Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 97–112
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Integrating UML and Formal Methods

Rafael Magalhães Borges[1] Alexandre Cabral Mota[2]

Centre of Informatics
Federal University of Pernambuco
Recife, Brazil

Abstract

UML is a widespread language used in both industry and academia, despite the fact that its semantics is still informal and allows ambiguities. On the other hand, *OhCircus* is a formal specification language which unifies Z, CSP, the refinement calculus of Morgan and object-oriented theories. In this work we integrate UML class diagrams and *OhCircus* by written UML elements in terms of *OhCircus* constructs. However, instead of a simply syntactical mapping, we also propose the concept of a *class model* to capture associations and global constraints. Finally, we use this integration to prove the refinement of associations as attributes, a result that relates analysis to design to implementation and which is very common in industry.

Keywords: UML, *OhCircus*, formal methods, translation, refinement.

1 Introduction

Formal Methods have proven effective in the development of critical systems [30,13,2]. However, they are not used in large scale due to many factors, specially their strong mathematical basis [5,28]. This represents a big obstacle to its widespread use.

Among various initiatives to make Formal Methods more accessible and used in industry, the current research direction is using a graphical and appealing language, such as UML, to encapsulate formal notations. This is usually accomplished by providing a mapping to constructions of a (often informal) language into another, a more formal one [17,21,3,14]. Thus, popular modelling languages, well-known by developers, are mapped into more powerful and formal, designed without major conceptual restrictions (although some are necessary for their practical usage).

As informal notation, UML [18,20] deserves special attention. It is composed of graphical elements to represent the variety of software entities and their relationship. Thanks to its apparent simplicity and ease of use, UML has become a *de facto*

[1] Email: rmb2@cin.ufpe.br
[2] Email: acm@cin.ufpe.br

1571-0661 © 2007 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2007.03.017

standard; nevertheless it can also express ambiguities and is insufficient to represent even simpler properties, as pointed out in [19].

OhCircus [4] was chosen as our formal language because it has an intuitive representation of constructions like classes and inheritance and is based on a refinement theory (the refinement calculus of Morgan [16]). These same arguments discarded languages like Z [29] and Object-Z [27], as stated in [4]. Furthermore, *OhCircus* is a language which integrates well-established concepts on the formal community: the model-based language Z, the process algebra CSP [22], the refinement calculus [16], and object-orientation, providing an unified language of classes and processes. Some of its design decisions came from UML-RT [3] [26], which turns it even more appropriate for out intended mapping.

Our goal in this work is the translation of UML class diagrams elements into *OhCircus* constructs. Our premises are to preserve all diagram structure, including their relationships and global invariants. This is achieved through a meta-class, syntactically equivalent to any other class, but that captures the overall structure. The main motivation is the exploration of refinement in UML [25].

Our approach differs on treating UML in the same semantic level of the formal language. For example, UML classes are mapped into classes in our chosen language. Other works, like those from the most important group in the area, offer a *denotational semantics* of UML in Z [9,7].

Finally, many works [3,6,15], including [20], assume the equivalence between associations and attributes as valid whereas we propose the use of associations as an abstract view of the class diagram. *These associations are eliminated along the refinement process through the introduction of attributes in the classes which participate in these associations.* This is a further contribution in the sense that, in addition to the result itself, we give and consolidate insight about our mapping and its notion of class model.

This works is organised as follow. In Sections 2 and 3, we describe the main elements of UML class diagrams and *OhCircus* specifications, respectively. In Section 4, we present our first contribution: the translation of a UML class diagram into an *OhCircus* specification (where we show our concept of the class model). The second contribution appears in Section 5, where we address refinement in UML diagrams using *associations-as-attributes*. Finally, we present our conclusion and future works in Section 6.

2 UML

The *Unified Modelling Language* (UML) is the language proposed by the OMG (*Object Management Group*) for modelling systems. It became a *de facto* standard because of its ease of use and intuitive notation.

A UML model represents the description of a set of objects which takes part in an application and the interactions to which they are submitted over time. A

[3] UML-RT is an UML extension that deals with concurrency.

snapshot of this set of objects and their instantaneous interactions represents a *configuration* of the system, and the collection of all possible configurations denotes the *semantics* of the model. So, a model can be seen as the description of a system.

