Structuring Mechanisms for an Object-Oriented Formal Specification Language

Márcio Cornélio and Paulo Borba

Centro de Informática
Universidade Federal de Pernambuco
CP 7851 CEP 50732-970 Recife PE Brasil
{mlc2, phmb}@di.ufpe.br

Abstract. In this work we propose an extension of the MooZ formal specification language with support for parameterized packages. This enhances MooZ's capabilities for software reuse and maintenance *in the large*. We discuss several design issues for MooZ's structuring mechanisms: the distinction between inheritance and subtyping, values and objects, vertical and horizontal composition, packages and classes. We also analyse the impact of our design decisions on software reuse.

Keywords : Formal Methods, Z, Object-Oriented Specification, Parameterized Programming, Language design

1 Introduction

The integration of formal methods with object-oriented concepts can be quite 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 [2] favor software reuse and maintenance.

Packages can enhance the support for reuse and maintenance offered by object-oriented concepts because they can group several (related) classes and data types, and so play an important role in structuring complex software. Parametric polymorphism [18] is another powerful concept for enhancing reuse and maintenance. It can be implemented by parameterization mechanisms at the level of packages, so that packages are parameterized by packages.

Here we discuss several design issues of structuring mechanisms for MooZ [24, 23], a formal specification language whose name stands for *Modular object-oriented Z*. MooZ is a conservative object-oriented extension of the Z [14, 13] specification language. Since the beginning of its development MooZ has been widely used to formalize computational models in several different application areas such as Artificial Neural Networks [20] and Medical Systems [8].

We extended MooZ to support 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. The extended version of MooZ is called MooZPP, an acronym for MooZ with Parameterized Packages.

This paper is organized as follows. In Sect. 2, we present the features introduced in MooZPP, an extension of MooZ. In Sect. 3 and 4, respectively, we discuss the impact that these features have on reuse *in the small* and on reuse *in the large*. In Sect. 5, we compare MooZPP with other works. Finally, in Sect. 6 we summarise our results.

In the following sections we assume basic knowledge of object-oriented concepts, Z or of a similar state-based formal specification language.

2 MooZPP

MooZPP [19] is an extension of MooZ that supports parameterized packages. A MooZPP specification consists of a set of packages. Each package can declare several (mutually) related classes and data types. In the following sections we discuss the main aspects considered during the design of MooZPP.

2.1 Inheritance and Subclassing

A key decision for MooZPP was to provide different mechanisms for inheritance and subtyping. Inheritance[1] is a key reusability technique [2]. 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 the definition of a class A, we indicate that A is an heir of B. MooZPP allows multiple inheritance and does not require the semantics of an ancestor class to be preserved by the heir class. So, an heir class does not have to refine the ancestor. Furthermore, an object of an heir 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, assuming we already have a class of current accounts for which withdrawals are not taxed. Thus, *TaxedAccount* can be specified as an heir of *CurrentAccount*:

Class *CurrentAccount*

protected *number*, *balance*

state

$number : \mathbb{N}$ $balance : \Re$
$balance \geq 0$

[1] There is usually a confusion between the notions of inheritance and "subtyping with inheritance".

operations

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Credit \qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta(balance) \\
m? : \Re \\
\hline
m? \geq 0 \\
balance' = balance + m? \\
\hline
\end{array}$$

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

...

EndClass $CurrentAccount$.

The protected clause is present only in MooZPP and indicates that attributes *number* and *balance* are visible to heir classes and subclasses of *CurrentAccount*. The private clause of MooZPP indicates that attributes are not visible to heir classes and subclasses. Both clauses can change the visibility of operations too. The predicate of the anonymous schema of *CurrentAccount* establishes that the balance of a current account will never be less than zero.

Class $TaxedAccount$

inherits $CurrentAccount$

operations

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

...

EndClass $TaxedAccount$.

