Experimenting Formal Methods through UML

P. Moura, R. Borges, A. Mota

Federal University of Pernambuco
Centre of Informatics
P.O.Box 7851 - Cidade Universitária
50740-540 - Recife - PE - Brazil

Abstract

In this paper we experience annotating UML class diagrams with fragments of the Object-Z specification language. This yields an encapsulation of formal methods in UML where modeling can be seen more practical and easy, and analysis could be automated straightforwardly by extending UML tools. We illustrate this approach by modeling a simple but very interesting object-oriented system as well as performing the required syntax and type checking, proof of desired properties, and investigation of data refinements, based on tool support.

1 Introduction

Nowadays, UML is a standard notation to document software artifacts [BRJ99]. This is probably due to its graphical notation as well as flexibility (lack of rigor). Indeed, there are software processes entirely based on the use of UML, such as the RUP [Kru00].

Contrarily, formal methods still suffer some rejection by the software Industry [CW96]. Partly because one has to use mathematical notation and partly due to its lack of integrated tool support.

Comparing UML and formal methods, clearly UML is seen as extremely easy to use (and consequently dangerous if used without care [CE97]), while formal methods are announced to be very difficult to apply in practice [Som02]. Therefore, bringing these two approaches together seems to be very promising.

In this paper we propose a practical use of formal methods where fragments of the Object-Z specification language [Smi00], an object-oriented version of the model-based language Z [Spi92], are embedded in UML class diagrams. This is enforced by the fact that integrating UML and Object-Z—by mapping the annotated UML elements to Object-Z constructs—naturally yields a formal semantics to UML [MS03]. Indeed, this is an overall goal of the project ForMULa (**For**mal **M**ethods and **U**M**L** Integr**a**tion) by which the present effort contributes.

Since formal methods need integrated tool support in order to be more acceptable by the software Industry, we have decided to experience our approach in a UML development tool. Our choice was the Rational Rose because it is a leader of market, UML static and dynamic diagrams could be easily drawn, and its add-in technologies allows us to extend its functionality for our specific purposes.

Our approach is very similar to the one proposed by IBM where the language OCL (Object Constraint Language) [WK99] is used as a standard formal specification language to formalize UML diagrams. However, OCL is still lacking a complete formal semantics and tool support beyond syntax and type checking [1]. Furthermore, the choice of Object-Z, which already has a formal semantics [Smi00], is a matter of convenience since OCL can be seen as a simple syntactical variant of a formal specification language with object-oriented support [DJM03].

As it is well known by the Z community, LaTeX is the standard input language for Z tools. Therefore, the Object-Z constructs used in this paper will appear as LaTeX elements. Nonetheless, for each Figure where such elements occur a corresponding graphical presentation will be given.

This paper is organized as follows. An overview of UML notation is discussed in section 2 using a simple but interesting object-oriented example. In section 3 we give a brief summary of introducing formal methods in UML class diagrams: we show how to annotate the diagram in section 3.1, how occurs the mapping from an annotated UML model to pure Object-Z as well as presents Object-Z itself very succinctly in 3.2; and we illustrate the use of tools to prove properties and refinements in these specifications in section 3.3. Tool support is explained in section 4. Finally, in section 5 we present our conclusions and future researches.

2 UML Overview

The Unified Modeling Language (UML), the standard graphical notation provided by the Object Management Group (OMG), is currently the most popular and widespread graphical modeling notation for object-oriented software development [BRJ99]. Indeed, there are software processes entirely based on the use of UML, such as the RUP [Kru00].

UML has various diagrams, each one with its own specific purpose. Some diagrams are provided for analysis and design, whereas others are related to implementation and distribution. The purpose of an UML diagram is to visualise some aspect of a software system easily. For instance, UML class diagrams are used to present the static view of a system, whereas state diagrams for the dynamic aspects.