UML is compound of several diagrams that express static and dynamic aspects of an application. Static aspects are related to the structure of the system, being true all the time. The purpose is to describe the entities of a system and how they will *always* be related. On the other hand, dynamic aspects refer to the evolution of the application: the creation and destruction of objects and their connections over time; formally, the *transformations* in the global state (the set of objects and relations) of the system.

2.1 Class Diagram

Class diagrams are the most common diagrams used in software development projects. They model concepts from the domain of the application and the structural aspects of the system using classifiers and relationships as their building blocks. They are also named *static view*, representing information that never changes.

The following example will be used throughout this paper. It is a simplified banking system that, although not exhaustive, exemplifies the main entities of class diagrams (Figure 1). In this model, we establish that persons own accounts or credit accounts. Persons, accounts, and credit accounts are represented by, respectively, the entities *Person*, *Account*, and *CreditAccount*. The ownership property is modelled through the *owns* relationship. Lastly, a (anonymous) relationship states that a credit account is a *specialisation* of a conventional account.

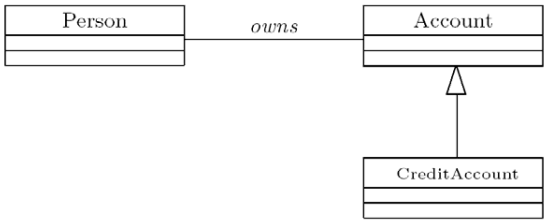

Figure 1. Static aspects.

2.1.1 Classifiers

A *Classifier* is a meta-class that groups all constructions which classify values. Its purpose is to introduce entities in diagrams. Classifiers are composed of *members*, which state behavioural and/or structural characteristics. Its main subclasses are *Class* and *DataType*. Other subclasses (like *Interface*) will not be considered in this work.

Classes

Classes are the key elements in a class diagram. They represent concepts inside the system's domain and introduce new types in the model. They describe the

structure and behaviour of a set of objects using fields, associations, and methods. In Figure 2 we can observe the class *Account*.

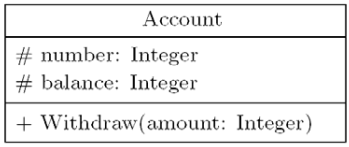

Figure 2. Class *Account*.

The fields describe values that the objects of a class contain. Each field has a type and, if desired, an initial value. The class *Account* of the above figure has two fields: *number* and *balance*. Each one holds an *Integer* value.

The methods of a class represent the implementation of operations. They embody a transformation on the state of the object from which they were invoked. They have a list of parameters and a return type. For instance, the class *Account* of Figure 2 has the method *Withdraw* that performs withdrawals, taking as input the amount and modifying the balance accordingly.

Visibility is a property shared by fields and methods (the class' members) that determines the accessibility of the member by other entities. Members can be private (denoted by −), protected (#), or public (+), being visible in the scope of the class itself, its subclasses, or any entity in the model, respectively. In the example of Figure 2, the fields of the class *Account* are protected while its method is public.

Data types

Data types (also known as *primitive types*) comprise values that are free from side-effects and do not have identity. Thus, two values that have the same representation are indistinguishable. They often model mathematical domains and their values are immutable. It is worth noting that the value stored in a field can be updated, but the value itself cannot. In UML, the numeric types, *strings*, and booleans are the predefined primitive types.

2.1.2 Relationships

Relationships denote semantical connections among the elements of the model. UML provides several ways to express these links, being associations and generalisations the main ones. Associations characterise structural relations among instances while generalisations create taxonomy among them.

Associations

An association establishes a structural relation between two classifiers. The associations may be named and have two endpoints (*association ends*), held by classes, to which behaviour can be designated ("roles") [4].

[4] In UML you can also get associations with more than two endpoints. However, they are uncommon and do not have simple semantics as the binary ones. Thus, we are not concerned with them in this paper.

In Figure 3, the association whose name is *owns* has two endpoints. The first one is held by the classifier *Person*, which plays the role of *owners*. The second one is *Account*, whose role is *accounts*. Semantically, *owns* relates instances of *Person* and *Account*.

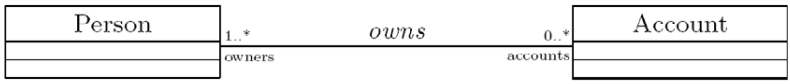

Figure 3. *owns* association.