So (specification) code is reused, but objects of *TaxedAccount* cannot be used where objects of class *CurrentAccount* are expected. In fact, they should not because they have incompatible behaviour (the operation *Withdrawal* of *TaxedAccount* behaves differently from *Withdrawal* of *CurrentAccount*). Therefore, it is possible to reuse code of a class to define another, even when they cannot establish a type relationship.

Subclassing is a classification technique, having a significant impact on extendibility. In MooZPP, subclassing really means "subtyping with 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 behaviour. A method of a superclass can be overriden by a method of a subclass, since they establish a refinement relation. By using superclasses B in a class A, we indicate that A is a subclass of B. MooZPP 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 rate to the account's balance.

Class *SavingAccount*

superclasses *CurrentAccount*

operations

ApplyInterest

$\Delta(balance)$
$interest? : \Re$

$balance' = balance + (balance \times interest?)$

...

EndClass *SavingAccount*.

The attributes and operations of class *CurrentAccount* were all inherited by *SavingAccount*, preserving the behaviour of its superclass. So, objects of *SavingAccount* can be used where objects of *CurrentAccount* are expected.

2.2 Data Types

Besides classes, MooZPP supports another mechanism for defining types. Abstract data types, which are sets of values together with 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 application. Basically, data types are related to the functional aspects of MooZPP directly inherited from Z. In fact, state independent definitions like free types, schema types, abbreviations, constants (introduced by axiomatic descriptions [14]) are used to define data types. For example, a trivial data type is defined by the following specification.

Datatype *DTriv*

EndDatatype *DTriv*.

which introduces the data type *DTriv*. As there are no specified constraints upon *DTriv*, nothing can be said about the values or operations of this type, achieving the same effect of the specification of a given set in Z.

The following data type *CmpValue* specifies the concept of a comparable value.

Datatype $\mathit{CmpValue}$

$$\_ = \_ : \; \mathit{CmpValue} \times \mathit{CmpValue} \rightarrow \mathbf{B}$$

$$\_ < \_ : \; \mathit{CmpValue} \times \mathit{CmpValue} \rightarrow \mathbf{B}$$

EndDatatype $\mathit{CmpValue}$.

The first operation allows comparing whether two values are the same, 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 (free types, schema types, abbreviations, and constants introduced by axiomatic definitions) in a class context, as it would be artificial and hinder reuse of that definition.

2.3 Packages

Packages are encapsulation units [5]. Each package in a MooZPP specification can group the declaration of several (mutually) related classes and data types. The relationship between concepts described by classes and data types is a fact that must be taken in account when deciding what type definitions should be grouped in a specific package. For example, a class that specifies the edges of a graph is closely related to a class that specifies the vertices. In fact, these classes are mutually dependent: the definition of edges depend on the definition of vertices, and vice versa. This is a strong indication that these classes must be declared in the same package.

For example, consider the package $\mathit{NONTAXEDACCOUNTS}$. This package contains the classes $\mathit{CurrentAccount}$ and $\mathit{SavingAccount}$ declared so far. These classes are dependent since the concepts of accounts they describe are very close.

This is the reason for these classes to be specified in the same package.

Package $\mathit{NONTAXEDACCOUNTS}$

Here should appear the specifications of CurrentAccount and SavingAccount.

EndPackage $\mathit{NONTAXEDACCOUNTS}$.

Package declarations are only available inside the package or by means of composition features. Package composition concerns the ways packages are combined to form new packages. In this way, a package can access declarations of other packages. Package import is a composition feature.

Import. Import allows package declarations to be available to other packages. So, if package B imports package A, all visible declarations contained in A will be available to B.

Packages can be horizontally and vertically imported [9]. Basically what differentiates horizontal and vertical import is the level of abstraction involved in the import and its accumulative aspect, as will be explained below.

The horizontal mode of import 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. Through horizontal import, all visible declarations of an imported package are available to a package X and to all packages that import X. This process is transitive [11], so that if the package C includes the package B and B, in turn, includes the package A, then all visible declarations contained in A are also available to C.