Unfortunately, almost all UML diagrams do not have a formal semantics[2]. What it is common for the UML community is that UML diagrams have rigid rules on how they can be depicted. Such rules are stated using a meta-model description with a core of UML itself. However, these rules are not sufficient to provide more specific characteristics of software systems and this is one reason the UML community has proposed the OCL (Object Constraint Language) language [WK99]. OCL is a "formal" language used to restrict forbidden instances allowed by the UML graphical notation alone. Therefore, OCL is a textual language very similar to the LaTeXversion of Z used in tools. Indeed, the UML community is not using OCL as suggested by its creators.

In this paper we are concerned specifically with UML class diagrams. They are explained in the following section.

[1] In particular, OCL automated type checking is partial. That is, some terms are not decided to be type compatible or not.

[2] In particular, statechart diagrams have a formal semantics since they are based on statecharts [Harel].

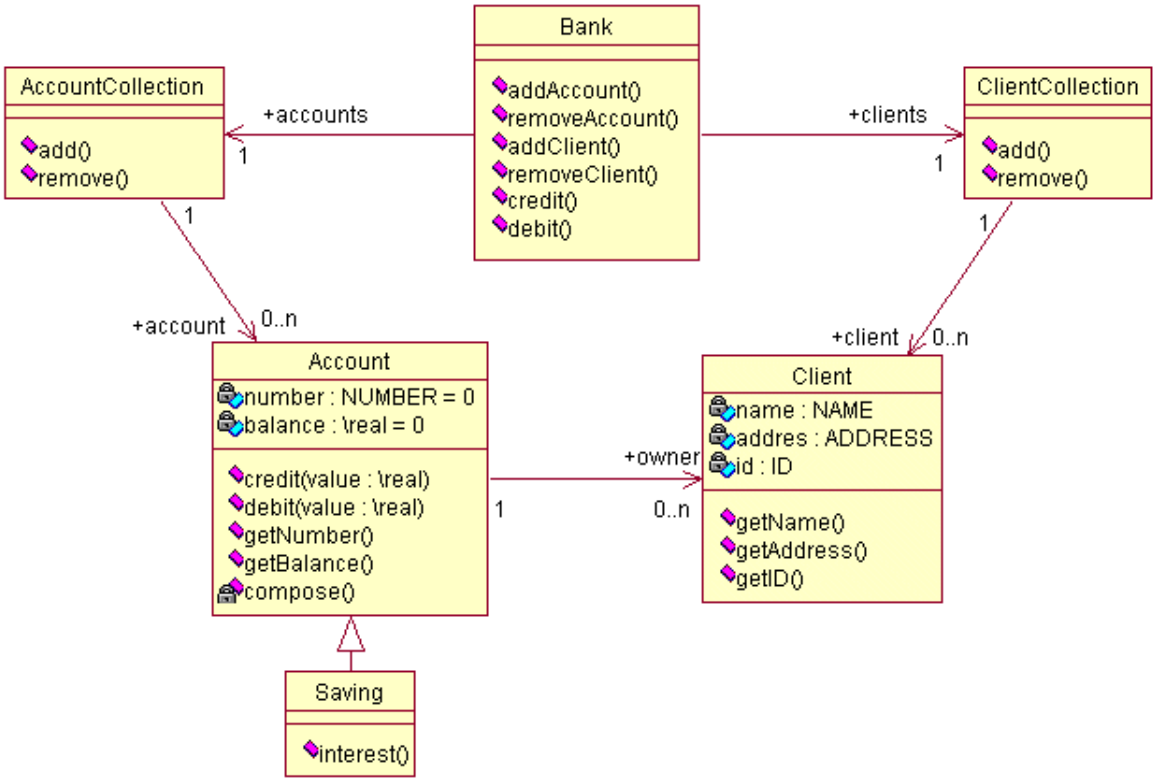


Figure 1: A UML class diagram for a banking system.

2.1 Class Diagrams

The static aspect of a system is generally catched by UML class diagrams. These diagrams are the most common diagrams found in software development projects.

In such diagrams, the specifier is concerned with classes, interfaces, collaborations, and relationships.

UML class diagrams are equipped with places to accomodate pre- and post-conditions of operations as well as class invariants, and general constraints as notes attached to any UML graphical element.