Multiplicity imposes a constraint on the number of entities of an endpoint that are related to a single element of the other. The cardinality of this set can also be expressed using ranges. For instance, consider the *owns* relationship, where we can relate a *Person* to zero or more (*0..**) instances of *Account*; and for each instance of *Account* it must relate one or more instances of *Person*. The latter constraint enforces that every account must have at least one owner.

Navigability defines visibility for associations. The entity in one endpoint is seen by the opposite entity if the association between them is navigable; otherwise, that entity cannot state anything about the instances to which it is associated. Navigability is no further discussed.

There are other kinds of associations, like association classes, recursive associations, and qualified associations, that, although included in the translation, are not discussed here [5].

Generalisations

Generalisations capture inheritance relations between a more general class (superclass) and a more specific one (subclass). In fact, all members held by the superclass are inherited by the subclass. This relationship also states that every instance of the subclass is also an instance of the superclass. It is worth noting that we are interested only in simple generalisations, where classes can have only one superclass.

In Figure 4, we can see a generalisation relation between *CreditAccount* and *Account*. All members of *Account* have been inherited by *CreditAccount*. Because of this relationship, credit accounts can participate in the *owns* association, indistinguishable from conventional accounts.

3 OhCircus

Formal methods comprehend an area of computer science that provides formal interpretations to the diverse aspects of a program, like data types and concurrency. Z and CSP, for example, are two of the most used formalisms in industry.

The investigation of each aspect individually is very important, but the current attitude is to unify several formalisms and verify the influence of one above the

[5] For further information, please see [1].

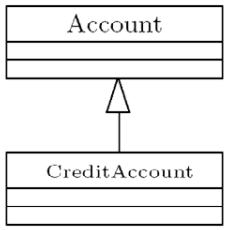

Figure 4. Generalisation relationship between *CreditAccount* and *Account*.

others. In particular, the attempts of integration between Z and process algebras (CSP-Z, CSP-OZ [8]) consider state and communicating aspects of concurrent systems in an unified language, taking advantage of the existing theories and tools. A similar formalism is *Circus* [31], which proposes a *refinement calculus* to that integration. Moreover, it is familiar to those who knows Z and CSP and enables the reuse of existing and well-established tools, like FDR [10] and Z/EVES [24]. *OhCircus* is an extension of *Circus* which adds classes, inheritance, dynamic binding and other features from the object-oriented paradigm.

3.1 Classes and Inheritance

A program in *OhCircus* is a sequence of paragraphs (much like Z and Circus) that defines classes and processes. To illustrate the object-oriented features[6], the next paragraph introduces the class *CreditAccount*, which models banking accounts that offer credit to the customers.

$$\textbf{class}\ \mathit{CreditAccount}\ \textbf{extends}\ \mathit{Account} \mathrel{\widehat=} \textbf{begin}$$

In *OhCircus*, a class declaration is introduced with the keyword **class**, followed by its name and the optional clause **extends**. This last part enables inheritance between classes; if omitted, the class inherits from the special class **object**. In this example, *CreditAccount* extends (inherits from) *Account*.

A class in *OhCircus* is quite similar to a Z specification. Fields, constructor, and methods are also introduced through paragraphs, often in the form of schemas.

state *CreditAccountState*

$$\textbf{private}\ \mathit{credit} : \mathbb{Z}$$

$$\mathit{balance} + \mathit{credit} \geq 0$$

The **state** clause indicates the schema which defines the state of a class. This schema is similar to that of Z, though its variable declarations can also contain qualifiers. If nothing is said, fields are assumed **private**. We can also declare them as **protected** or **public**. Despite its stated semantics, the modifiers do not constrain the access to the fields in *OhCircus*.

[6] Concurrency features are beyond the scope of this article.

Implicitly, the *CreditAccount* state inherits all fields and invariants declared in the state of its superclass. Moreover, new fields and invariants can be defined. For example, the *CreditAccountState* introduces a new private field, *credit*, and states that the sum of balance and credit cannot be negative. Note that the invariant is made up of fields from the subclass and also from the superclass.

public *Withdraw*

$\Delta CreditAccount$

$amount? : \mathbb{N}$

$amount? \leq balance + credit$

$balance' = balance - amount?$

$number' = number$

Methods are differentiated from other paragraphs by the use of **private**, **protected**, **public**, or **logical** qualifiers. The first three are directly related to the visibility of the method, again with standard meaning. The logical methods are just specification artefacts, useful for the calculation of complex expressions, for instance, but not necessarily present in the implementation.