Horizontal import is indicated by the includes clause. For example, let us specify a package that contains a class that specify taxed accounts. The package *ALLACCOUNTS* contains such class and those declared in the horizontally imported package.

Package *ALLACCOUNTS*

includes *NONTAXEDACCOUNTS*
Here should appear the specification of TaxedAccount
EndPackage *ALLACCOUNTS*.

Vertical import has to do with different levels of abstraction [11, 9]. A package that vertically imports another package uses the functionality provided by the imported package so as to realize a given functionality.

Through vertical import all visible declarations of an imported package are available to a package X that imports it, however these declarations are not available to packages that import package X. Vertical import is not a transitive process. The kind of dependence among packages that vertically import other packages is just circumstantial, not conceptual. The uses clause corresponds to the vertical mode of import. An example of vertical import is presented in Sect. 2.4.

Renaming. In order to adapt packages to new contexts, MooZPP 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. Package renaming in MooZPP is performed by the operator '#' followed by a renaming list. For example,

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

results in a new package with the same functionality as *NONTAXEDACCOUTS*, but with class *CurrentAccount* renamed to *RegularAccount*.

The renaming operator actually yields a new package, with the same functionality as its target, but with different declaration names. In the above example, the package *NONTAXEDACCOUNTS* remains the same after the application of the renaming operation. Renaming is also useful to solve name clashes between classes, data types and operations.

Slicing. MooZPP also supports an operation for slicing the functionality provided by a package. Slicing allows eliminating unwanted functionality from a package. If the functionality provided by a package exceeds what is needed to define another one, the former can be sliced to eliminate what is not necessary.

Slicing is accomplished by the operator '\' followed by a list of declaration names. For example,

$$NONTAXEDACCOUNTS \backslash (Withdrawal.SavingAccount)$$

denotes the package that is obtained by discarding the operation *Withdrawal* from class *SavingAccount* contained in the package *NONTAXEDACCOUNTS*.

The slicing operator actually yields a new package with a different functionality from the original one. The package used by the slicing operation remains the same after the application of the operation. In the above example, the package *NONTAXEDACCOUNTS* remains the same after the application of the slicing operation.

2.4 Parameterized Packages

MooZPP supports the concept of parametric polymorphism [18] through parameterized packages. These packages generalize a particular behaviour across a family of related packages. A parameterized package, also called a generic package, is very similar to a non-parameterized package, except that it is parameterized by other packages, and so can refer to these parameters in its body. In order to be used, a parameterized package must be instantiated with arguments (packages). Each instantiation yields a new package. Thus, a generic package defines a set of related packages, one for each possible instantiation. Packages of such a set differ on the type of information they manipulate and present a common behaviour pattern [6].

A generic package usually must impose restrictions on its parameters, in the same way a typed method imposes type constraints on its parameters.

Package Types. Package parameters have associated package types that state exactly what constraints valid arguments have to satisfy. This satisfaction corresponds to package refinement, as discussed in Sect. 2.4, which states that classes and data types present in the argument must refine those in the package type. We refine a formal specification by adding more information [15]. For example, we may be more precise about how data is to be stored, or about how certain calculations are to be carried out. A package type is similar to an ordinary package. For example, parameters having the following package type

Package $DTRIV$

Datatype $DTriv$

EndDatatype $DTriv$.

EndPackage $DTRIV$.

impose minimal restrictions on arguments because the package type contains a data type that does not declare any associated operation. Clearly stated, it specifies that any package containing any data type can be used to instantiate a parameterized package whose parameter has $DTRIV$ as package type.

Like ordinary packages, package types can contain several mutual dependent classes and data types. This indicates that the package type imposes constraints to a group of mutually dependent concepts. For example, the package *NONTAXEDACCOUNTS*, when used as a package type, requires argument packages to contain at least two classes offering the functionality offered by those classes in *NONTAXEDACCOUNTS*.

Parameterization. Parameterization is a key technique for obtaining abstraction. There are two kinds of parameterization in MooZPP: horizontal and vertical. The difference between them is the kind of dependence a parameterized package establishes with arguments used for instantiation, as will be exemplified below.

