An Abstract Equivalence Notion for Object Models

Rohit Gheyi , Tiago Massoni , Paulo Borba

[1]Informatics Center — Federal University of Pernambuco
PO Box 7851 – 50.732-970 Recife, PE

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

Abstract. *Equivalence notions for object models are usually too concrete in the sense that they assume that the compared models are formed by elements with the same names. This is not adequate in several situations: during model refactoring, when using auxiliary model elements, and when the compared models comprise distinct but corresponding elements. So, in this paper, we propose a more abstract and language-independent equivalence notion for object models. It supports, as desired, abstraction from names and elements when comparing models. We use the PVS system to specify and prove properties of our notion. It is illustrated here by comparing simple models in Alloy, a formal object-oriented modeling language, but has also been applied for deriving a comprehensive set of algebraic laws for Alloy.*

1. Introduction

Comparing deliverables during the software development process is quite important. In fact, there are several approaches for comparing the behavior of programs. For instance, these are useful during maintenance, when we may wish to replace a given component by a behaviorally equivalent, better structured, one. It is similarly useful to compare design models, which can be expressed by object modeling languages such as UML [Booch et al., 1999] or Alloy [Jackson, 2004].

However, current equivalence notions for object models are usually too concrete. They assume that the compared models are formed by elements with the same names. This is not adequate in several situations. Model refactoring, for example, which change the structure of models, yet maintaining the properties previously stated. Nevertheless, it is difficult to verify whether the resulting model preserves the semantics, especially when model elements are replaced by alternative structures. Furthermore, auxiliary model elements may be used, which should be ignored when calculating equivalences. Also, when the compared models comprise distinct but corresponding elements, we can easily find models that are intuitively equivalent but cannot be proved so based on most equivalence notions.

In this paper, we propose a more abstract equivalence notion for object models. It supports, as desired, abstraction from names and elements when comparing models. It is flexible enough for comparing only parts of object models, by relying on a chosen set of relevant elements— the alphabet— and a mapping from relevant elements to their equivalent counterparts in the corresponding model— the view. Furthermore, this equivalence notion is applicable to any object modeling language. We encoded it in the Prototype Verification System (PVS) [Owre et al., 2001a] specification language and formally derived several properties by using the PVS prover [Owre et al., 2001b]. These proofs show the independence of the notion with respect to the underlying semantics of the modeling language.

Here we illustrate our equivalence notion by comparing simple models, but it has also been applied for deriving a comprehensive set of algebraic laws for Alloy. It is also useful for other applications of semantics-preserving model transformations. For instance, we used it in an atomization process [Edwards et al., 2004], which transforms an Alloy model to improve the analysis performance of the Alloy Analyzer tool [Jackson et al., 2000]. We show that this transformation preserves the semantics of the model using algebraic laws and the equivalence notion proposed here [Gheyi et al., 2004]. Moreover, this notion can be used to formally derive model refactorings that can be useful for introducing design patterns [Gamma et al., 1994] into a model. Additionally, a flexible equivalence notion can be useful, for instance, when we are interested in verifying whether partial models are equivalent. This notion can also be valuable for comparing components' specifications. In case they are equivalent, one component can be replaced by another following approaches described elsewhere [Zaremski and Wing, 1997].

The remainder of this paper is organized as follows. Section 2 illustrates, through an example, the main concepts and intuition behind our equivalence notion. Section 3 overviews the PVS system, used here to formalize and prove properties regarding our equivalence notion. Section 4 then formalizes the equivalence notion, followed by Section 5, which shows some of its properties. The following section discusses related work. Finally, Section 7 concludes the paper.

2. Comparing Models

In this section, we use an example in order to introduce the main concepts and intuition behind our equivalence notion, which is formally defined in Section 4. Figure 1 describes two object models [Liskov and Guttag, 2001] of a banking system. Each box in an object model represents a set of objects. The arrows are relations and indicate how objects of a set are related to objects in other sets. For instance, the arrow labeled `accounts` specifies that each object from `Bank` is related to a set of `Account` objects. The multiplicity symbols are defined as follows: `!` (exactly one), `?` (zero or one), `*` (zero or more) and `+` (one or more). Multiplicity symbols can appear on both ends of the arrow. If a multiplicity symbol is omitted, `*` is assumed.