As it is well known to the Object-Z community, by providing operations with pre- and post-conditions one can capture behavioural aspects of a system as well. Therefore, we can use UML class diagrams to characterise both static and dynamic aspects of a system by annotating them with invariants, pre- and post-conditions.

Figure 1 presents a class diagram that models a banking system. This diagram uses almost all elements of UML class diagrams: classes, associations and inheritance.

2.2 Classes

A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class has a name, attributes and operations. Figure 1 also shows a class whose name is *Account*. This class has two attributes: *number* of type NUMBER, and *balance* of type $\mathbb{R}$ (*\real*), both invisible (private) outside the class. And to deal with these attributes , we can see four visible (public) methods: *credit* (with a real number parameter), *debit* (with a real number parameter), *getNumber*, and *getBalance*.

Figure 2: An example of refinement association.

The diagram shown in Figure 1 consists of a main entity, *Bank*, which has two attributes: *AccountCollecion* and *ClientCollection*, collections of *Account* and *Client* that become related through an association. Each class has its proper attributes and operations. *Saving* represents a generalization of *Account*.

2.3 Associations

In order to capture relationships among objects, UML class diagrams provides associations. An association is a structural relationship that specifies what objects of a type (class) are connected to objects of, possibly, another type.

In Figure 1 we have many examples of associations. The *Bank* class relates, at the same time, with the classes *AccountCollection* and *ClientCollenction* through two relationships. Each extremity of an association has a name of the role, a multiplicity, a class that the extremity is joined and an attribute to describe the navigability. The class *AccountCollection* has the role *account* and *ClientCollection* has the role *client*, both with multiplicity 1.

An interesting relationship between two classes is that of refinement. This relation is common for the formal methods community, where one can represent that some class is better than (refines) another class. Although this is not common for UML, we can characterize this this relationship by using a stereotype association, or an association with a specific purpose (stereotype). Figure 2 presents two versions of the class *ClientCollection*, where the left class lies in the package *abstract* and the right one in the package *concrete*. From the packages naming, it is obvious that we are trying to represent that the right class is better than (refines) the left one.

Generalization A generalization is a taxonomic relationship between a more general class (superclass) and one more specific (subclass). Figure 1 shows an example of generalization between the class *Account* (superclass) and the class *Saving* (subclass). Generalization is the UML element designed to characterize inheritance aspects.

3 Precise Modeling with UML

The specification language Object-Z [Smi00] is an extension of the model-based language Z [Spi92]. It was designed to support object-oriented features as well as originate modular specifications more easily for non-experienced users.

An Object-Z specification is built around a collection of classes, each having its own attributes and methods. More specifically, attributes are better known as the class state variables, and methods as class operations. To see what an Object-Z class looks like, we