Horizontal. Horizontal parameterization involves parameters in the same level of abstraction of the parameterized package. It states that there is a conceptual dependence between a parameterized package and arguments used for instantiation.

For example, the package *BANK*, containing a class that defines a bank, is horizontally parameterized by a package that is supposed to define the classes that specify the accounts to be stored in the bank. The reason for horizontal parameterization is that the concept of accounts is inherent to the concept of bank. In other words, we cannot describe a bank without the concept of account. The symbol '::' binds the formal parameter $NTXACCT$ to its associated package type *NONTAXEDACCOUNTS*.

We import package *SET*, which contains the definition of a class of sets, such package is instantiated with the same argument used for instantiating *BANK* (we will discuss instantiation later).

Package $BANK[NTXACCT :: NONTAXEDACCOUNTS]$

uses $SET[NTXACCT]$

Class *Bank*

...

EndClass *Bank*.

EndPackage *BANK*.

BANK denotes a set of related packages that can be differentiated by the kind of accounts that are kept by banks. We obtain a different package for each different argument given to *BANK*, which is a parameterized package. In contrast to the specification of *BANK* given here, suppose this package were not parameterized, but horizontally imported package *NONTAXEDACCOUNTS*. In this case, we fix the kind of accounts a bank can keep, differently from the case of a parameterized package in which the kind of information to be manipulated depends on the argument given to the package.

Vertical. Vertical parameterization is another abstraction feature which involves packages in different levels of abstraction. For example, a vertically parameterized package depends on the representation of storage provided by an argument given to it, as will be discussed below. This kind of parameterization establishes a circumstantial dependence between a parameterized package and arguments used for instantiation.

Here we can point out the difference between horizontal and vertical parameterization. Horizontal parameterization states that a package inherently depends on the concepts defined by its parameters. For example, a bank inherently depends on the concepts of accounts. On the other hand, vertical parameterization states that the dependence on the concepts defined by package parameters is not inherent. For example, we could specify banks using sets. In this case, the package *BANK* would be vertically parameterized by the package *SET*. Here we parameterize *BANK* by a package that contains the descriptions of sequences in order to show that the concept of sets and sequences are not inherent for banks, which justifies the vertical mode of parameterization. The horizontal parameter has to do with the accounts to be stored in the bank, while the vertical parameter is the mathematical model of storage of a bank.

Package *BANKSEQ*
$[NTXACCT :: NONTAXEDACCOUNTS]$
$\{NTXACCTSEQ :: SEQUENCES[NTXACCT]\}$

Class *Bank*

...

EndClass *Bank*.

EndPackage *BANKSEQ*.

The package *BANKSEQ* has one horizontal parameter and one vertical parameter. The horizontal parameter is *NTXACCT* that can only be bound to argument packages that satisfy the properties specified by the package *NONTAXEDACCOUNTS*. The vertical parameter is *NTXACCTSEQ* whose package type is denoted by the package expression *SEQUENCES*[*NTXACCT*]. The parameter *NTXACCTSEQ* can be bound to argument packages that satisfy the requirements of the package type denoted by that expression. Notice that *SEQUENCES*[2] is instantiated with the horizontal parameter *NTXACCT* (we will discuss instantiation later).

In order to be used, this package must be instantiated with two argument packages. The first provides the definition of accounts that will be stored in the bank whose mathematical model is described by the second argument.

Views. An argument package may satisfy a package type in more than one way. So, it is necessary to describe the particular ways that argument packages satisfy package types. For example, the following package contains a data type that is an abbreviation for the natural numbers.

Package *NATURAL*

Datatype *Natural*

$Natural == \mathbb{N}$

EndDatatype *Natural*.

EndPackage *NATURAL*.

This package can satisfy the package type *CMPVALUE*, which contains only the data type *CmpValue* (see Sect. 2.2), which requires a data type with an operation for equality and another one for ordering. In fact, the required order could be instantiated either with $\leq$ or with $\geq$, among other options.