Similar to Z, the methods of an *OhCircus* class interact with the state, modifying it (Δ) or not (Ξ). If a method is redefined, there is no inclusion of the superclass method into the new specification. However, it is necessary that the new definition retains the original behaviour. In other words, the new specification must be a *refinement* of the original one.

The operation *Withdraw* needs to encompass the new situation stated by accounts which have credit: it is not required that the balance is sufficient; only that the withdrawal does not surpass the limit. Note the weakening of the pre-condition and how it expresses a refinement of the original operation.

Methods which are not redefined are implicitly inherited by the new class, with the safeguard that they do not modify the components introduced by the new state.

end

Every class declaration is completed with an **end** clause.

3.2 Associations

To represent the banking application itself, it is also necessary to specify the class *Bank*, relating accounts and customers. Particularly, we present only its state about the *Person*, *Account*, and *owns* entities [7] (Figure 3).

[7] Note that *OhCircus* does not have a constructor equivalent to the UML association.

state *Bank*

$$
\begin{array}{l}
accounts : \mathbb{P}\, Account \\
persons : \mathbb{P}\, Person \\
owns : Person \leftrightarrow Account \\
\hline
owns \in persons \leftrightarrow accounts \\
\forall\, a : accounts \bullet \#owns^{\sim}(\!|\, \{a\} \,|\!) \in \mathbb{N}_1
\end{array}
$$

The fields of the class *Bank* include sets of accounts and customers and a relationship between accounts and customers. The invariant states that only enrolled accounts and customers can participate in the relationship, and that every account must be associated with at least one owner.

4 Mapping

In *OhCircus*, the "static view" is captured directly through its set of classes. As seen previously they have invariants, fields, and methods, like those of UML. However, the interaction between two classes is only captured through fields, as long as there is no notion of *association*. For example, recall that in the previous section the class *Bank* serves as the link between *Account* and *Person*. Other important aspect is that, in *OhCircus*, it is not possible to establish global invariants. Classes can only constrain their own states.

Our solution to these problems is the introduction of a class named *Model*, responsible for capturing all the structure of a class diagram: the sets of instances of classes, the relationships, and the global invariants. We believe that this approach offers a more abstract view of the class diagrams when compared to others ([6,15,3]), which only consider the representation of the classes, capturing associations directly through fields and ignoring global invariants. Note that the class *Model* is not part of the class diagram itself; it arises from our interpretation of the diagram. Thus, it is a kind of *meta-class*.

Illustrating our earlier discussion about capturing the association between accounts and persons, we present the state of a class *Model* that would realise our purpose.

state *ModelState*

$$
\begin{array}{l}
persons : \mathbb{P}\, Person \\
accounts : \mathbb{P}\, Account \\
owns : Person \leftrightarrow Account \\
\hline
owns \in persons \leftrightarrow accounts \\
\forall\, a : accounts \bullet \#(owns^{\sim}(\!|\, \{a\} \,|\!)) \in \mathbb{N}_1
\end{array}
$$

Note how this class is similar to the class *Bank* of the previous section. Furthermore, observe that this class represents the semantics of a class diagram similarly

to [18,20]. Each possible value denoted in this class reproduces some valid configuration of the class diagram. In other words, every instance of *Model* reflects an object diagram.

4.1 Classes

Once they have the same constructions, like fields and methods, the UML classes are easily mapped into *OhCircus* ones, though some constraints must be imposed. For instance, in Figure 5 we have the class *CreditAccount* mapped into an *OhCircus* class.

