Formal Refactoring for UML Class Diagrams

Tiago Massoni, Rohit Gheyi, Paulo Borba

[1]Informatics Center – Federal University of Pernambuco (UFPE)
PO Box 7851 50.732-970 Recife – Brazil

{tlm,rg,phmb}@cin.ufpe.br

Abstract. *Refactoring UML models for evolution is usually carried out in an* ad hoc *way. These transformations can become an issue, since it is hard to ensure that the semantics of models is preserved. We provide a set of semantics-preserving transformations for UML class diagrams annotaded with OCL. Using the proposed transformations, software designers can safely define larger transformations and detect subtle problems when refactoring models. Semantics-preserving transformations can also be useful from design pattern introduction to MDA. We prove that our transformations are sound using a semantic model that is based on Alloy, which is a formal modeling language. Due to Alloy's ammenability to automatic analysis, our approach may additionally bring such analysis to class diagrams.*
Keywords: *model refactoring, class diagram, OCL, Alloy*

Resumo. *Refatorar modelos UML para evolução é normalmente uma atividade realizada de forma ad hoc. Estas transformações podem apresentar problemas, já que é difícil assegurar que a semática dos modelos é mantida. Desta forma, propomos um conjunto de transformações que preservam a semântica de diagramas de classe UML com OCL. Ao aplicar as transformações propostas, projetistas de software podem, de forma segura, definir transformações maiores, além de detectar problemas sutis ao refatorar modelos. Refatoração de modelos podem ser úteis tanto em MDA quanto para introduzir padrões de projeto. A corretude das transformações propostas é garantida através de um modelo semântico para UML baseado em Alloy, uma linguagem de modelagem formal. Como Alloy permite a realização de análises automáticas de modelos, nossa abordagem pode trazer benefícios similares para diagramas de classe.*
Palavras-chave: *refatoração de modelos, diagrama de classes, OCL, Alloy*

1. Introduction

The need to evolve is natural when thinking about software in the real world. However, the originally defined structure usually does not accommodate adaptations, demanding new ways for reorganizing software, in order to allow smoother and cheaper evolution. Modern development practices, such as refactoring, improve the design of programs while maintaining its observable behavior, mainly preparing software for evolution. Additionally, as in other engineering fields, modeling can be a useful activity for tackling significant problems early in software development. As an accepted standard, the UML [Object Management Group 2003] plays a significant role. Applying refactoring to UML models can help lowering the maintenance burden, since models can be restructured for a better understanding of the intended properties, laying the groundwork for further

changes. The introduction of a number of design patterns to a structural model can be accomplished by the application of model refactoring.

Commonly, no support is offered to verify whether applied refactorings preserve the semantics of UML models. In particular, class diagrams annotated with invariants using the Object Constraint Language (OCL) [Warmer et al. 2003]. Simple mistakes can lead to incorrect transformations that might, for example, introduce inconsistencies to a model, usually hard to detect in an *ad hoc* fashion (as showed in Section 2). In current practice, even using refactoring tools, programmers have to rely on compilation and a good test suite to ensure that the observable behavior is maintained [Fowler 1999]. In case of model refactorings, most rely on informal argumentation.

In this paper, we propose a set of small semantics-preserving transformations for UML class diagrams annotated with OCL, decribed in Section 3. In order to prove their soundness, we define a translational semantics for class diagrams and OCL. We use Alloy [Jackson et al. 2001] as the underlying semantic model (Section 4). Alloy is a formal object-oriented modeling language founded on relational logic. Alloy is suitable for domain modeling [Larman 2001], employing sets and relations as a simple semantic basis for objects and its relationships. In addition, Alloy models can be automatically analyzed, by means of the Alloy Analyzer tool [Jackson et al. 2000].

These transformations can formally derive model refactorings, which can be useful for introducing design patterns with the benefit of abstraction. Moreover, the increasing interest in the OMG's Model-Driven Architecture [Soley 2000] has drawn attention to model transformations, preferentially in an automatic fashion. We are especially interested in transformations regarding structural properties of domain models, which can be represented by UML class diagrams annotated with OCL. In this context, these transformations can be mostly useful for carrying out PIM-to-PIM [Soley 2000] semantics-preserving transformations. Other possible application lies on automatic synchronization between models and source code refactoring [Massoni et al. 2005], development of improved tool support for refactoring, or even comparing whether the specification of two software components are equivalent from based on syntatic conditions [Gheyi et al. 2004]. Also, translating class diagrams annotated with OCL to Alloy offers automatic simulation and analysis for UML [Massoni et al. 2004], given a translator tool and the Alloy Analyzer [Jackson et al. 2000]. Using Alloy as the semantic model supports an equivalence notion [Gheyi et al. 2005b] that is sufficiently flexible to compare class diagrams with distinct elements and structures.

2. Motivating Examples

In this section we show how apparently semantics-preserving transformations on models may lead to unexpected changes in the structural business rules. These examples provide useful insight on the problem of model refactoring, which should restructure models whereas maintaining the semantics stated by the original model. Figure 1 shows, in the context of a simple banking application, an example of a transformation that leads to class diagrams with different semantics if compared to the left-hand side (LHS) diagram. This diagram states that customers may own more than one account, besides constraining accounts to be handled by at most one bank card. Suppose a designer tries to restructure this diagram by adding a straightforward association from `Customer` to `BankCard`, repre-

senting an alternative path to relate customers and their cards, as reflected in the right-hand side (RHS) diagram. The association relates exactly one card to each customer.