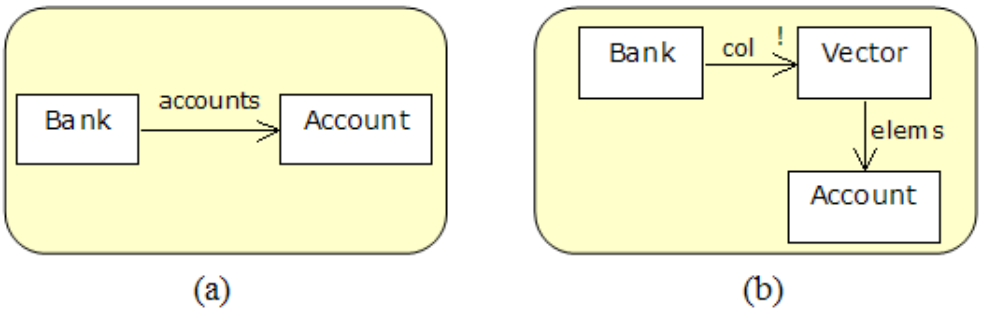

Figure 1: Two Models for a Simplified Banking System

Figure 1(a) shows a model stating that each bank is related directly to a set of accounts, whereas the model in Figure 1(b) establishes that each bank is related to a collection, which is directly related to a set of accounts. Here we are interested in verifying whether they have the same semantics and are, therefore, equivalent alternative designs. Considering, as usual, that the semantics of an object model is the set of valid— that meet all modeled constraints— assignments (interpretations) for its sets and relations, it is enough to verify whether any valid interpretation for one model is also valid for the other, and vice-versa.

For instance, Figure 2(a) shows a valid interpretation for Figure 1(a) model. This interpretation denotes a banking system containing two banks (`b1, b2`) and two accounts (`c1, c2`). The bank `b1` is related to accounts `c1` and `c2` . Figure 2(b) shows a valid interpretation for Figure 1(b) model. It has the same banks and accounts of Figure 2(a) interpretation, however the bank `b1` is related to the vector `v1`, which is related to accounts `c1` and `c2`.

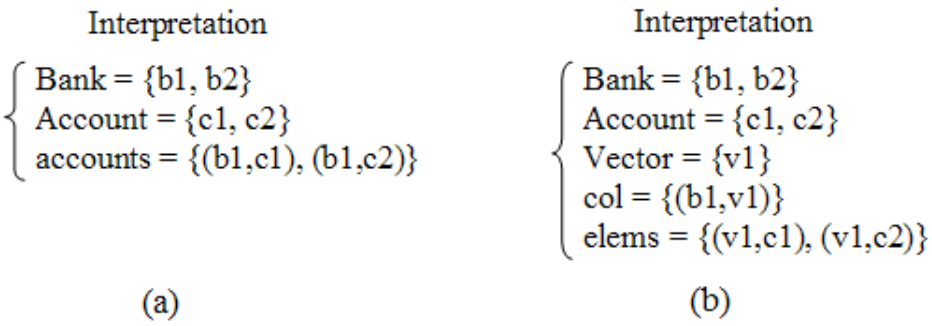

Figure 2: Interpretations

This common equivalence notion, which compares whether two models have the same semantics, is useful, but not flexible enough to compare equivalent models with auxiliary elements such as `Vector`, or with different forms of representing the same concept, such as `accounts` in Figure 1(a). The models in Figure 1 are intuitively equivalent, taking into consideration the relationship between banks and accounts, which is maintained whether there is an intermediate collection or not. However, these models are not equivalent under this equivalence notion; the interpretations, such as Figure 2(a) interpretation, for the model in Figure 1(a) have, for example, values assigned to `accounts`, contrasting with the interpretations, such as Figure 2(b) interpretation, for the model in Figure 1(b), where `accounts` does not appear.

In order to compare models such as those ones, we propose a flexible equivalence notion. Our approach compares the semantics of two object models only for a number of relevant model elements (sets and relations), abstracting the values assigned to the others. The set of relevant elements 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. If both models have the same interpretations for those names, they are considered to be equivalent under this equivalence notion. The other names, such as `col`, `Vector` and `elems`, are regarded as auxiliary, thus not considered when searching for an equivalent interpretation in the corresponding model. So, now we are able to compare Figure 1 models.

However, sometimes we might have model elements that, although relevant, cannot be compared, since they are not part of both models. For instance, suppose that we include `accounts` to Σ. In this case, we cannot compare Figure 1 models since `accounts` does not appear in Figure 1(b) model. Another example, 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 concepts. For instance, in Figure 1(b), `accounts` 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, with n and exp having the same type.