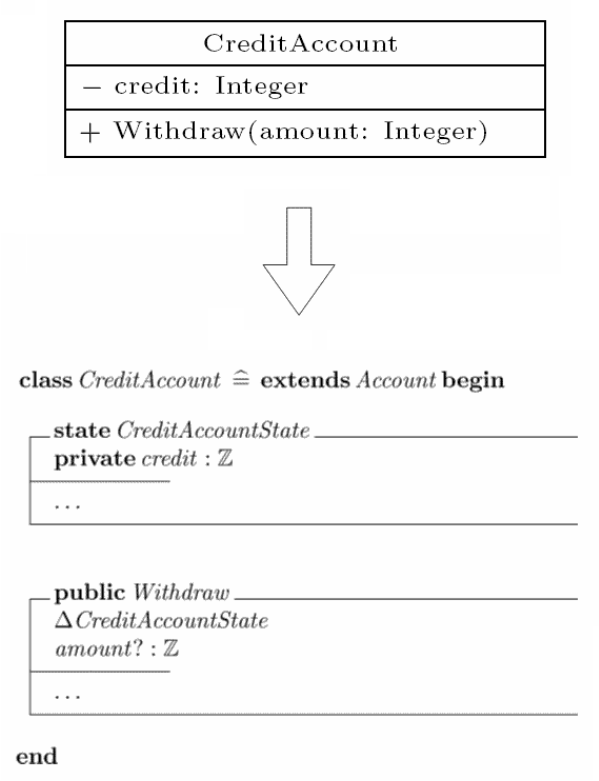

Figure 5. Mapping the class *CreditAccount*.

The fields of a UML class are mapped into schema state variables (the **state** *schema*) of a *OhCircus* class as well as the visibilities and types of these fields. Observe how the field *credit* of class *CreditAccount* is directly mapped into a variable of the schema *CreditAccountState*.

Methods are mapped into operation schemas. By default, these schemas modify the state (Δ-schemas) except when labelled *query*; in this case, they are mapped into Ξ-schemas. The input parameters are mapped into input variables (decorated with ?) while the result of the method, if present, is mapped into the variable *result*!. This transformation can also be observed in Figure 5.

state *ModelState*

$creditAccounts : \mathbb{P}\ CreditAccount$

$\ldots$

As suggested before, each class introduces into *Model* a set of instances, directly represented through power-sets. From the previous state schema, we can see the set of instances of the class *CreditAccount*.

Another interesting point is that if the type of a field is also a class, then it also introduces invariants in the class *Model*. These invariants assert that the values of these fields must be contained into their respective set of instances.

4.2 Data types

UML has four basic data types: `Integer`, `UnlimitedNatural`, `Boolean` e `String`. In *OhCircus*, we have the direct correspondence between $\mathbb{Z}$ and `Integer`, $\mathbb{N}$ and `UnlimitedNatural`, and $\mathbb{B}$ and `Boolean`. However, the `String` type does not have an equivalent one. So, we represent it by a sequence of elements in some standard character encoding system (such as ASCII or UNICODE) where the standard itself is naturally defined by traditional Z enumerations. Thus, operations like `concat`, `size`, and `substring` become readily available [8].

4.3 Generalisation

For mapping inheritance, one must include, in the *OhCircus* representation of the subclass, the **extends** clause followed by the name of the superclass. Moreover, an invariant in the class *Model* must assert the inclusion of the elements from the set of instances of the subclass into the set of instances of the superclass. In the following example, the invariant ensures that the elements of *creditAccounts* are also values of *accounts*.

state *ModelState*

$accounts : \mathbb{P}\ Account$

$creditAccounts : \mathbb{P}\ CreditAccount$

$creditAccounts \subseteq accounts$

4.4 Associations

Associations are an interesting construction to be captured, given that they are not readily available in purely object-oriented languages. The most common approach is to represent them directly using fields, though we believe that this is not the most natural way because of their conceptual distinction.

Once they introduce entities in the model, associations must be captured globally by the class *Model*. The roles played by the classes become fields, and an invariant

[8] Once data types do not have identity, they do not introduce a "set of all its instances" in the class *Model*.

links them all. Such fields are only syntactic sugar, because the consistency of the association is maintained by a relation in the class *Model*. However, they are necessary because the UML semantics allows statements about the classifiers to which a class is associated. Lastly, constraints are also established regarding multiplicity.

state *ModelState*

$$
\begin{array}{l}
persons : \mathbb{P}\, Person \\
accounts : \mathbb{P}\, Account \\
owns : Person \leftrightarrow Account \\
\hline
owns \in persons \leftrightarrow accounts \\
\forall\, a : accounts \bullet \#owns(\!|\, \{a\} \,|\!) \in \mathbb{N}_1 \\
\\
\forall\, p : persons \bullet p.accounts = accounts\,(\!|\, \{p\} \,|\!) \\
\forall\, a : accounts \bullet a.owner = accounts^{\sim}\,(\!|\, \{a\} \,|\!)
\end{array}
$$

Observe the relationship *owns* of Figure 3. This association becomes a field in the class *Model*, as can be seen. The invariant links the domains of the association with the respective set of instances and multiplicities. The *owner* and *accounts* roles become fields of, respectively, the classes *Account* and *Person*. Finally, the invariant of *Model* defines how the fields are interpreted by means of the original relationship [9].