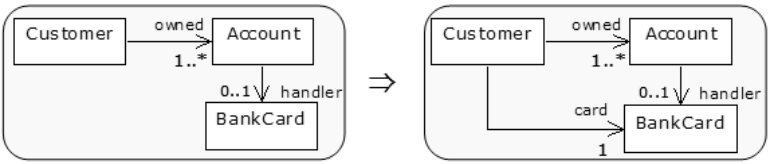


Figure 1. Introducing an association.

A deeper analysis of the possible snapshots (objects and links) of the LHS diagram shows that `Customer` objects may exist with no cards. One possible snapshot is shown in Figure 2, by means of a UML object diagram [Object Management Group 2003]. In the refactored diagram, this scenario is ruled out, since every customer owns exactly one card, demanding instances of `BankCard` whenever a customer exists.

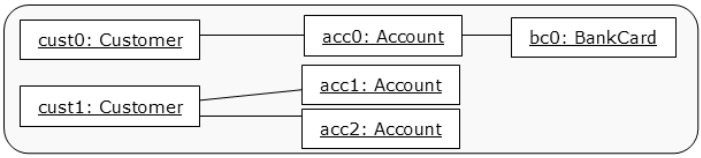


Figure 2. A possible instance for the original diagram.

Additionally, Figure 3 depicts a similar transformation. On the LHS diagram, accounts may be checking or savings, and the `primary` association represents a relationship between accounts. This association is intended to model a business rule defining that savings account is the primary type of account, and checking accounts can only be opened given there is an existing savings account for that customer. As such, the diagram is restructured by pushing down the association to the subclasses, stating that checking accounts depend on a savings account to exist. However, analyzing the LHS diagram, we see that it possibly yields snapshots that associate two `SavAcc` objects, same for the `ChAcc` class. These snapshots are not modeled in the refactored diagram, over-constraining the solution space. This transformation does not preserve semantics considering class diagrams without OCL constraints.

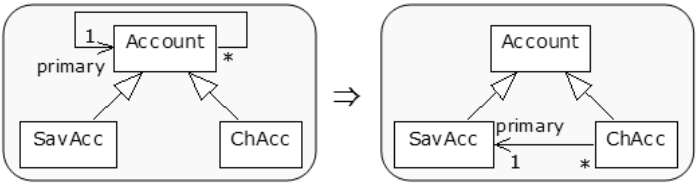


Figure 3. Pushing down an association.

These examples suggest that designers need a guide to define and apply refactorings to class diagrams, backed by a formal semantics, to ensure that improvements do not change the semantics of a model. We believe that a catalog of transformations that are known to be semantics-preserving can lead designers to accomplish their goals in restructuring models.

3. Semantics-Preserving Transformations for Class Diagrams

In this section, we present a set of transformations for UML class diagrams. These transformations manipulate classes, associations, generalizations and OCL constraints, based on UML 1.5 [Object Management Group 2003]. As the transformations must involve equivalent diagrams, we first address an equivalence notion for domain models. With a comprehensive set of simple transformations, we aim to provide powerful guidance on the derivation of more complex transformations.

3.1. An Equivalence Notion

The common equivalence notion states that two class diagrams are equivalent if they have the same semantics. This notion is useful, but not flexible enough to compare equivalent models with auxiliary elements such as `Vector` in Figure 4(b), or with different forms of representing the same concept, such as `accs` in Figure 4(a). These models are intuitively equivalent, taking into consideration the relationship between banks and accounts, which is maintained whether there is an intermediate collection or not. However, using this common notion, they are not equivalent since they have different elements.

Figure 4. Comparing UML Class Diagrams.

In order to compare models in such scenario, we propose a flexible equivalence notion. Our approach compares the semantics of two class diagrams only for a number of relevant model elements (class, attribute and association[1]). The set of relevant element names is called alphabet (Σ). The names that are not in the alphabet are considered auxiliary, or not relevant for the comparison. For instance, suppose that Σ contains only the `Bank` and `Account` names in the previous example. If both diagrams have the same interpretations for those names in all valid snapshots, they are considered to be equivalent under this equivalence notion. Other names, such as `col`, `Vector` and `elems`, are regarded as auxiliary.

However, sometimes we might have model elements that, although relevant, cannot be compared, since they are not part of both diagrams. For instance, suppose that we include `accs` to Σ. In this case, we cannot compare the models in Figure 4, since `accs` is not part of the model in Figure 4(b). Some structures may have been replaced by other elements during refactoring activities, even though the resulting model maintains the original semantics and expresses the same invariants. For instance, in Figure 4(b), `accs`

[1]Names are given only to navigable ends of an association

is not part of the model, but can actually be expressed as the composition of `col` and `elems`. In those cases, our equivalence notion can consider a mapping, called view (v), establishing how an element of one model can be interpreted using elements of another model. Views consist of a set of items such as $n \rightarrow exp$, where n is an element's name and exp is an expression, specifying how the concept n can be expressed in terms of other concepts. Notice that although the values of auxiliary names are not compared, they can be used to yield an alternative meaning to relevant names. In the previous example, we may choose a view containing the following item: $accs \rightarrow col.elems$`->`$asSet()$[2]. Now we can infer that both models are equivalent. Notice that `accs` is defined in terms of two names that belong to Figure 4(b). More on the formalization of this equivalence notion in the Prototype Verification System (PVS) [Owre et al. 2005] – which contains a formal specification language and a theorem prover – can be found elsewhere [Gheyi et al. 2005b].

