance in the definition of a new class. Semantically, the newly defined class is the collection formed by merging the inherited classes together with newly defined features of the current class. It is possible that the merge of more than one inherited class may cause type conflicts. The following typing rules are corresponding to such situations:

Let C_1 and C_2 are two inherited classes of class C. We say the merge of C_1 and C_2 is *type correct* if the following conditions are satisfied: 1. Any common attributes (constants and variables) of C_1 and C_2 should be of the same type. 2. Any common operation of C_1 and C_2 should have the same signatures.

typing rule.

3.1.2. Instantiation

A class is considered to be *type correct* if, at least, its declarations and operations are type correct. Then, the first typing rule is simple but important:

Let *r: ClassName* be a constant or variable declaration in a class, where *r* is a constant or variable name. We say that the declaration is *type correct* if:

1. Class *ClassName* is type correct.

Class *ClassName* is an existing class.

The reason for requiring this rule is simple. If *ClassName* is not defined yet, then all references of the features of the *ClassName* in the current class or any class inheriting the current class are not defined (often called *message not understood* in object-oriented programming languages).

The second typing rule concerns reference of inheritance hierarchies:

Let $r.\downarrow$ *ClassName* be a constant or variable declaration in a class *C*. We say that the constant or variable declaration is *type correct* if the following conditions are satisfied.

1. Class *ClassName* and every class inherited from *ClassName* are *type correct.*
2. There is no operation defined in *ClassName* which is removed from any of the classes inherited from *ClassName.*

The first condition is not difficult to understand as a feature reference to *ClassName* or any of its inherited classed could be undefined if they are not type correct.

Recall that the notation $\downarrow$ is introduced for reference to any class in a class inheritance hierarchy in Section 2. The intuition of the second condition is that, if any of the operation defined in the root class of a class inheritance hierarchy is removed from a class inherited from the root class, a feature reference to that operation in an object reference to that class may cause reference undefined. In order to avoid such situations, the second condition must be imposed.

3.2. Typing rules for expressions with class references

There are two main types of expressions in Object-Z as in classical first-order logic: the terms and the predicates. *Terms* correspond to function applications, and *predicates* are predicates associated with state schemas, initialisations, operations schemas and definitions in a class. Two tasks are involved in type checking expressions, and they involve the notion of well-typedness:

A term is *well-typed* if:

1. It is a constant or variable of a declared type (this declared type is called the *type* of the term), or
2. It is of the form $f(t_1,...,t_n)$, where all the t_j are well-typed, and of type T_j and the declared type of f is $T_1 \times ... \times T_n \rightarrow T$. Then T is called the *type* of the term.

A predicate is *well-typed* if:

1. It is of the form $R(t_1,...,t_n)$, where all the terms t_j are well-typed, and of type T_j and the declared type of the relation symbol R is $T_1 \times ... \times T_n \rightarrow T$, or

2. It is of the form $\neg P$ for a well-typed predicate P, or
3. It is of the form $P \wedge Q$ for well-typed predicates P and Q, or

Similarly for other logical connectives and quantifiers.

Then, an expression is *type-correct* if it is well-typed.

For example, suppose that $+$ is a two-place operation symbol requiring both arguments to be of type *integer*. Then $x + y$ is well-typed if and only if both x and y are of type *integer*, and the type of the expression is *integer*. Suppose that $=$ is a two-place predicate symbol for equality. The predicate $E_1 = E_2$ is well-typed if and only if E_1 and E_2 are two well-typed expressions and are of the same type.

Type checking expressions and predicates of Object-Z can be largely based on the type checking method used for Z expressions and predicates, except the well-typedness of expressions with class feature references.

Let us now consider the typing rules for expressions of Object-Z with class feature references. Other aspects of type checking of Object-Z expressions and predicates are the same to that of Z expressions and predicates[6]. There are two typing rules for expressions with class feature references:

Let *r.op* be an expression occurring in a class definition, where *r* is a constant or variable of the class, and *op* is an operation. We say that *r.op* is *well-typed* if the following conditions are satisfied:

1. *r* is declared as an object instance of class *C* or of any of its subclasses -- as *r*: *C* or $r: \downarrow C$,
2. *r*: *C* or r: $\downarrow C$ is a *type correct* declaration, and
3. *op* is an operation defined in *C*.

Let *r.att* be an expression occurring in a class definition, where *r* is a constant or variable of the class, and *att* is an attribute (a constant or state variable). We say that *r.att* is *well-typed* and of type *T* if the following conditions are satisfied:

1. *r* is declared as an object instance of class *C* or of any of its subclasses -- as *r*: *C* or $r: \downarrow C$,
2. *C* or $r: \downarrow C$ is a *type correct* declaration, and
3. *att* is an attribute of *C* and of declared type *T*.

4. FUTURE WORK

We have described type checking classes and object references of Object-Z specifications in this paper. We regard type consistency is an important aspect in improving the quality of specifications. This work is part of our effect towards quality formal specifications of software in general, and Object-Z specification in particular.

Future work based on this paper includes: (1) combining Z type checking with class type checking reported in this paper to achieve a unified type checking system for Object-Z, and (2) integrating type checking with other tools for analysing and reasoning Object-Z specification to provide a CASE tool for quality specification construction.

References

1. Barden, R., Stepney, S. and Cooper, D., "The Use of Z", in *Proc. of 6th Z User Workshop*, Nicholls, J.E. (ed.), Springer-Verlag, 1991, pages 99-124.
2. Carrington, D., Duke, D., Duke, R., King, P., Rose, G. and Smith, G., "Object-Z: An Object-oriented Extension to Z", in *Formal Description Techniques, II (FORTE'89)*, Vuong, S. (ed.), North-Holland, 1990, pages 281-296.
3. Chen, J. and Durnota, B., "Type Checking Object-Z Specifications", in preparation.
4. Craigen, D., Gerhart, S. and Ralston, T., *An International Survey of Industrial Applications of Formal Methods*, two volumes, U.S. Department of Commerce, MD 20899, 1993.
5. Duke, R., King, P., Rose, G. and Smith, G., "The Object-Z Specification Language - Version 1", Technical Report No. 91-1, Software Verification Research Centre, The University of Queensland, May 1991.
6. Reed, J.N. and Sinclair, J.E., "An Algorithm for Type-Checking Z", Technical Monograph RG-81, Oxford University, 1990.
7. Rose, G., "Object-Z", in *Object Orientation in Z*, Stepney, S., Barden, R. and Cooper, D. (eds.), Springer-Verlag, Chapter 6, 1992.
8. Spivey, J.M., *The Z Notation: A Reference Manual*, International Series in Computer Science, Prentice-Hall, 1989.
9. Weber-Wulff, D., "Selling Formal Methods to Industry", in *Proc. of FME'93: Industrial-Strength Formal Methods*, Woodcock, J.C.P. and Larsen, P.G. (eds.), LNCS 670, Springer-Verlag, 1993, pages 671-678.