5 Refinement

The refinement relation expresses a notion very common in Software Engineering: a "better" component can be used in the place of another, without modifying the properties of the system. Generally, refining means introducing details into a model, such as design decisions or looking for unexplored situations. It is worth noting that these improvements can be done gradually, producing models increasingly closer to a possible implementation.

In particular, the Formal Methods community has a standard definition for a refinement relation (although close-related to model based languages, such as Z): weakening pre-conditions to increase applicability of operations and strengthening post-conditions to decrease non-determinism. To guarantee the correctness of this procedure, there are proof obligations: the applicability and correctness theorems [30].

A direct consequence of the mapping we have proposed is the ability to explore UML refinement in formal ways: refinement of UML models can be assured by data refinement in *OhCircus*.

The example in this section copes with an old question from the object-oriented community: the representation of associations as fields [12,11,23]. In this work, we try to give some formal support to this approach, transforming models that

[9] The relational image of a set of objects ($(\!|\, \_ \,|\!)$) is the set of objects associated to the first through the relation.

mix fields and associations in ones that have only fields, bringing them closer to implementation. However, the support provided by a theorem prover is a very important feature to guarantee the correctness of a demonstration, and *OhCircus* still lacks it. In the other hand, *OhCircus* has the refinement theory of Z, which has the support from Z/EVES. Thus, we chose Z to specify our model.

5.1 Models

In this subsection, we will present the two class diagrams related to abstract and concrete models. In our mapping, each one of these class diagrams introduces a class *Model* in the specification. These classes will serve as the state of the specifications and their (meta-)operations will be refined.

The operations identified by the classes *ModelR* and *ModelA* change the sets of instances and associations of a diagram through the addition and removal of elements, in a similar approach to the one reported in [15]. However, since the representation of the sets does not change from one diagram to another, it is trivial to prove its refinement. So, we are interested only in the operations of addition and removal of an association pair. The steps of the proof can be seen in [1].

$$[A, B]$$

Initially, we want to establish that the structures of A and B are arbitrary. Abstracting the structure of the classes makes the formalisation more general.

5.1.1 Abstract model

The abstract model is very simple. It contains two classes and an association between them. To generalise as much as possible, no constraint will be imposed on the classes or association.

ModelR

$$iA : \mathbb{P}\, A$$
$$iB : \mathbb{P}\, B$$
$$R : A \leftrightarrow B$$

$$\operatorname{dom} R \subseteq iA \land \operatorname{ran} R \subseteq iB$$

This schema represents the class model of the abstract diagram, with some small changes in notation: iA and iB represent the sets of instances of A and B, respectively, while R represents the relation between them. Note the absence of the roles of the association; they will be introduced later.

The addition (*AddR*) and removal (*RemR*) operations represent the two possible ways of interaction of objects with the association: it is only possible to add or remove *links*. In the abstract model, this is represented using union and subtraction of pairs of a relation.

5.1.2 Concrete model

The concrete model changes the representation of the association; now, it is captured by means of fields and invariants.

ModelA

$$
\begin{array}{l}
iA : \mathbb{P}\, A \\
iB : \mathbb{P}\, B \\
as : B \pfun \mathbb{P}\, A \\
bs : A \pfun \mathbb{P}\, B \\
\hline
\mathrm{dom}\, as = iB \land \mathrm{ran}\, as \subseteq \mathbb{P}\, iA \\
\mathrm{dom}\, bs = iA \land \mathrm{ran}\, bs \subseteq \mathbb{P}\, iB \\
\forall\, a : iA;\ b : iB \bullet b \in bs\, a \Leftrightarrow a \in as\, b
\end{array}
$$

The concrete model introduces "fields" in classes A and B using the as and bs functions. This is the only possible representation, since Z does not allow mutually recursive schemas. But note the similarity between the notations $a.bs$ and $bs\, a$. Again, these "fields" must be related to the sets of instances of A and B. The last line of the invariant establishes the consistency of the association using fields: if the pair (a, b) is *linked*, then $a \in b.as$ *iff* $b \in a.bs$.