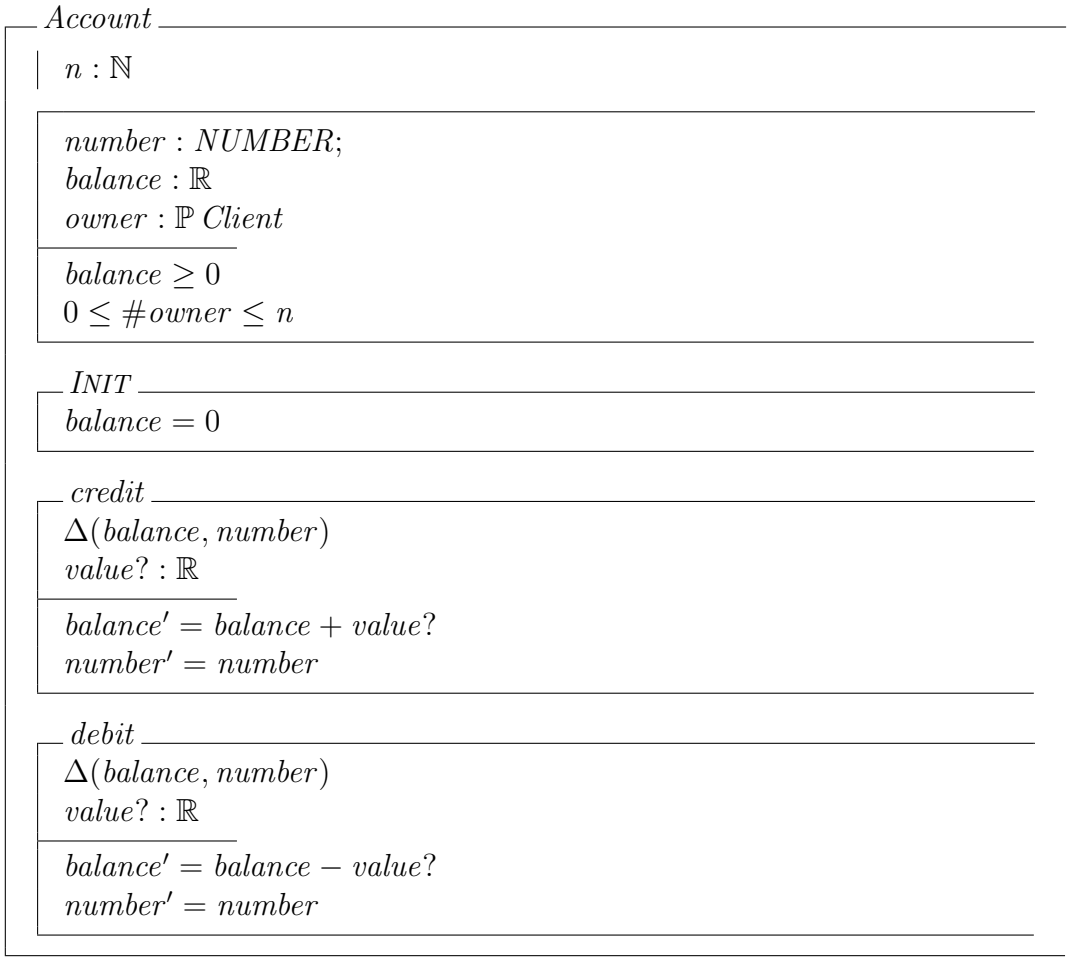


Figure 3: Object-Z class specification.

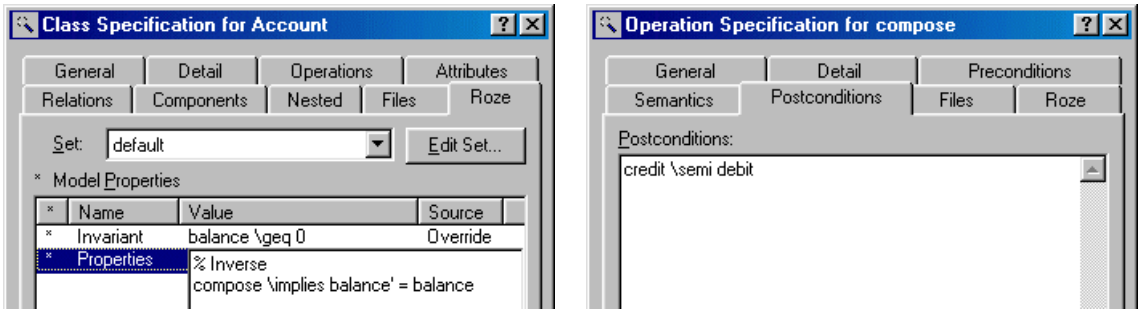
present the class *Account* of the diagram shown in Figure 3. Bite that the association *owner* appears as a class attribute and its multiplicity in the class invariant.

It is worth noting that object's initialization is performed by the operation *Init*.

3.1 Annotating UML Class Diagrams with Object-Z

From the previous section we can see that Object-Z classes support the elements of a UML class diagram. Additionally, an Object-Z class has the advantage to be completely formal whereas UML are not. Thus, following the combination defined in [MS03], in this section we present how UML class diagrams can be annotated by Object-Z elements in such a way that we can transform this annotated UML class diagram into a pure Object-Z specification.

Since the present work is oriented towards industrial applications, our annotation approach is presented using the Rational Rose CASE tool.



(a) (b)

Figure 4: Screen shot of Account class specification.