Notice that although the values of auxiliary names are not compared, they can be used to yield the values of other names.

In the example, we may choose a v containing the following item: *accounts*→*col.elems*, where the dot operator represents relational composition. Now we can infer that both models are equivalent. Notice that `accounts` is defined in terms of two names that belong to the model in Figure 1(b). Using this item, we can extend Figure 2(b) interpretation with `accounts`'s value, as shown in Figure 3; hence we can compare the values of all names in Σ. The view allows a strategy for representing relevant elements using an equivalent combination of other elements.

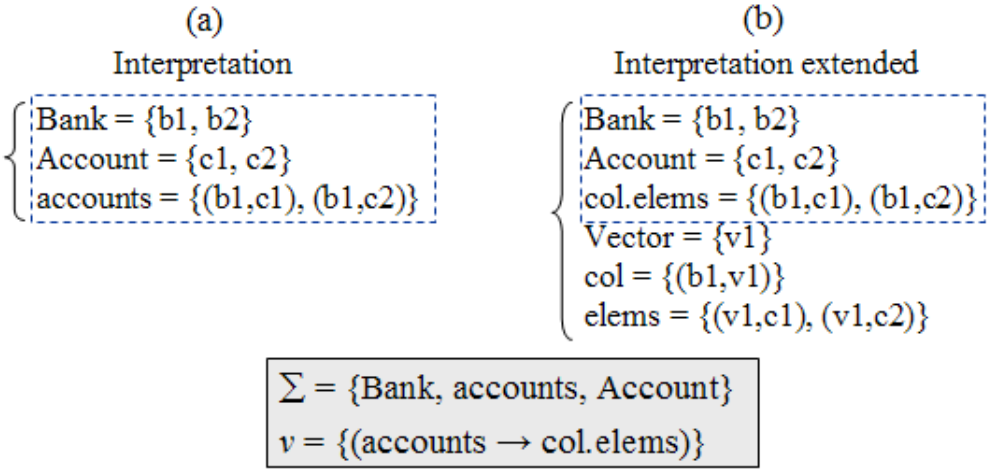

Figure 3: Interpretation Extended

It is important to mention that two models can be equivalent with respect to an alphabet and a view, and the choice of a different alphabet or view yields a different notion. For example, if we take the same Σ (`Bank`, `Account`, `accounts`) and v containing the following item: *accounts*→(*Bank*`->`*Account*), where the '`->`' operator denotes cartesian product. In this case, these models are not equivalent, since they may have different values for `accounts` in some interpretations, such as Figure 2 interpretations. Now suppose that we take Σ to contain the `Bank`, `Account`, `accounts` and `col` names. In this case, these models cannot be compared since it is impossible to provide a view representation for `col` in Figure 1(a).

Our equivalence notion based on alphabets and views is more flexible since it supports abstraction from names and elements when comparing models. By choosing specific alphabets and views, as desired, the developer can choose the appropriate abstraction level for a given situation. In fact, the usual equivalence notion is a particular instantiation of our notion when we simply take an empty view and an alphabet containing all names in the model.

3. PVS Overview

The Prototype Verification System (PVS) provides mechanized support for formal specification and verification [Owre et al., 2001a]. The PVS system consists of a specification language, a prover, specification libraries, and several browsing tools. The language is based on simply typed higher-order logic. Each specification consists of a collection of theories. Each theory may introduce types, variables, constants, and may introduce axioms, definitions and theorems associated with the theory. Specifications are strongly typed, meaning that every expression has an associated type.

Suppose that we want to model part of a banking system in PVS, on which each bank contains a set of accounts, and each account has an owner and a balance. Next, we declare a theory named `BankingSystem` that declares two uninterpreted types (`Bank` and `Person`), representing sets of banks and persons, and a record type denoting an account. An uninterpreted type imposes no assumptions on implementations of the specification, contrasting with interpreted types such as `int`, which imposes all axioms of the integer numbers. Record types, such as `Account`, impose an assumption that it is empty if any of its components types is empty, since the resulting type is given by the cartesian products of their constituents. The `owner` and `balance` are fields of `Account`, denoting the account's owner ant its balance, respectively.

```
BankingSystem: THEORY
BEGIN
  Bank: TYPE
  Person: TYPE
  Account: TYPE = [# owner: Person, balance: int #]
```

Each theory may import other theories using the `IMPORTING` clause. By default, all theories import the *prelude* library, which provides, among other things, the boolean operators, sets, equality, and the real, rational, integer and natural number types and their associated properties [Owre et al., 2001a].