AddA

$$
\begin{array}{l}
\Delta ModelA \\
a? : A \\
b? : B \\
\hline
a? \in iA \land b? \in iB \\
a? \notin as\, b? \land b? \notin bs\, a? \\
\\
bs' = bs \oplus \{a? \mapsto (bs\, a? \cup \{b?\})\} \\
as' = as \oplus \{b? \mapsto (as\, b? \cup \{a?\})\} \\
iA' = iA \land iB' = iB
\end{array}
$$

The operation that adds a pair to the association (*AddA* and, analogously to removal, *RemA*) has been changed to support the new data representation. Only those instances to which a *link* is being added (or removed) will have their "fields" updated. Other unrelated elements are not modified.

5.2 Refinement proof

To demonstrate that the concrete model refines the abstract one, it is necessary to establish the retrieve which relates both representations of state and prove that it represents a relation of *simulation* [30]. This means to formulate and prove applicability and correctness theorems for all operations.

Retrieve

ModelR
ModelA

$$\forall\, a : iA \bullet bs\ a = R\ (\!|\ \{a\}\ |\!)$$
$$\forall\, b : iB \bullet as\ b = R^{\sim}\ (\!|\ \{b\}\ |\!)$$

Observe that our *Retrieve* is exactly what we proposed to the mapping of roles of an association as fields: the set of elements to which some instance is associated via a relation. This relation is also in conformity with that one established in the UML specification [20].

6 Conclusion

In this work we considered the UML formalisation using the formal specification language *OhCircus*. Although not exhaustive, we dealt with the most important UML static constructions. The originality of the approach, where we connect isolated elements of other works, and also our contribution to support the use of associations and their representation as fields are the main points of this paper.

The first contribution of this work is the transformation of UML class diagrams into *OhCircus* specifications. We chose to treat them in the same semantic level, giving a syntactic mapping to the UML elements direct representation in terms of *OhCircus* constructions. We believe that this alternative is more natural as long as *OhCircus* is an object-oriented language, thus not requiring the (re)definition of notions like classes and inheritance.

However, UML defines some elements which are not available in *OhCircus*. The proposal of a class *Model* arose as an interesting addition to this mapping. Bringing the "semantics" of an object-oriented model (i.e. their sets of instances, interactions, and constraints) to the specification itself revealed a valuable achievement: now we can naturally capture associations, global invariants, and even dynamic aspects.

The second contribution of this paper is the analysis of refinement in UML. In particular, the class *Model* allowed exploring the same refinement theory of Z, where there is only one (global) state and operations that act over it; this gave us a direct tool support. The case study was the previously discussed representation of associations as fields, and our contribution was that the representation of fields is a refinement of the one with associations.

It is worth noting that the use of Z/EVES was essential to the proof. The support of a consolidated tool is very important to give more credibility to the work. Unfortunately, it is not so friendly, requiring some practise to use it effectively.

6.1 Future works

Various works can be derived from this one; the obvious ones are directly related to the extension of this mapping, concerning UML aspects not yet explored. The

rest is related to the use of this mapping as the formal basis to analysing model transformations.

- The most immediate extension of this work regards the other static elements that were not captured, like abstract classes, interfaces, and some kinds of modifiers of associations. Following this direction, investigating the potential of using OCL (a language to express constraints) to annotate the UML models is also valuable.
- Investigate the possibility of "inverting" the mapping, examining how to transform a specification in the shape of those of this work (and even unconstrained ones) back into UML. This is important when some tool support for *OhCircus* will be available, formally analysing the model, but presenting the results in terms of UML constructions.
- The incorporation of UML dynamic aspects through the class *Model* is an appealing related work: analysing what sequences of instances of the diagram are valid and the provision of dynamic invariants are some of the issues which can be inspected.
- Concurrency in UML through *OhCircus* is particularly being exploited through the *real-time profile* [25], with much promising results.
- Related works to refinement are also important. The formal proof, using this mapping, of the validity of design patterns is an example of relevant contribution to the area of software engineering. Other refinements can be inspected, like the transformation of bidirectional associations into unidirectional ones and the inclusion or removal of a class from the model, proposing, for example, a set of transformation laws to UML models.

References

[1] Borges, R. M., *Integrando UML e Métodos Formais*, Final year project, Centro de Informática, Universidade Federal de Pernambuco (in Portuguese) (2004).
URL www.cin.ufpe.br/~rmb2/pdf/borges04integrando.pdf

[2] Bowen, J. and M. Hinchey, "Applications of Formal Methods," Prentice Hall PTR, 1995.