A view [10] is a mapping from the types (classes and data types) and operations of a package type to types and operations of an argument package, indicating how an argument package should satisfy a package type. In fact, views are used in order to express how an argument package satisfies a package type in a particular instantiation.

The semantic aspect of views is determined by the relation between declarations of the argument package and declarations of the package type. The argument package must provide declarations that satisfy (refine) declarations of the package type under the interpretation given by a view. In fact, a view is valid only if related types and operations hold the refinement notion.

More precisely, let D be a set of type declarations contained in a package type PT and D' a set of type declarations contained in an argument package AP. A

[2] This package contains the specification of different kinds of sequences, such as *FiniteSequence* and *NonEmptySequence*.

view $v : PT \rightarrow AP$ maps each element d in D to an element d' in D', and each operation op present in d is mapped into an operation op' in d'. Furthermore, the types of op and op' should match.

A *default view* is provided when there is an obvious view to use and it is not necessary to write out the view in full detail. For example, a view that consists of mapping classes and data types with the same names, since they present the same number of operations with the same name and type, is obvious. Unless the *default view* is not the desired one to map declarations from a package type to declarations of an argument package, it is necessary to explicitly declare the desired view.

Instantiation. Parameterized packages must be instantiated with actual arguments (packages) to be used. Instantiating a package requires views for its parameters. Any package that satisfies the requirements of a package type can be used to instantiate the corresponding parameterized package yielding a new package that is obtained from the parameterized package body by replacing references to the parameter package by references to the argument package, using the views to bind types and operations of the package type to types and operations of the argument package. In this way, references to types and operations of a package parameter become references to types and operations of an argument package. For example, the package *BANK* should be instantiated in order to obtain a specific class of banks:

Package *USUALBANK*

includes *BANK*[view to *NONTAXEDACCOUNTS*
    (*CurrentAccount* to *CurrentAccount*;
    `Withdrawal` to `Withdrawal`...)]

EndPackage *USUALBANK*.

This is an ordinary package obtained from the instantiation of *BANK* using *NONTAXEDACCOUNTS* as argument. As we are using the default view, we could specify the package *USUALBANK* in the following way:

Package *USUALBANK*

includes *BANK*[*NONTAXEDACCOUNTS*]

EndPackage *USUALBANK*.

Instantiations of a parameterized package yield different packages unless equivalent views are used. Two views are equivalent if, independent of type and operation names and order of appearance, they contain the same binding components [6]. In fact, considering that an argument package may satisfy the requirements of a package type in different ways, there might be different mappings from types and operations of a package type into types and operations of an argument package that are described by views, and consequently different bindings between parameter and arguments. So, different packages are yielded for each different view.

Compatibility Rules. In order to obtain valid instantiations, argument packages must offer at least the same functionality as their associated package types. In order to guarantee that an instantiation is valid, some compatibility rules must be satisfied by the views used for instantiation. These rules attest that types and operations of the package type are mapped to "compatible" types and operations of the argument package. They are further discussed elsewhere [19].

Argument packages must satisfy the constraints imposed on package parameters by their associated package types. This notion of satisfaction corresponds to refinement in the sense that an argument A satisfies a formal parameter P if $P \sqsubseteq A$, where $\sqsubseteq$ denotes the refinement relation on specifications. The argument given at instantiation must refine the types of the package type.

A view v can arbitrarily map type names contained in a package type to type names in an argument package, however this mapping is valid only if it preserves the refinement relation between specifications. If A and B are types and $A \sqsubseteq B$, then if $v(A)$, the image of A under v, is a valid view for instantiating a parameterized package, then $v(A) \sqsubseteq v(B)$.

3 Reuse *in the Small*

MooZPP provides an inheritance mechanism that favors class reuse. This mechanism allows class reuse between different type hierarchies (families). It avoids specifying from scratch classes that present some similarities even if these classes belong to different type hierarchies. In fact, class inheritance does not require the notion of subtyping to hold between parent and heir classes. This allows code reuse among classes that should not be related by subtyping, but that have enough similarities that justify the use of inheritance. So inherited code can be freely modified. A heir class has the freedom to modify the inherited code so that it has a different behaviour from the original.