In PVS, we can also declare function types. Next, we declare two functions types (mathematical relation and function, respectively). There are various forms to declare functions in PVS. One of them is to just declare its name and parameters and result types, such as the first function establishing that each bank relates to a set of accounts. Another way is to also define the associated mapping, as in the second function, which denotes the withdraw operation.

```
  accounts: [Bank -> set[Account]]

  withdraw(acc: Account, amount: int): Account =
    acc WITH [balance := (balance(acc)-amount)]
```

The `balance(acc)` expression denotes the balance of the `acc` account. The `WITH` keyword denotes the override operator. In the `withdraw` function, the expression containing the `WITH` operator denotes an account with the same owner of `acc`, but with a balance subtracted of `amount`. Similarly, we can declare a function representing the credit operation.

Besides declaring types and functions, a PVS specification can also declare axioms, lemmas and theorems. For instance, next we declare a theorem stating that the balance of an account is not changed when performing the withdraw operation after the credit operation with the same amount.

```
  withdrawCreditTheorem: THEOREM
    FORALL(acc: Account, amount: int) :
      balance(withdraw(credit(acc,amount),amount)) = balance(acc)
END BankingSystem
```

The `FORALL` keyword denotes the universal quantifier. The previous quantification is over an account and an amount to be deposited and then withdraw.

The PVS proof checker provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. For instance, the previous theorem can be proved by applying the `expand` rule twice, which expands a definition at its occurrence, in the `withdraw`

and `deposit` functions. These proof commands can be combined to form proof strategies [Owre et al., 2001b]. For instance, we can prove the previous theorem by just applying the `grind` strategy, which installs rewritings and successive simplifications.

4. Equivalence Notion

In this section, we formalize our abstract notion of object models equivalence. As mentioned in Section 2, we consider that two models are equivalent with respect to an alphabet and a view. An alphabet contains a set of element names and a view contains a set of mappings, each one mapping a name to an expression, as described by the meta-model in Figure 4 which can be directly translated to PVS. The `Expression` type denotes expressions from the object modeling language.

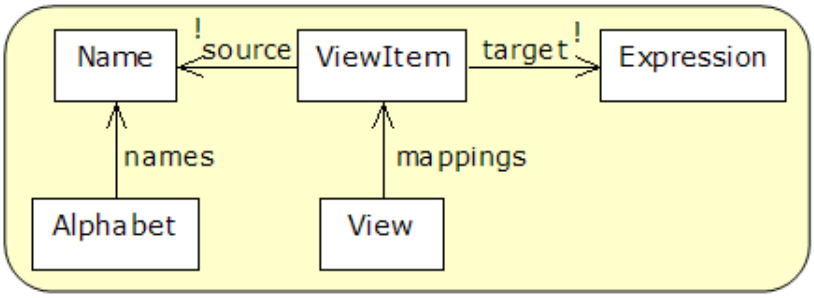

Figure 4: Alphabet and View

After describing alphabets and views, the conditions for equivalence between models are established. Two models are equivalent with respect to an alphabet and a view, given the view is valid for both models. Equivalence is simply defined as mutual refinement, modulo the view and alphabet, as stated as follows:

```
equivalent(m1,m2:Model, a:Alphabet,
    v:{ vi:View | valid(vi,m1,a) ∧ valid(vi,m2,a) }): boolean =
  refines(m1,m2,a,v) ∧ refines(m2,m1,a,v)
```

Hereafter, we used some mathematical symbols rather than PVS keywords and functions in order to improve readability. The `Model` type represents the models of the object modeling language that is subject to the equivalence notion. In PVS, we can declare dependent types [Owre et al., 2001a], such as the type of the `v` parameter in the previous function. It establishes that the relation is only defined, and can be applied, for arguments `v` that are valid views for both models. Thus, models are not compared when the considered view is invalid. In the following sections, we explain our notions of valid views and refinement for models.

4.1. Valid View

As mentioned before, a view must satisfy some properties in order to be valid for a given model and alphabet. A valid view must indicate, in a unambiguous form, how elements in the alphabet that are not part of the considered model can be expressed, in terms of alternative elements in that model. We explain that by using the example in Figure 5, which illustrates the equivalence between extended versions of the two banking systems described in Figure 1. Besides each bank being related to a set of accounts, each bank is related to a set of customers. Moreover, each account is owned by a customer and there are two kinds of accounts: checking and savings.

