Design of an object-oriented formal specification language

Márcio Cornélio and Paulo Borba

Departamento de Informática
Universidade Federal de Pernambuco
E-mail: {malc, phmb}@di.ufpe.br

Abstract

In this work we propose an extension of the MooZ specification language with support for parameterized packages. This enhances MooZ's capabilities for reuse and maintenance in the large. We discuss other aspects considered during the design of this extension: the distinction between inheritance and subtyping, values and objects, vertical and horizontal composition, packages and classes.

Key Words: Formal Methods, Z, Object-oriented specification, Parameterized Programming, Language design

1 Introduction

The integration of formal methods with object-oriented concepts can be useful for the development of high quality software. Formal methods are important when reliability is essential. On the other hand, object-oriented concepts such as objects and classes, subtype polymorphism, inheritance, and dynamic binding [1] favor software reuse and maintenance.

Packages can enhance the support for reuse and maintenance offered by object-oriented languages because they can group several (related) classes and data types, and so play an important role in structuring complex software. Parametric polymorphism [8] is another concept for enhancing reuse and maintenance. It can be implemented by parameterization mechanisms at the level of packages, so that packages can be parameterized by packages.

Here we discuss the main aspects considered during the design of an extension of MooZ [11, 10], a formal specification language whose name stands for *Modular object-oriented Z*. MooZ is a conservative object-oriented extension of the Z [7, 6] specification language. We extended MooZ for supporting parameterized packages. This enhances MooZ's capabilities for reuse and maintenance *in the large*. In addition, we distinguish inheritance from subtyping, values from objects, vertical composition from horizontal one, and packages from classes.

This extended abstract is organized as follows. First we discuss the concepts of inheritance and subtyping, then we present the concept of data types. We discuss packages, vertical and horizontal composition, and parameterized packages in the subsequent sections.

2 Inheritance and subtyping

A key decision for the extended version of MooZ was to provide different mechanisms for inheritance and subtyping. Inheritance[1] is a key reusability technique [1]. It avoids solving a problem from scratch and redoing what has already been done. It is a way of defining new classes based on already defined ones. By using **inherits** B in a class A, we indicate that class A is a heir of class B. MooZ allows single and multiple inheritance and does not require the semantics of an ancestor class to be preserved by the heir class. So, a heir class does not have to refine the ancestor. Furthermore, an object of a heir class cannot be used where an object of an ancestor class is expected because inheritance does not establish a type relationship between heir and ancestor classes. As an example of inheritance, let us specify the class of bank accounts for which every withdrawal is taxed behaving differently from the class of current accounts for which withdrawals are not taxed. Thus, *TaxedAccount* can be specified as a heir of class *CurrentAccount*:

Class $CurrentAccount$

protected $number, balance$

state

$$\begin{array}{|l} number : \mathbb{N} \\ balance : \Re \\ \hline \end{array}$$

operations

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Withdrawal}} \\ \hline \Delta(balance) \\ m? : \Re \\ \hline balance - m? \geq 0 \\ balance' = balance - m? \\ \hline \end{array}$$

...

EndClass $CurrentAccount$.

Class $TaxedAccount$

inherits $CurrentAccount$

operations

$$\begin{array}{|l} \multicolumn{1}{l}{\textit{Withdrawal}} \\ \hline \Delta(balance) \\ m? : \Re \\ \hline balance - (m? \times 1.01) \geq 0 \\ balance' = balance - (m? \times 1.01) \\ \hline \end{array}$$

EndClass $TaxedAccount$.

So code is reused, but objects of *TaxedAccount* cannot be used where objects of class *CurrentAccount* are expected.

Subclassing is a classification technique, having a significant impact on extendibility. In MooZ, subclassing really means subtyping tied to inheritance. Classes related by subclassing indicates that objects of a subclass are also objects of a superclass because they establish a subtype relationship. Moreover, objects of a subclass can be used in contexts where objects of a superclass are expected, and objects of the subclass must have at least the same public attributes and methods of the superclass, preserving their behavior. By using **superclasses** B in a class A, we indicate that class A is a subclass of class B. MooZ allows a class to have multiple superclasses.

For instance, the class of saving accounts is very similar to the class of current accounts, but the former has an operation that applies an interest to the account's balance.