Inheritance is more flexible, in terms of reuse, than subclassing because it does not require the preservation of semantics between classes. So, (specification) code can be reused when subclassing makes no sense. On the other hand, it might be possible to prove that properties hold between classes related by inheritance because semantic preservation is not compulsory.

Data types play a relevant role in structuring a specification since all operations related to the same set of values are grouped together and are syntactically delimited. This avoids scattering these operations over a specification text, and thus, improves the legibility and reuse of specifications. In MooZPP, data types are declared inside packages. So, they are available to other classes and data types defined inside the same package and to classes and data types specified inside other packages by means of package composition features.

4 Reuse *in the Large*

In this section we evaluate the impact on reuse caused by the introduction of features like packages in MooZPP. Packages and classes are complementary concepts [11, 21]. Classes only support reuse and maintenance *in the small*, whereas packages support reuse and maintenance *in the large*. This is based on the fact that whole classes hierarchies and related data types can be reused together through composition features. In other words, whole type hierarchies can be reused, preserving the relations (dependencies) they have among each other.

The horizontal and vertical modes of composition help to document the structure and dependencies of a specification. They can indirectly improve reusability since a well-structured specification is easier to understand. A specification structure is related to levels of abstraction. Import, which is a composition feature, can be classified into two modes: horizontal and vertical. The horizontal mode of import has to do with structuring a given level of abstraction whereas the vertical mode has to do with different levels of abstraction. This allows differentiating inherent and circumstantial concept dependence between packages. Complex horizontal and vertical structures of packages can be imported by other packages. So, import allows not only structuring, but also reusing complex structures.

In object-oriented programming languages such as Eiffel [2] there is no distinction between packages and classes. This is based on the idea that the encapsulation and abstraction provided by classes can substitute packages. However, other object-oriented programming languages support the concept of packages (packages in Java [17] and namespaces in C++ [3]). In fact, packages can enhance software reuse [11] beyond that supported by the object-oriented paradigm, as we have seen so far. PJava [6] is a proposal of parameterized packages for Java which uses concepts of parameterized programming [10] such as *views* and package renaming. PJava does not support the horizontal import mode and the package slicing operation that are supported by MooZPP.

4.1 Parameterized Packages

Parameterized packages allows grouping several related classes and data types, which can refer to declarations present in parameters. Instantiating a parameterized package yields a new package which can be combined with others packages by means of composition features.

A parameterized package can declare several classes and data types. Furthermore, parameterized packages put packages together to compose new packages. As each package can contain several related classes and data types, when packages are put together, several classes and data types are put together to compose several other related classes and data types.

A parameterized class puts classes together to compose a new class, whereas a parameterized package puts packages together to compose a new package. But, note that a single package can group the declaration of several types. Consequently, we achieve larger granularity for reuse.

Parameterized packages are at least as expressive as parameterized classes. In fact, a parameterized class can be considered a special case of a parameterized package: a package containing just one class and parameterized by package types, each of them containing a single class as well.

For software reuse to be effective, it is necessary to have the capability of adapting software components to contexts that are different from those ones in which they were specified. It is important to reuse software components differently in different contexts without having to modify the component itself. In MooZPP, *views* and *renaming* allows adapting software components to particular contexts.

4.2 The Role of *Views*

Views are mapping from declarations (classes, data types, and associated operations) of a package type to declarations of an argument package. The mapped declarations may have different names in the source (package type) and in the target (argument package), but they must have the same type. This contrasts with other approaches that can only accept arguments with operations having the same name and types [2]. In this way, it is necessary to know previously the parameter operation names in order to specify types with the same operation names so that they could be (valid) instantiation arguments.

In MooZPP, *views* are used to express adapter packages. However, differently from adapters, *views* do not introduce any new concept that would be used only for instantiation. Moreover, *views* are simpler to write than adapter packages.