3.1.1 Annotating Classes

For each class we need to define the invariant and properties which must be satisfied. In order to do that, we have extended the Rose tool to support two new fields: *Invariant* and *Properties*. Figure 4 (a) illustrates the *Properties* field, where only the property

Theorem *Inverse*
$compose \Rightarrow balance' = balance$[3]

is stated. It is worth noting that *compose* is indeed the composite Z operation $credit \fatsemi debit$ as illustrated in Figure 4 (b). This is actually a Z grammar restriction which forced us to create a boolean field (isDefinition) indicating whether an operation is single or composite as explained below. This property captures the notion that applying a *credit* operation, followed by a *debit* operation, using the same amount of money, does not change the class state. It is worth observing that both the *Invariant* and the *Properties* fields are located in the tab Roze of a class specification window. Invariants can also be associated to class attributes and are accessed from the attributes window.

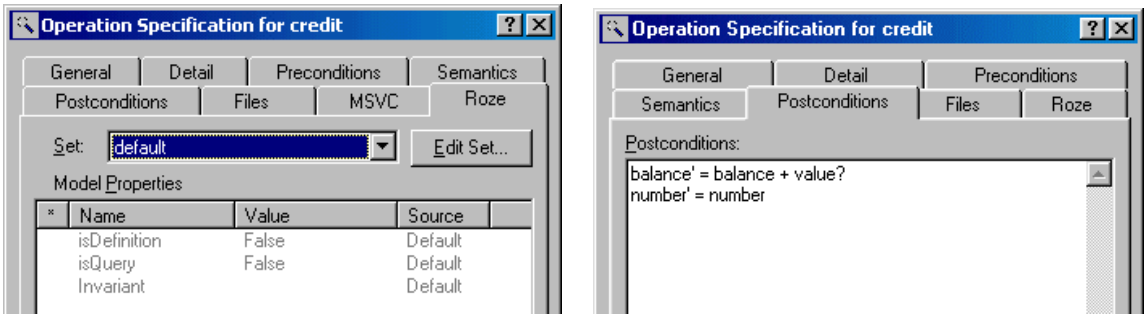
Regarding operations, each one has its own pre- and post-conditions, that establish the conditions under which the operation must operate. As pointed out previously, we have two choices to define an operation: we can define a single operation (a schema as in Figure 5 (b)) or a composite operation ($compose \mathrel{\widehat=} credit \fatsemi debit$ as in Figure 4 (b)). For simplification purposes, we reuse the Rose´s tabbings for pre- and post-conditions for single or composite operations. However, composite operations have the pre-condition tabbing always empty because Object-Z does not make explicit distinction between pre- and post-conditions. As composite operations depend on single operations then single pre- and post-conditions are simply expanded according to the rules of the Z schema calculus.

Although the Rose tool has predefined locations for stating pre- and post-conditions, other elements are necessary to allow the annotations to exploit certain features of Object-Z.

Figure 5 (a) shows the specification of the operation *credit* where we emphasize the extra fields:

- IsQuery—it is a boolean value used to characterize whether the operation modifies (when set to false) or not (when true; in this case, we must enforce that all attributes

[3] *compose* is used in place of $credit \fatsemi debit$ due to Z grammar restrictions.



(a) (b)

Figure 5: Screen shot of credit operation specification of the class Account.

must not change by an operation execution) the class state; the default value is set to false. It is included in the UML specification [OMG03], but is not implemented in Rose.

- IsDefinition—it is a boolean value indicating whether the current operation is describing a compound Object-Z operation. That is, an operation built around the composition of other operations using Z schema definition.

3.1.2 Annotating Associations

Associations also need specific fields to insert formal notation. UML associations are interpreted as being attribute of the other class, if is the case that the final class is navigable from the initial class. Therefore the necessity exists to also stipulate invariants for the associations between classes.

We define two fields where invariants on each end of the association can be inserted. So each role, that in the future will be mapped as an attribute of the other class of the association, will have individual fields for its invariant. In Figure 6 we see the fields *RoleAInvariant* and *RoleBInvariant*, which are exactly the fields of the invariant for the roles A and B respectively.