3.2. Laws

Next, we describe a number of primitive laws for class diagrams based on the equivalence notion described before. Each primitive law, when applied in any direction, defines one transformation that preserves semantics. They define templates that must be matched by class diagrams in order to be applied, imposing syntactic conditions for the transformations that guarantee preservation of semantics.

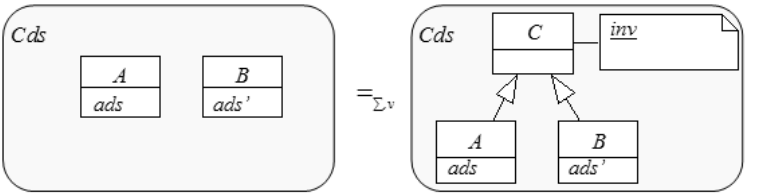
Regarding classes, Law 1 states that we can introduce a generalization between two classes that have no superclasses, provided the name of the new superclass is not previously used in the model. As a well-formedness rule, a UML class diagram cannot contain two classes with the same name (for simplicity, we do not consider packages). We can also remove a generalization between them (from right to left) if the superclass is not used elsewhere. *Cds* denotes all classes, attributes, associations, invariants and generalizations that are part of the model but not depicted in the law representation (we take a closed-world assumption, considering that all model elements are included into *Cds*). We write ($\rightarrow$), before the condition, to indicate that this condition is required when applying this law from left to right. Similarly, we use ($\leftarrow$) to indicate that it is required when applying the law in the opposite direction, and we use ($\leftrightarrow$) to indicate that the condition is necessary in both directions.

We use *ads* to denote the set of attribute declarations and associations of a class (associations whose opposite end is navigable from the class). The boolean operation `oclIsKindOf` tests whether the target object is subtype of the type indicated by the parameter. The *inv* invariant indicate that `A` and `B` define a partition of `C` (in practice, `C` is an abstract class or interface). Law 1 also deals with the introduction or removal of a generalization for any number of classes, using similar OCL invariants. `C->forAll` is short for `C.allInstances->forAll` (analogous simplified expressions are used throughout the text). The ($\leftarrow$) condition guarantees that there is no attribute, association or invariant including *C*, except *inv*, as *C* must be taken as auxiliary. Similarly, we have defined laws for introducing an empty class or a subclass.

An additional condition for all names in Σ that are not on the opposite diagram, v must contain exactly one valid item for it. We consider this condition for laws that

[2]the $asSet$ operation is used for converting the resulting expression to set type, since composition of associations in OCL denotes a bag type

Law 1 ⟨*introduce generalization*⟩



where

inv: `self.oclIsKindOf(A) or self.oclIsKindOf(B)`

provided

(↔) if *C* belongs to Σ, then *v* contains the *C*→*A*`->`*union(B)* item;

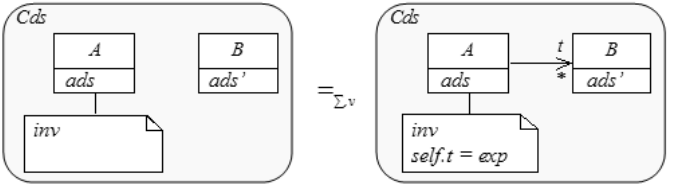
(→) *Cds* does not declare any class named *C*;

(←) *C* does not appear in *Cds*, *ads* or *ads'*.

introduce or remove elements, such as Laws 1 and 2. In order to improve readability, we do not express them in those laws. This law is also valid for introducing an interface.

Next, we present laws for associations. Law 2 states that we can introduce a new association (or attribute) with opposite end named *t*, along with its definition as an OCL invariant in the form *seft.t* = *exp*. The *exp* expression can be *self.t* itself or an expression not containing *t*, defined in terms of existing associations. We can also remove an association that is not being used.

Law 2 ⟨*introduce association with definition*⟩



provided

(↔) if *t* belongs to Σ then (*v* contains the *t*→*exp* item and *t* does not appear in *exp*);

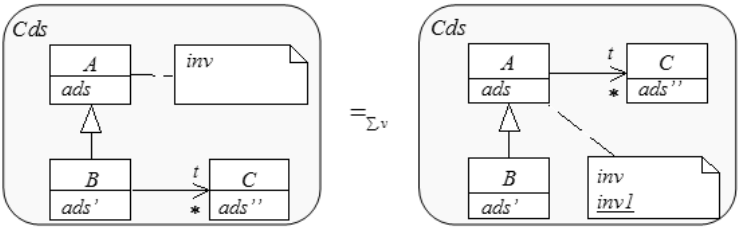
(→) (1) *A* and its family does not declare any association or attribute named *t*; (2) *t* does not appear in *exp*, or *exp*=*self.t*; (3) *exp* ≤ *t* in *Cds* and *inv*.