4.3 Renaming

Renaming is fundamental in adapting packages and package types to new contexts. This improves reuse because already specified packages can be adapted to new contexts; it avoids rewriting packages just to give contextual relevant names to classes and data types. Renaming also serves the purpose of improving specification readability. Through renaming classes and data types (operation names can be renamed as well) can be given names that are relevant not only to the context of a specification itself, but also to the application domain. In this way names present in a specification can be easily related to the entities of the part of the real world that they are expected to represent.

5 Related Work

GenVoca [7] shares many of our concerns and distinguishes between components and “realms” which are our package types although without semantic constraints. GenVoca is primarily based on vertical parametrization, although a limited form of a horizontal parametrization allows constants and types without any horizontal composition. GenVoca does not involve specification, but is used, in fact, to generate systems supported by the P++ language, an extension of C++.

Another related work, RESOLVE [26], also shares many concepts we have used in the design of MooZPP. Horizontal and vertical import modes of MooZPP are also supported by RESOLVE by means of *addition* and *enhancement* of abstract components. However, in RESOLVE there is no way to delete an operation from an interface as we have in MooZPP. RESOLVE involves both specification (abstract components) and implementation (concrete components). The latter is not supported by MooZPP. Larch [25, 16] supports the horizontal and vertical modes of import. Parameterization is limited to the horizontal mode. There is no support for the concepts of *views* or other kind of adapter. Notice that this discussion is limited to components described using the Larch Shared Language.

In RESOLVE and in Larch, specified components are in some way implemented. MooZPP is an object-oriented specification language and so it cannot be used in all phases of software development. A method for refining MooZPP specification into a programming language is a way to bridge the gap between specification and implementation (a method for refining MooZ specifications into Eiffel programs was presented in [4]).

OOZE [1], a general wide spectrum language based on the notation and style of Z which presents modularization mechanisms similar to the ones presented here, MooZPP clearly distinguishes the vertical and horizontal modes of composition that help to document the structure and dependencies of specifications. Although OOZE presents three different kinds of modules (one for encapsulating declaration of classes, theories and data types), we can achieve the same functionality by using MooZPP packages. Both languages support *views* which are used when instantiating a parametrized package. OOZE allows complex methods to be defined only as sequential and *parallel* composition of already defined methods, whereas MooZPP conserves Z schema calculus, allowing operations such as conjunction and sequential composition of operation (method) schemas. So, MooZPP supports a more powerful schema calculus than the one of OOZE.

MooZPP is an extension of the model based language Z. MooZPP data types declare a set of values and operations on these values. In order to satisfy the constraints of a data type specified in a package type, we must provide a package that also contains a data type whose set of values has, at least, corresponding values of the data type of the package type. In OOZE, however, constraints on data types are not based on a single model. The functional module of OOZE consists of an order sorted signature, which gives sort and function symbols, and a set of equations that relate symbols of the signature. In order to satisfy a functional theory in OOZE, argument modules must provide data types that satisfy the equations of the theory. In summary, data types in MooZPP and the functional module of OOZE have a different semantics. This comes from the fact that OOZE is a derivative of OBJ [12] that also supports all concepts of parameterized programming.

MooZPP reuse capabilities are more powerful than those supported by Object-Z [22], an extension of the formal specification language Z with object-oriented concepts, because MooZPP supports a modularization mechanism at package level whereas Object-Z supports only modularization at class level. In fact a package in MooZPP can group several related classes and data types.

6 Conclusions

In this paper we discussed some design issues of an extension of the formal specification language MooZ. Such extension supports concepts of parameterized programming [10]. The introduction of parameterized packages enhanced MooZ's capabilities for reuse and maintenance. In fact, MooZPP modularization capabilities are more expressive and offer a better support for reuse and maintenance *in the large* than those provided by MooZ.