[3] Breu, R., U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe and V. Thurner, *Towards a Formalization of the Unified Modeling Language*, in: *ECOOP* (1997), pp. 344–366.

[4] Cavalcanti, A., A. Sampaio and J. Woodcock, *A Unified Language of Classes and Processes*, in: *St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering*, Satellite Workshop at FM'03, 2003.

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

[6] Evans, A., *Reasoning with UML Class Diagrams*, in: *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques* (1998), p. 102.

[7] Evans, A. and T. Clark, *Foundations of the Unified Modeling Language*, in: *Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop*, Ilkley, UK, 1997.

[8] Fischer, C., *How to Combine Z with Process Algebra*, in: *Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation* (1998), pp. 5–23.

[9] France, R. B., A. Evans and K. Lano, *The UML as a Formal Modeling Notation*, in: H. Kilov, B. Rumpe and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics* (1997), pp. 75–81.

[10] Goldsmith, M., "FDR: User Manual and Tutorial, version 2.77," Formal Systems (Europe) Ltd (2001).

[11] Graham, I., J. Bischof and B. Henderson-Sellers, *Associations Considered a Bad Thing*, Journal of Object Oriented Programming **9** (1997), pp. 41–48.

[12] Génova, G., *Semantics of navigability in UML associations*, Technical Report UC3M-TR-CS-2001-06, Computer Science Department, Carlos III University of Madrid (2001).

[13] Heimdahl, M., *Experiences and Lessons from the Analysis of TCAS II*, SIGSOFT Softw. Eng. Notes **21** (1996), pp. 79–83.

[14] Kim, S. and D. Carrington, *A Formal Mapping between UML Models and Object-Z Specifications*, Lecture Notes in Computer Science **1878** (2000), pp. 2–21.

[15] Lano, K. and J. Bicarregui, *UML Refinement and Abstraction Transformations*, ROOM 2 Workshop, Bradford University (1998).

[16] Morgan, C., "Programming from Specifications (2nd ed.)," Prentice Hall International (UK) Ltd., 1994.

[17] Moura, P., R. Borges and A. Mota, *Experimenting Formal Methods through UML* (2003), submitted to WMF'2003.

[18] OMG, *UML 2 Infrastructure Final Adopted Specification*, Whitepaper, Object Management Group (2003).
URL http://www.omg.org/cgi-bin/doc?ptc/2003-09-15

[19] OMG, *UML 2 OCL Final Adopted Specification*, Whitepaper, Object Management Group (2003).
URL http://www.omg.org/cgi-bin/doc?ptc/2003-10-14

[20] OMG, *UML 2 Superstructure Final Adopted Specification*, Whitepaper, Object Management Group (2003).
URL http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

[21] Roe, D., K. Broda and A. Russo, *Mapping UML Models incorporating OCL Constraints into Object-Z*, Technical Report 2003/9, Imperial College London (2003).

[22] Roscoe, A. W., C. A. R. Hoare and R. Bird, "The Theory and Practice of Concurrency," Prentice Hall PTR, 1997.

[23] Rumbaugh, J., *A Search for Values: Attributes and Associations*, Journal of Object Oriented Programming **9** (1996), pp. 6–8.

[24] Saaltink, M., *The Z/EVES 2.0 User's Guide*, Technical Report TR-99-5493-06a, ORA Canada, One Nicholas Street, Suite 1208 - Ottawa, Ontario K1N 7B7 - CANADA (1999).

[25] Sampaio, A., A. Mota and R. Ramos, *Class and Capsule Refinement for UML-RT*, in: *WMF 2003: 6th Workshop on Formal Methods, Brazil*, 2003, pp. 16–34, extended version to appear in Electronic Notes in Theoretical Computer Science, Elsevier, 2004.

[26] Selic, B. and J. Rumbaugh, *Using UML for Modeling Complex Real-Time Systems*, Whitepaper, Rational Software Corp. (1998).

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

[28] Sommerville, I., "Engenharia de Software (6a ed.)," Prentice-Hall, 2003.

[29] Spivey, M., "The Z Notation," Prentice-Hall, 1992.

[30] Woodcock, J. and J. Davies, "Using Z: Specification, Refinement, and Proof," Prentice Hall, 1996.

[31] Woodcock, J. C. P. and A. L. C. Cavalcanti, *The Semantics of Circus*, in: D. Bert, J. P. Bowen, M. C. Henson and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science **2272** (2002), pp. 184–203.