We represent a data refinement in UML through a special association with stereotype *refine*. Since a data refinement needs a retrieve relation, the refine association has a field named *Retrieve* where we place the predicate relating abstract and concrete class states. Figure 6 also presents the field *Retrieve* in specification window of an association.

Depending on the multiplicity of each single association end, the corresponding role can represent single object or collection of objects of the same class.

Since we represent associations through Object-Z class attributes, we need to know the right type o use in each case. In general, for a single instance we use a type T and for a collection the type $\mathbb{P}\, T$, seq T or bag T. However, this point has revealed an interesting consideration concerning refinement. Imagine a design decision that uses a sequence, instead of a set, to keep the collection of objects. At this point we have two choices: either considering this decision directly in Object-Z or adding another field to characterize the type an association must be interpreted. In order to support this kind of data refinement and still use the graphical notation of UML, we decided to consider the second choice. Thus, Figure 6 shows two fields, *TypeOfRoleA* and *TypeOfRoleB*, used to accomplish such a design goal.

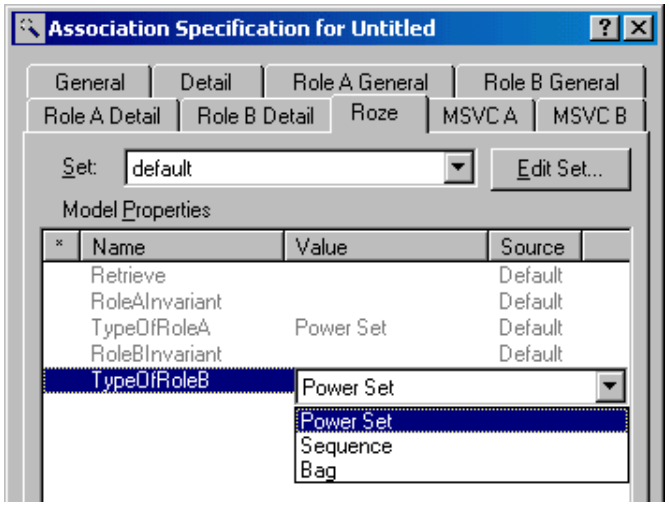


Figure 6: Association Specification Window.

3.2 From Annotated UML Class Diagrams to pure Object-Z

Since both UML and Object-Z are based on [Boo91], we can verify that the mapping between them is almost direct although there are some special cases reported in [MS03]. In this section, we show how our annotations described in section 3.1 originate a pure Object-Z specification. For ease of presentation, the translations are related to the UML class diagram of Figure 1.

The formal mapping between UML classes and Object-Z classes is based on [KC00] and [MS03], which provides a formal mapping between them. In particular, we will discuss here the UML constructs shown in section 2.

When we interpret the class or the association as a component of the diagram, they represent the set of existing instances of that class or association at certain point. Thus, we can map the class diagram as the set of all classes, associations and generalizations.

3.2.1 Classes

An Object-Z class is generated from an annotated UML class by taking the class name as the Object-Z class name; the operations in the UML class as operation schemas in the Object-Z class; the attributes form the state schema (since the attributes in the UML class represents the state of the object) and out created new field *Invariant* serves to register the predicate to be put in the predicate part of the Object-Z class state; and the visibility of the attributes we put in the visibility list. For each Object-Z operation, we simply join together the contents of the fields pre- and post-condition of the UML class specification, since Object-Z operations do not distinguish pre- and post-conditions.

3.2.2 Associations

Object-Z has no direct support to associations. Nevertheless associations can be captured by class attributes. This is exactly our mean of representation.

Using the association between *Account* and *Client* from the Figure 1, where one *Account* may be related with many *Clients*, we could show that an UML association may be mapped to an Object-Z attribute. In the example the *Client* appears as the attribute named *owner*.

It is a power set of *Client* and has an associated invariant that represents the cardinality of the relation. This can also be seen in Figure 3.

Generalizations This is the easiest feature to be mapped to. The Object-Z class

Saving
Account
...

corresponds to the relationship between *Saving* and *Account*. Note that apply since both approaches supports the concept of generalization, a map among them is easy: we just take the superclass of a class in the UML diagram and put its name in the inherited field in the Object-Z corresponding class.