MooZPP supports, as we have seen, concepts of parameterized programming that are implemented with limitations in some programming languages. Object-oriented extensions of Z, e.g. Object-Z, do not support parameterized packages. Only OOZE supports parameterized packages, but it is an algebraic-based specification language (as it is a derivative of OBJ), not a model-based specification language like all other extensions of Z.

Some specifications using MooZPP have been developed, and the first results show that these features have a satisfactory behaviour, improving MooZ capabilities and expressiveness. The specification of real systems described in MooZ such as those presented in [8, 20] using MooZPP will serve to confirm our ideas based on comparison between specifications. We are now formally describing the semantics of the module language we presented here.

Acknowledgements

Thanks to the referees for the excellent comments which helped to improve the final version.

References

1. A. J. Alencar and J. A. Goguen. OOZE: An Object Oriented Z Environment. In *ECOOP'91 - V European Conference on Object-Oriented Programming*, Geneva - Switzerland, 1991. Springer-Verlag.
2. B. Meyer. *Object-Oriented Software Construction*. Prentide-Hall, second edition, 1997.
3. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.
4. V. A. de O. Cordeiro. From MooZ to Eiffel: A Rigorous Approach to System Development. Master's thesis, Universidade Federal de Pernambuco, Departmento de Informática, 1994. In Portuguese.
5. D. A. Watt. *Programming Languages Concepts and Paradigms*. C. A. R. Hoare Series Editor. Prentice Hall International, 1989.

6. D. Aranha and P. Borba. Parameterized packages and Java. *II Brazilian Symposium on Programming Languages*, pages 208–218, September 1997. Campinas, Brazil.
7. D. Batory, V. Singhal et. al. The GenVoca Model of Software-System Generators. *IEEE Software*, pages 89–94, September 1994.
8. G. H. M. B. Motta. Object-Oriented Formal Specifications: Application in the Development of an Effort Eletrocardiogram Processing System. Master's thesis, Departamento de Informática, Universidade Federal de Pernambuco, 1992. In Portuguese.
9. J. Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986.
10. 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.
11. J. Goguen and A. Socorro. Module composition and system design for the object paradigm. *Journal of Object-Oriented Programming*, 7(9), 1995.
12. J. Goguen and T. Winkler. Introducing OBJ3. *Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab*, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud, in *Applications of Algebraic Specification Using OBJ*, edited by Joseph Goguen.
13. J. M. Spivey. *Understanding Z: A specification language and its formal semantics.* Cambridge University Press, 1988.
14. J. M. Spivey. *The Z Notation: a reference manual.* C. A. Hoare Series Editor. Prentice Hall International, 2nd edition, 1992.
15. J. Woodcock and J. Davies. *Using Z - Specification, Refinement and Proof.* C. A. R. Hoare Series Editor. Prentice-Hall International, 1996.
16. J.V. Guttag, J.J. Horning and A. Modet. Report on the Larch Shared Language - Version 2.3. Technical report, Digital Equipment Corporation, 1990. SRC Research Report 58.
17. K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.
18. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.
19. 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.
20. P. D. de L. Machado. EASY — An Environment for Simulation of Artificial Neural Networks. Master's thesis, Departamento de Informática, Universidade Federal de Peranambuco, 1994. In Portuguese.
21. R. Borges and R. Ierusalimschy. Módulos em Linguages Orientadas a Objetos. *I Brazilian Symposium on Programming Languages*, pages 371–384, September 1996. Belo Horizonte, Brazil.
22. R. Duke, P. King, G. A. Rose and G. Smith. The Object-Z Specification Language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, Software Verification Center, April 1991.
23. 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.

24. 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 at http://www.di.ufpe.br/~mooz.
25. S.J. Garland, J.V. Guttag and J.J. Horning. An Overview of Larch. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 329–348. Springer-Verlag Lecture Notes in Computer Science, 1993.
26. W.F. Ogden, M. Sitaraman et. al. Special Feature: Component-Based Software Using RESOLVE. *Software Engineering Notes*, 19(4):21–67, October 1994.

This article was processed using the LaTeX macro package with LLNCS style