---

[1] There is usually a confusion between the inheritance and the notion of subtyping tied to inheritance.

Class *SavingAccount*

superclasses *CurrentAccount*

operations

$$
\begin{array}{|l}
\multicolumn{1}{l}{\quad ApplyInterest} \\
\hline
\Delta(balance) \\
interest? : \Re \\
\hline
balance' = balance + (balance \times interest?) \\
\hline
\end{array}
$$

EndClass *SavingAccount*.

The mechanisms for inheritance and subclassing provide the separation between the inheritance hierarchy and the type hierarchy. Therefore, it is possible to reuse code of a class to define another, even when they cannot establish a type relationship.

3 Data types

Besides classes, MooZ supports another mechanism for defining types. Abstract data types, which are sets of values and operations over them, can also be defined. The main difference between classes and data types is that the first defines objects, whereas the second defines values. Values are immutable, whereas objects can have their state changed by method execution. Basically, data types are related to the functional aspects of MooZ directly inherited from Z. In fact, state independent definition like free types, schema types, abbreviations, constants (introduced by axiomatic descriptions [7]) are used to define data types. For instance, the following data type *CmpValue* specifies the concept of a comparable value.

Datatype *CmpValue*

$$| \_ = \_ : \quad CmpValue \times CmpValue \rightarrow \mathbf{B}$$

$$| \_ < \_ : \quad CmpValue \times CmpValue \rightarrow \mathbf{B}$$

EndDatatype *CmpValue*.

The first operation allows comparing whether two values are equivalent, and the second one allows comparing values with respect to an ordering relation like greater than or less than.

The distinction between classes and data types avoids declaring stateless definitions such as those mentioned above in a class context as it would be artificial and hinder reuse of that definition.

4 Packages

Packages are encapsulation units [2]. Each package in a MooZ specification can group the declaration of several related classes and data types. Thus packages are essential for structuring large object-oriented specifications consisting of many classes and data types.

Package *NONTAXEDACCOUNTS*

Here should appear the specification of classes CurrentAccount and SavingAccount.

EndPackage *NONTAXEDACCOUNTS*.

introduces the package *NONTAXEDACCOUNTS* that contains the classes that describe non-taxed accounts (*CurrentAccount* and *SavingAccount*) declared so far.

In order to adapt packages to new contexts, MooZ allows declarations (classes, data types and operations) contained in a package to be given new names, thus favoring reuse. It avoids duplicating specification text to obtain declarations with the desired names. For instance, the expression

$$NONTAXEDACCOUTS\#(CurrentAccount \backslash RegularAccount)$$

gives the name *RegularAccount* to the class *CurrentAccount* (contained in the package *NONTAXEDACCOUTS*).

Renaming is also useful to solve name clashes between classes, data types and operations. MooZ supports another operation on packages that slices the functionality provided by a package, but it is discussed elsewhere [9].

5 Vertical and horizontal composition

Public declarations present in a package are available inside this package and can only be accessed by other packages by means of composition features. Package composition concerns the combination of packages in order to form new packages. Package importation is a composition feature. In MooZ, packages can be horizontally and vertically imported. The horizontal mode of importation is used when the concepts of a package inherently depend on the concepts of an imported package. They are said to be in the same level of abstraction. A package that vertically imports another package simply uses the auxiliary functionality (services) provided by the imported package. In other words, the imported package is used as an ancillary one. The vertical mode of importation involves different levels of abstraction. Packages in a lower level of abstraction support those in an upper level of abstraction [5]. Another difference between the horizontal and vertical modes of composition is that the first is transitive (cumulative) whereas the second is not transitive. In MooZ, the horizontal mode of importation is indicated by the clause includes whereas the vertical mode is indicated by the clause uses. For instance,

Package *TAXEDACCOUNTS*

includes *NONTAXEDACCOUNTS* ...

EndPackage *TAXEDACCOUNTS*.

introduces the package *TAXEDACCOUNTS* that horizontally imports the package *NONTAXEDACCOUNTS* since the former inherently depends on the concepts described by the second. Consequently, both packages are at the same level of abstraction.