3.3 Formal Development

Differently of traditional object-oriented development, a formal development has important tasks which guarantee the overall software product. The most common are the proof of desired properties and refinement of specifications. The first effort is in the direction of investigating the satisfiability of software requirements. The latter concerns the stepwise development approach, which originates concrete preserving versions of the original abstract specification.

From the practical viewpoint, the previous tasks are handled using tool support, such as type-checkers, theorem provers, and model checkers. Unfortunately, Object-Z has a limited set of tools. Basically, Object-Z has a type-checker named Wizard [Smi96], a model checking strategy, and a non-direct theorem prover provided by an Object-Z embedding in the HOL system.

However, Object-Z is an extension of Z and thus, for a reasonable subset of Object-Z, it is possible to use the Z tools. In particular, single Object-Z classes can be seen as simple Z specifications. This is exactly the approach ew take in the present work. We use the Wizard tool to perform syntax and type-checking, and the theorem prover Z-Eves to check for valid properties as well as data refinements.

In the rest of this section we point out our current experience of a formal development approach using the UML graphical context.

Our starting point is syntax and type-checking. This is the easiest task to be accomplished since the Wizard tool is a proper Object-Z tool. We simply have to transform the annotated class diagram into a pure Object-Z specification and execute the Wizard tool. This simple verification step can reveal a lot of typographical problems in the annotations.

Due to the lack of tools to check Object-Z specifications, we adopted the use of Z to prove properties and refinements. As cited in [SBC92], the semantics of an Object-Z class is similar to that of a Z specification with only one state schema. So, after the mapping of classes in UML to Object-Z, to the purpose of experimentating formal methods embedded in UML, we can investigate properties and prove refinement of one class in the diagram.

Since we are working with one class in Z, some of the features introduced by Object-Z are lost. One of these is the visibility: the members do not need visibility, because they can be seen inside the class, independently of its implicit modifier and there are no other classes that will try to access hidden class members. The other is associations: the classes

which are in the other end of an association with the generated classes are treated as given sets; this weaks our model, but for the purpose of the experimentations, this is enough.

In this section we will use the example given in Figure 2. So, we need to specify all operations in Z.

add

$$\Delta(ClientSet)$$
$$c? : Client$$
$$c? \notin clientSet$$
$$clientSet' = clientSet \cup \{c?\}$$

remove

$$\Delta(ClientSet)$$
$$c? : Client$$
$$c? \in clientSet$$
$$clientSet' = clientSet \setminus \{c?\}$$

The most precise things that we can extract from the problem are its properties. When we are constructing a model for some problem, we want that our model represents it and we would like to check if it have the desired properties. In a formal specification, the properties can be used to verify the correctness of the model.

In the previous example, we will try to check a simple property: the Inverse property. Given the sequential application of the add and remove operations, the final state remains unchanged in relation to the initial. To do this in an UML model, we added the fields *Property* and *isDefinition* in the class specification. *Property*, as the name says, serves to specify the properties of the class. *IsDefinition* serves to introduce an operation which sequentially composes others two. This is necessary for formal specification but does not affect the UML model, since that auxiliary operation can be made private, fitting well the UML methodology.

Theorem *Inverse*

$add \fatsemi remove \Rightarrow clientSet' = clientSet$

Once we have written an abstract specification and proved its correctness, we would like to improve it, giving more details about the state space and its associated operations, replacing mathematical structures by computer-oriented structures. We must be more precise about storing data or making calculations. But this is not enough: we need to establish a relationship between the two state schemas. This relationship is called retrieve and, through it, we must prove that the behavior of the concrete specification is the same of abstract. In the UML model, we insert the retrieve into a new field, the *Retrieve*, which is associated with a refinement association. Since we are mapping UML classes to Z specifications, this refinement will be done between two classes in the class diagram. The operations' specifications of the new class, which refines the prior are presented. The Retrieve is also shown, where we establish a refinement association between the abstract and concrete *ClientCollection* classes.

add

$\Delta(clientSeq)$
$c? : Client$

$c? \notin \text{ran } clientSeq$
$clientSeq' = clientSeq \frown \langle c? \rangle$

remove

$\Delta(clientSeq)$
$c? : Client$

$c? \in \text{ran } clientSeq$
$clientSeq' = clientSeq \upharpoonright (\text{ran } clientSeq \setminus \{c?\})$

Retrieve

$clientSet = \text{ran } clientSeq$

In this particular concrete specification, we can see sequences as some kind of linked lists, the concatenation of sequences as concatenation of lists and the filter as a traverse in list eliminating the nodes which do not satisfy some predicate.