(←) *t* does not appear in *Cds* or *inv*.

The *exp* ≤ *t* expression denotes that *exp* is a subtype of *t*'s type. Constraints involving Σ and *v* must be carefully introduced. When introducing or removing an association whose name belongs to Σ, we must guarantee that the *t*→*exp* item belongs to the view. Also, the target class *B* may be *A* itself. The family of a class is the set of its super and subclasses in *Cds*, whether direct or indirect. There is one implicit condition applying this law from left to right is that the new invariant *seft.t* = *exp* must be well typed. From this law, we can derive another law which introduces an association without a definition. In this case, *exp* is made *self.t* itself, which results in a tautology, and *t* is not in Σ.

We establish Law 3 for moving an association within a family of classes. We can pull up an association from a class to its superclass by adding an invariant stating that this association only relates objects of the subclass. Similarly, we can push down an association if the source model includes an equivalent invariant. In this case, we have to make sure that decreasing *t*'s type does not introduce type errors. A similar law can be proposed for pulling up associations in the opposite direction.

Law 3 ⟨*pull up association to superclass*⟩



where

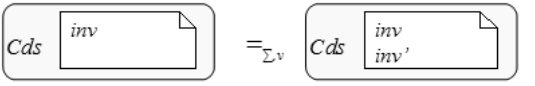
inv1: `(not self.oclIsKindOf(B)) implies self.t->isEmpty()`

provided

(←) there is no expression containing *t*, where $t \not\leq B \rightarrow C$ and $t \leq A \rightarrow C$, in *Cds* or *inv*.

$B \rightarrow C$ express a binary association type. We also define laws for manipulating OCL invariants. Law 4 establishes that we can add or remove an OCL invariant as long as it can be logically deduced from other invariants in the same model. When introducing an invariant, *inv'* must be well typed.

Law 4 ⟨*introduce OCL invariant*⟩



provided

(↔) *inv'* can be deduced from the invariants in *Cds* and *inv*.

We have proposed other laws dealing with associations, OCL invariants and syntactic sugar constructs, such as a law for converting composition to associations with multiplicity constraints. Also, there are laws for introducing multiplicity constraints to associations and attributes. Notice that our laws, such as Laws 2 and 3, are enunciated with unconstrained multiplicities. However, it is important to mention that we can have or deduce invariants stating multiplicities. We proved the soundness of these laws by giving a translational semantics for class diagrams using Alloy. We prefer to propose fine-grained transformations, because they are easier to be proven semantics preserving.

3.3. Applications

Although our laws define fine-grained transformations, in this section we show how to derive a number of coarse-grained transformations, such as refactorings, by law composition, which consequently preserve semantics. The LHS diagram of Figure 5 shows part

of a banking system in which each account (savings and checking) directly relates to a customer, by a specific ownership association. However, software designers may want, for instance, to apply a refactoring similar to Extract Interface [Fowler 1999] in this diagram, resulting in the RHS diagram. The software designer can use our laws to formally and safely transform it, as we will describe next.

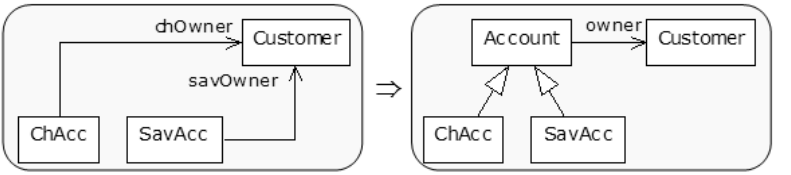


Figure 5. Extract Interfact Refactoring.

Consider that Σ={`ChAcc,SavAcc,owner,Customer`} and v={`owner`→`chOwner->union(savOwner)`}. It is important to mention that we may not choose the right alphabet and view from the beginning in a sequence of refactorings. However, we propose some theorems that allow us to increase or decrease them only by checking syntactic conditions in this sequence, as described elsewhere [Gheyi et al. 2005b].

We first apply Law 1 from left to right to introduce the `Account` abstract class. We can apply this law since `Account` is a new name. As this name does not belong to Σ, we do not need any item in v. Next, we are able to introduce the `owner` association in `Account`, along with its definition (`owner=chOwner->union(savOwner)`), by applying Law 2 from left to right. We can apply this law since there is no association or attribute named `owner` in the `Account`'s family. Moreover, since `owner` belongs to Σ, v has an item similar to its definition. Notice that the type correctness conditions are also satisfied.

Since `Account` is an abstract class, `owner` just relates objects from checking and savings accounts to customers. Applying some predicate calculus and relational operator laws, besides the definition of `owner` previously introduced, we can add, by applying Law 4 from left to right, definitions for the `chOwner` and `savOwner` associations. We can deduce that `chOwner` is `owner` restricting its type from `ChAcc` to `Customer`. We are now able to replace every occurrence of `chOwner` and `savOwner` in all invariants in the diagram, except in their definition, by applying the same law.