6 Parameterized packages

Packages can also be parameterized, then called generic packages. A generic package is very similar to a non-parameterized package, except that it is parameterized by other packages, and so can refer to those parameters in its body. In order to be used, parameterized packages must be instantiated with arguments (packages). Each instantiation yields a new package. So, a parameterized package defines a set of related packages, one for each possible instantiation. Packages of such set offer similar behavior, but differ on the type of information they manipulate [3].

In MooZ, package parameters have associated *package types* for specifying which arguments are valid. They impose restrictions on its parameters, stating exactly what constraints valid arguments have to satisfy. This satisfaction corresponds to package refinement: an argument package must provide types (classes and data types) that refine those of the package type. An argument has to offer at least the same functionality stated by the corresponding package type. Packages are used as package types. Parameterization allows us to obtain different packages by simply varying arguments. Similarly to importation, MooZ supports the horizontal and vertical modes of parameterization.

As an example of parameterization, let us specify a package that defines a class of banks. This package is horizontally parameterized by the package that defines the account to be stored in the bank. Arguments to this package must offer at least a class with the same functionality of those in *NONTAXEDACCOUNTS*.

Package $BANK[ACCT :: NONTAXEDACCOUNTS]$

We model a bank as a set of regular accounts. The class that describes regular accounts was specified in the package *NONTAXEDACCOUNTS* which is the type of parameter *ACCT*. So we can refer to types, which were declared in the package type, in the body of the parameterized package. We could also refer to the parameter *ACCT* in the body of *BANK*, for example, if we instantiate and import a package.

Class *Bank*

state

$accounts : \mathbb{P}\, RegularAccount$

...

EndClass *Bank*.

EndPackage *BANK*.

An argument package may satisfy a package type in more than one way. MooZ provides *views* [4] for indicating how a package should satisfy a package type in a particular instantiation. Basically, a view is a mapping from type names and operations of a package type to types names and operations of an argument package. The use of *views* allows software components to be viewed differently in different contexts. It favors software reuse because the contexts a component might be reused are usually different. So, a view allows adapting components without having to modify them. *Views* can express relationships between arguments and package types that have not already been expressed [5, 3].

7 Conclusion

In this extended abstract we discussed some aspects we pondered on during the design of an extension of MooZ. One difficulty we faced is related to the textual aspect of MooZ. For instance, it has a great impact on data type refinement, requiring *name equivalence* (a data type that introduces schemas for specifying messages, for example, is refined by another data type only if the latter specifies equivalent messages).

Some specifications using the new features of MooZ have been developed, and the first results show that these features have a satisfactory behavior, improving MooZ capabilities and expressiveness. We propose, as a future work, the formal description of the module system presented here.

References

[1] B. Meyer. *Object-Oriented Software Construction.* Prentide-Hall, second edition, 1997.

[2] D. A. Watt. *Programming Languages Concepts and Paradigms.* C. A. R. Hoare Series Editor. Prentice Hall International, 1989.

[3] D. Aranha and P. Borba. Parameterized packages and Java. *II Brazilian Symposium on Programming Languages*, pages 208–218, September 1997. Campinas, Brazil.

[4] J. Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability*, volume I: Concepts and Models, chapter 7, pages 159–225. ACM Press, 1989.

[5] J. Goguen and A. Socorro. Module composition and system design for the object paradigm. *Journal of Object-Oriented Programming*, 7(9), 1995.

[6] J. M. Spivey. *Understanding Z: A specification language and its formal semantics.* Cambridge University Press, 1988.

[7] J. M. Spivey. *The Z Notation: a reference manual.* C. A. Hoare Series Editor. Prentice Hall International, 2nd edition, 1992.

[8] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.

[9] M. L. Cornélio. Design and evaluation of an object-oriented formal specification language. Master's thesis, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1998.

[10] S. L. Meira and A. L. C. Cavalcanti. MooZ Case Studies. In S. Stepney, R. Barden and D. Cooper, editor, *Object Orientation in Z*, Workshops in Computing, chapter 5, pages 37–58. Springer-Verlag, 1992.

[11] S. R. L. Meira and A. L. C. Cavalcanti. The MooZ Specification Language. Technical report, Universidade Federal de Pernambuco, Departamento de Informática, Recife - PE, 1992. Also available on the Internet at http://www.di.ufpe.br/~mooz.