Following this line, we can build classes which are more and more operation oriented, proving refinement between them, until we have an almost operational design. Then, we can apply a refinement calculus and derives the code from the specification [WD96]. All of these steps can be assisted by CASE tools, like Rose.

4 Tool Support

We developed, from the Rational Rose extension interface, one add-in responsible for the generation of specifications, theorems and properties associated in Object-Z/Z. The add-in was written in Visual Basic, using a modular structure aiming modules reuses in the diverse functionalities of the tool. It was also necessary to create archives to describe the extra menus and properties that had been inserted.

A Web interface was also developed to submit specifications written in LaTeX for the Wizard, thus making possible use of any operational system to analyze and type check its specifications. This is done through a simple and universal interface.

The RoZe is available and can be downloaded from the RoZe page[4], and the Wizard interface can be accessed through the the Wizard page[5].

5 Conclusion

In this paper we have proposed a practical use of formal methods where fragments of the Object-Z specification language [Smi00] are embedded in UML class diagrams. This work is in the direction of the project ForMULa (**For**mal **M**ethods and **UML** Integr**a**tion).

[4] www.cin.ufpe.br/∼rmb2/Arquivos/IC-MF
[5] www.cin.ufpe.br/∼pvsm/IC-MF/typeCheckerOnline.html

Since formal methods need integrated tool support in order to be more acceptable by the software Industry, we have decided to experience our approach in the Rational Rose where its add-in technologies allows us to extend its functionality for our specific purposes.

Our approach is very similar to the one proposed by IBM where the language OCL (Object Constraint Language) [WK99] is used as a standard formal specification language to formalize UML diagrams. In particular, we indend to use OCL as specification language (front-end) although reasoning with a translation to Object-Z [DJM03]. The reason is simple, OCL is the OMG specification language standard but still lacks a complete formal semantics and tool support as well as capabilities to prove properties and refinements.

As future research we intend to integrate our current approach to model checking Object-Z specifications, incorporate refactoring and refinement laws in the tool, and incorporate treatment for dynamical aspects by using CSP.

Acknowledgement. We would like to thank Augusto Sampaio for his comments on earlier drafts of this paper.

References

[Boo91] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin–Cummings, Redwood City, Calif., 1st edition, 1991.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[CE97] T. Clark and A. Evans. Foundations of the Unified Modeling Language. In *2nd Northern Formal Methods Workshop*. Springer-Verlag, 1997.

[CW96] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 1996.

[DJM03] R. Duarte, J. Júnior, and A. Mota. Precise Modelling with UML: Why OCL? *submitted to the Workshop of Formal Methods*, 2003.

[KC00] S. Kim and D. Carrington. A formal mapping between UML models and Object-Z specifications. *Lecture Notes in Computer Science*, 1878:2–21, 2000.

[Kru00] P. Kruchten. *An Introduction to the Rational Unified Process*. Addison-Wesley, 2000.

[MS03] A. Mota and A. Sampaio. Integrating Object-Z and a subset of UML. *submitted to the Workshop on Critical Systems Development with UML*, 2003.

[OMG03] OMG. Unified modeling language. Specification v1.5, Object Management Group, March 2003. http://www.omg.org/technology/uml/.

[SBC92] S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.

[Smi96] G. Smith. Wizard: A Type-Checker for Object-Z Specifications. Technical Report SVRC 96-24, The University of Queensland, 1996.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[Som02] I. Sommerville. *Software Engineering*. Addison-Wesley, 2002.

[Spi92] M. Spivey. *The Z Notation*. Prentice-Hall, 1992.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.