Finally, since `chOwner` and `savOwner` do not appear in the diagram except in their definition, and they are not in Σ, we can remove them and their definition using Law 2 from right to left. As described before, notice that in some steps, when using Laws 1 and 2, we have to make sure that, when introducing or removing elements, the view must remain valid.

Our laws can be useful not only to refactor models stepwisely, but also to derive general coarse-grained transformations. For instance, the previous process can be generalized, stating the *Extract Interface* refactoring similarly to the laws. Since this refactoring is derived using primitive laws, it also preserves semantics. In addition, our

laws can be useful for indicating when a transformation may not be semantics preserving. Law 3, for example, shows more evidence on the transformation depicted in Figure 3. An UML diagram without OCL invariants cannot express the invariant demanded by the law, explaining why the diagrams present conflicting snapshots.

4. Translational Semantics for Class Diagrams

In this section, we propose a translational semantics for domain models as class diagrams, in terms of Alloy, a formal modeling language. The essence of a translational approach to semantics is to translate constructs in a language under study into another that has a well-defined semantics [Greenfield et al. 2004]. Translation can be a effective way to provide semantics in a number of contexts, such as compilation or code generation. In this work, the translation of class diagrams to Alloy models can be used to define transformations by reasoning about UML model elements on Alloy's semantics. After providing a brief overview of Alloy, we present how the translation to Alloy was designed and employed to prove the soundness of our laws.

4.1. Alloy

Alloy [Jackson et al. 2001] is a formal modeling language based on first-order logic, allowing specification of – primarily structural – invariants in a declarative fashion. In general, models in Alloy are described at a high level of abstraction, ignoring implementation details. We base our translation rules on Alloy 3 [Jackson 2005].

The language assumes a universe of objects partitioned into subsets, each of which associated with a basic type. An Alloy model is a sequence of paragraphs of two kinds: signatures, used for defining new types; and formula paragraphs, such as facts and predicates, used to record invariants. Analogous to classes, each signature denotes a set of objects. These objects can be mapped by the relations (associations) declared in the signatures. A signature paragraph may introduce a collection of relations.

As an example, we show an Alloy model for part of the banking system, where each bank contains a set of accounts and a set of customers. An account can only be a savings account. The next Alloy fragment declares four signatures representing system entities, along with their relations and invariants:

```
sig Bank {
  accs: set Account,
  custs: set Customer
}
sig Customer, Account {}
sig SavAcc extends Account {}
```

```
fact BankInvariants {
  Account = SavAcc
  all a:Account | #a.~accs=1
}
```

In the declaration of `Bank`, the `set` keyword specifies that the `accs` relation maps each object in `Bank` to a set of objects in `Account`, exactly as an unconstrained association in UML. `SavAcc` denotes one kind of account. In Alloy, one signature can extend another one by establishing that the extended signature is a subset of the parent signature. For example, the set of `SavAcc` objects is a subset of the `Account` objects.

A fact is a formula paragraph, used to package invariants. Differently from OCL invariants, a fact does not introduce a context for its formulae, allowing global invariants

on models (such as set cardinality). Facts' formulae are declared as a conjunction, establishing general invariants about the declared signatures and relations. The first formula of the `BankInvariants` fact states that every account is a savings account; the second one states that every account is related to one bank by `accs`. The `all` keyword is the universal quantifier. The # symbol is the cardinality set operator. The expression `~accs` denotes the transpose of `accs`. The join of relations[3] `a` and `~accs` is the relation yielded by taking every combination of `a` and `~accs` elements, joining the tuples with common values. The type of `a` must match the domain type of `~accs` (both `Account`). The formula guarantees that each account relates to exactly one bank. The semantics of an Alloy model is the set of all assignments of objects and links to signature and relation names that satisfy all constraints [Jackson 2005].

4.2. Semantics

We provide a semantics for UML class diagrams by translation to correspondent Alloy models. Alloy constructs can semantically represent a number of UML constructs, contributing with a semantics to a subset of UML that may be automatically analyzed. Furthermore, translating a representative subset of OCL expressions into Alloy is relatively straightforward, since both languages are used for expressing invariants in domain models, based on first-order logic, exhibiting similar expressiveness [Edwards et al. 2004]. OCL, however, presents syntax and type system closer to programming languages, which usually restrains simplicity. Due to this property, we limit our translational semantics to a core subset of the language, based on the most important logical and set operations, which is sufficient to express other constructs in invariants for domain models.

In order to define translation rules between UML and Alloy, we do not consider a number of UML constructs (we focus on domain models), such as operations (methods) and their effects, besides timing constraints (e.g. {`frozen`}). Moreover, Alloy has also been used for modeling properties over state transitions [Dennis et al. 2004], which shows the language's usefulness in behavioral modeling as well. We believe that related UML constructs can be similarly analyzed by means of analogous translation. In order to simplify the translation, binary associations must include role names for each navigable end. Regarding OCL, we only consider class invariants, due to the reasons previously explained. Since recursive operations have an undefined semantics in OCL, we do not deal with those in our translation. Also, numeric types (except integer) are ignored, as our focus lies on domain models. The keyword `self` is mandatory when expressing context-dependent invariants.

Our translation rules are divided into two categories: from UML diagrammatic constructs to Alloy constructs and from OCL invariants with core constructs to logically-equivalent Alloy formulae. Regarding OCL constraints, we based our translation on the OCL specification version 1.5 [Object Management Group 2003]. Basic set theory and relational calculus guided the translation rules, neglecting OCL constraints that may present undefined semantics (such as recursion). Figure 6 depicts a class diagram describing an extended version of the banking system. The invariant over `Customer` states that a customer identifier must be unique, while the invariant over `Account` states that it is an abstract class.

[3] In Alloy, set elements are designed as singleton unary relations.

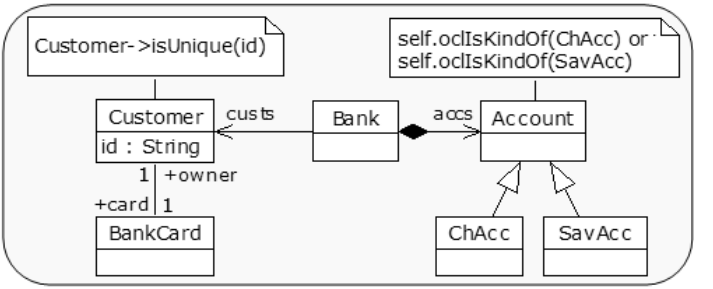


Figure 6. Extended Class Diagram for the Banking System.

Regarding diagrammatic constructs, each class and interface is translated to a signature in Alloy. Regarding interface, a formula is introduced for ensuring that the signature has no direct instances. In addition, each binary association is translated to two relations declared with the `set` qualifier. Similarly, attributes also translate to relations. A relation is created for each navigable association end, using the opposite role name. This rule is required due to the limitation in representing navigability constraints with binary relations in Alloy (a binary relation is bidirectional by definition). In our example, `Customer` and `BankCard` can be represented in Alloy as follows:

```
sig Customer {                    sig BankCard {
  card: set BankCard,               owner: set Customer
  id: set String                  }
}
```

In case we have two navigable ends for an association, each signature declares a relation for its opposite navigable end, as exemplified by `owner` and `card`. A constraint is added, stating that a relation is the transpose of its opposite counterpart.

```
fact BankInvariants {
  card = ~owner
  all c:Customer | #c.card=1
  all a:Account | #a.~accs=1 ...
}
```

For example, the second formula in `BankInvariants` states that there is exactly one object of `BankCard` mapped by each customer. In addition, generalization is translated to Alloy's `extends`. The `SavAcc` class is translated to a similar signature declared in Section 4.1.

OCL invariants are translated into equivalent Alloy formulae, being universally quantified on `self`. This limits expression of invariants in OCL (such as number of instances of a class), since universally quantified formulae can be true either if the formula is valid or the quantified set is empty. Even though Alloy allows global invariants without quantification, we do not intend to fix the problem, as we provide a semantics for OCL.

Next, we show an Alloy fragment that translates the invariants from the `Account` and `Customer` contexts. Set membership results from translating `oclIsKindOf`,

whereas `isUnique` is translated to an equivalent quantified formula. The translation can create a separate fact for each OCL context, although not shown here for simplicity.

```
all self:Account | (self in ChAcc) or (self in SavAcc)
all self:Customer | all disj c,c':Customer | c.id != c'.id
```

The `in` operator denotes set membership, while `!` denotes negation. The `disj` keyword states that the declared variables are distinct. Table 1 formulates the translation rules for a core of OCL formulae and expressions, where `X,Y` denote collections, `P,Q` denote formulae, `a,b` denote variables and `r` an attribute.

Table 1. Translation rules from a core OCL to Alloy.

OCL	Alloy
`X->forAll(a|P)`	`all a:X | P`
`P and Q`	`P && Q`
`X.allInstances`	`X`
`a equals b`	`a = b`
`not(P)`	`!(P)`
`X.oclIsKindOf(Y)`	`X in Y`
`X.isUnique(r)`	`all disj a,a':X | a.r != a'.r`
`X->isEmpty()`	`no X`
`X.size()`	`#X`

Currently, the translation is performed systematically, but manually. Nevertheless, the process was designed to be decidable, allowing automatization. Both languages can be defined by meta-modeling, and transformations can be implemented as a correspondence between meta-model elements, using an approach similar to OMG's Model-Driven Architecture [Soley 2000]. XML-based representations, such as XMI [Object Management Group 2001], can certainly help obtaining Alloy paragraphs from UML classes and OCL invariants. Furthermore, OCL constraints can be translated into Alloy formulae as source-to-source transformations, involving a parser for OCL and manipulation of abstract syntax trees.

4.3. Soundness

We have proposed a comprehensive set of laws for Alloy, which were specified and proved in PVS based on an Alloy's semantics and equivalence notion [Gheyi et al. 2004]. Our approach to prove class diagram laws is to translate into Alloy and use our Alloy laws to reason about UML laws. In fact, the UML laws presented here can be reduced to equivalent Alloy laws. Next we present a formal argumentation stating that our laws are sound.

As described in Section 3.1, we compare two class diagrams with respect to a set of important names. For example, Law 1 present diagrams with the same model elements, except for class *C* and an invariant stating that it is an abstract class. Notice that classes *A* and *B* are disjoint in the LHS diagram. The same constraint is preserved in the RHS diagram, since subclasses from the same parent class are disjoint. Moreover, on the RHS diagram, objects of classes *A* and *B* are instances of class *C*, and *C* is an abstract

class. These constraints are preserved in the LHS diagram, if C is an important name (considering C in Σ). In this case, this law has a condition establishing that v has the *C→A->union(B)* item; hence satisfying both constraints. Therefore, both models have the same semantics.

Furthermore, both models in Law 2 have the same elements and invariants, except for the *t* association and an invariant with its definition. If *t* is in Σ, there is a condition stating that v has an item for *t* that is equivalent to the invariant introduced on RHS diagram. Therefore, both models have the same meaning. In Law 3 both diagrams have the same structures, except for the *t* association and an invariant. On the RHS diagram, *t* relates objects from class *A*, which is the parent class of *B*, to objects in class *C*. On the LHS diagram, *t* relates objects from class *B* to class *C*. However, there is an explicit invariant on the RHS diagram stating this same constraint. Therefore, both diagrams have the same semantics. Finally, Law 4 preserves semantics since it introduces or removes an invariant that is deduced from the model.

5. Related Work

Recent approaches for defining UML model transformations have been proposed. Evans et al. [Evans 1998], for example, define transformations for manipulating class diagrams, which are sound according to a formal semantics. Some of these transformations weaken constraints, resulting in more abstract models. In a similar approach, a semantics for UML class and state diagrams — in terms of extended first-order set theory — allows definition of abstraction and refinement transformations [Lano and Bicarregui 1998]. These approaches do not state in which conditions the transformations can be applied, also not considering OCL invariants. This can be harmful, since simple transformations can lead to inconsistencies or type errors. Another work [Gogolla and Richters 1998] introduces a number of semantics-preserving transformations for class diagrams. These transformations express complex UML constructs in terms of primitive ones and OCL invariants, removing syntactic sugar, which is similar to a subset of our laws. However, none of those formalize an equivalence notion, which is an important contribution.

Laws for top-level design elements of UML-RT (Real Time) have been proposed by recent work [Sampaio et al. 2003]. They propose laws for both structural and behavioral constructs (capsules), not intended to be primitive, as ours. They assume that relationships are directed and constraints involve only relationships as attributes, not considering global invariants. In contrast, our laws consider global constraints, focusing on models at high level of abstraction.

Model refactoring has been exploited as an application of transformations for UML. Sunyé et al. [Sunyé et al. 2001] presents primitive refactorings for class and state diagrams, grounded on OCL constraints at the metamodel level. However, these transformations do not consider OCL invariants. Furthermore, the equivalence notion based on metamodel elements does not compare models with different elements, as our notion does. More recently, a work discusses implementation of model refactorings as rule-based transformations [Porres 2003], not regarding preservation of semantics.

There have been a number of efforts on proposing formal semantics for UML and related modeling languages, in order to clarify the semantics of its diagrammatic constructs, supporting tool development. For example, related approaches [Evans et al. 1999]

give a formal semantics to a subset of UML class diagrams. We defined a translational semantics for leveraging to UML class diagrams transformations proposed for Alloy [Gheyi et al. 2005a].

Bordeau and Cheng [Bourdeau and Cheng 1995] define a similar translational approach for a related modeling notation. They automatically map models to algebraic specifications, allowing formal reasoning on the semantics of the translated specification. In contrast, Alloy admits a more direct translation from UML, since both are similarly suitable to domain modeling [Dennis et al. 2004]. Also, automatic simulation and analysis in Alloy may be more appealing to software architects and designers.

Our translational semantics allows generation of instances and counterexamples to claims over class diagrams, by means of the Alloy Analyzer tool [Jackson et al. 2000]. Recent work in this subject shows how the analysis can be leveraged to a comprehensive subset of class diagrams by using our translational semantics [Massoni et al. 2004].

6. Conclusion

In this paper, we have proposed semantics-preserving transformations for UML class diagrams annotated with OCL that are proven sound according to a translational semantics in Alloy, helping avoid incorrect transformations and detect subtle problems. The novelty of our work is twofold: defining semantics-preserving transformations for UML, and proposing a comprehensive set of small transformations that can be composed to derive more complex transformations, such as structural model refactorings.

The given translational semantics has also offered an abstract equivalence notion for class diagrams. In this notion, we are able to compare whether two class diagrams are equivalent, even if these diagrams present distinct classes and associations, or structures. Moreover, our translational semantics makes it possible to leverage to class diagrams automatic simulation and analysis from the Alloy Analyzer. The tool yields automatic instances for a translated diagram, regardless of input test cases, as well as analyzes specific properties on a diagram. The latter can be useful for identifying otherwise subtle changes in semantics [Massoni et al. 2004].

Our transformations are very simple to apply because they require simple syntactic conditions, allowing straightforward implementation for transformations. In the context of MDA, derived refactorings can be used to enhance PIMs aiming at smoother impact of requirement changes. In an on-going work, we formalize the relationship between model and program transformations, defining correspondent refactorings [Massoni et al. 2005]. Therefore, transformations can be applied to PSMs and source code, based on PIM model refactorings derived from the primitive laws.

Our translational semantics is limited in the sense that Alloy cannot represent implementation-oriented class diagram constructs. For instance, attributes are mapped to simple binary relations, disregarding properties such as visibility and default value. Further, the translation is systematic, yet currently manual. However, regarded as future work, tool support will carry out source-to-source translation between metamodels of both languages, and OCL to Alloy invariants. Moreover, we intend to propose more transformations. A comprehensive set of transformations for Alloy has been proposed [Gheyi et al. 2004]. These transformations were specified and proved in PVS, increasing confidence on the results, based on an Alloy's semantics and equivalence notion

codified into PVS [Gheyi et al. 2005a]. As future work, we intend to follow a similar approach for proving transformations for class diagrams.

Acknowledgments

We would like to thank all anonymous referees, whose appropriate comments helped improving the paper, and members of the Software Productivity Group. This work was partially funded by CAPES and CNPq.

References

[Bourdeau and Cheng 1995] Bourdeau, R. and Cheng, B. (1995). A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Soft. Engineering*, 21(10):799–821.

[Dennis et al. 2004] Dennis, G. et al. (2004). Automating Commutativity Analysis at the Design Level. In *Inter. Symposium on Software Testing and Analysis*, pages 165–174.

[Edwards et al. 2004] Edwards, J. et al. (2004). A Type System for Object Models. In *12th Foundations of Software Engineering*, pages 189–199. ACM Press.

[Evans et al. 1999] Evans, A. et al. (1999). The UML as a formal modeling notation. In *UML'98 - First International Workshop*, pages 336–348. Springer.

[Evans 1998] Evans, A. S. (1998). Reasoning with UML Class Diagrams. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 102–113. IEEE CS Press.

[Fowler 1999] Fowler, M. (1999). *Refactoring—Improving the Design of Existing Code*. Addison Wesley.

[Gheyi et al. 2004] Gheyi, R., Massoni, T., and Borba, P. (2004). Basic laws of object modeling. In *Third Specification and Verification of Component-Based Systems (SAVCBS), affiliated with ACM SIGSOFT 2004/FSE-12*, pages 18–25, Newport Beach, United States.

[Gheyi et al. 2005a] Gheyi, R., Massoni, T., and Borba, P. (2005a). A Rigorous Approach for Proving Model Refactorings. Submitted for publication.

[Gheyi et al. 2005b] Gheyi, R., Massoni, T., and Borba, P. (2005b). An Abstract Equivalence Notion for Object Models. In A. Mota, A. M., editor, *Electronic Notes in Theoretical Computer Science*, volume 130, pages 3–21. Elsevier.

[Gogolla and Richters 1998] Gogolla, M. and Richters, M. (1998). Equivalence Rules for UML Class Diagrams. In *UML'98 - First International Workshop*, pages 87–96.

[Greenfield et al. 2004] Greenfield, J. et al. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley.

[Jackson 2005] Jackson, D. (2005). Alloy 3.0 Reference Manual. http://alloy.mit.edu/beta/-reference-manual.pdf.

[Jackson et al. 2000] Jackson, D. et al. (2000). Alcoa: the Alloy Constraint Analyzer. In *22nd International Conference on Software Engineering*, pages 730–733. ACM Press.

[Jackson et al. 2001] Jackson, D. et al. (2001). A Micromodularity Mechanism. In *9th Foundations of Software Engineering*, pages 62–73. ACM Press.

[Lano and Bicarregui 1998] Lano, K. and Bicarregui, J. (1998). Semantics and Transformations for UML Models. In *UML'98 - First International Workshop*, pages 107–119.

[Larman 2001] Larman, C. (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall.

[Massoni et al. 2004] Massoni, T., Gheyi, R., and Borba, P. (2004). A UML Class Diagram Analyzer. In *3rd International Workshop on Critical Systems Development with UML, affiliated with 7th UML Conference*, pages 143–153.

[Massoni et al. 2005] Massoni, T., Gheyi, R., and Borba, P. (2005). A Model-driven Approach to Program Refactoring. Submitted for publication.

[Object Management Group 2001] Object Management Group (2001). XMI specification.

[Object Management Group 2003] Object Management Group (2003). Unified Modeling Language Specification Version 1.5.

[Owre et al. 2005] Owre, S. et al. (2005). PVS language and prover reference. At http://pvs.csl.sri.com.

[Porres 2003] Porres, I. (2003). Model refactorings as rule-based update transformations. In *6th UML Conference*, volume 2863 of *LNCS*, pages 159–174. Springer.

[Sampaio et al. 2003] Sampaio, A. et al. (2003). Class and Capsule Refinement in UML for Real Time. In *6th Brazilian Workshop on Formal Methods*, pages 16–34.

[Soley 2000] Soley, R. (2000). Model Driven Architecture. OMG Document 2000-11-05.

[Sunyé et al. 2001] Sunyé, G. et al. (2001). Refactoring UML Models. In *4th UML Conference*, volume 2185 of *LNCS*, pages 134–148. Springer.

[Warmer et al. 2003] Warmer, J. et al. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2nd edition.