In this Figure, an arrow with a closed head form, such as the one from `ChAcc` to `Account`, denotes a subset relationship. In this case, `ChAcc` is a subset of `Account`.

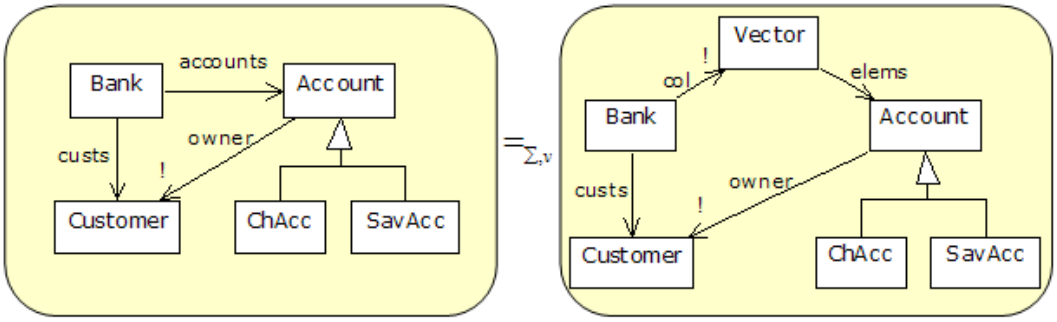

Figure 5: Equivalence Notion of an Extended Banking System

Since the two subsets share an arrow, they are disjoint. If the arrowhead is filled, the subsets exhaust the superset, so there are no members of the superset that are not members of one of the subsets. In both object models representing a banking system, the subsets form a *partition*: every member of the superset belongs to exactly one subset.

The first property of a valid view states that it cannot be recursive. As a consequence, each item of a view cannot refer to its `source` name in the `target` expression. For instance, we do not allow an item in a view such as *ChAcc*→*Account* & *ChAcc*, where the '&' operator represents set intersection. This restriction is due to the fact that a view item must represent an unambiguous mapping for an element in the alphabet, which is impossible for a recursive item, which tries to map an element to itself. It is important to mention that we allow a view containing the following items: *ChAcc*→*Account* and *Account*→*ChAcc*. Our notion ensures that the value for each name must be yielded by a single mapping; hence avoiding mutual recursion.

Another property is that a valid view must have only items that map names in the alphabet. Suppose that Σ consists of the `Bank`, `Account`, `Customer`, `owner`, `custs` and `accounts` names. So we do not allow mappings for `ChAcc`, which is not in the alphabet, since its values are not be used to compare the models. For example, a view containing the *ChAcc*→*Account* − *SavAcc* mapping, where the '−' operator represents set difference, would be invalid.

Finally, the last property establishes, which is described in the following PVS function, that the alphabet's names not in the model must be mapped by the view. Moreover, the view must have exactly one valid item for that name and the model under consideration. For instance, the right-hand side model of the banking system in Figure 5 does not contain the `accounts` relation, despite of its presence in the alphabet. Consider a view with two mappings: *accounts*→*col*.*elems* and *accounts*→*custs*.˜*owner*, where the '˜' operator yields the transpose of a relation. Notice that both mappings can be used to express `accounts`'s value since the expression of each item only contains names that belong to the right-hand side model of Figure 5. Therefore, both expressions must yield the same value, in order to maintain consistency. However, in order to avoid this additional complexity, we establish that for all names in the alphabet that are not in the model, the view must have exactly one item that can express its value.

```
hasMappingsForAlphabet(m:Model, a:Alphabet, v:View): boolean =
  ∀ n:names(a) | n ∉ names(m) ⇒ oneValidItem(n,m,v)
```

Notice that the first two properties previously described are model-independent. Moreover, as mentioned before, the name and expression part of each view item must have the same type. It is important to stress that a view may have more than one item for a name. However, if a model does not contain an alphabet's name, the view must present exactly one valid item for this name. Observe these properties include only syntactic

conditions; hence it is straightforward to automatically check whether a view is valid for a model.

4.2. Refinement

As previously defined, two models are equivalent precisely when they refine one another. We have then to formalize our notion of refinement. Given two models `m1` and `m2`, we say that `m1` refines `m2` if, for each interpretation that satisfies `m1`, there exists an equivalent interpretation that satisfies `m2`.

```
refines(m1,m2:Model, a:Alphabet,
    v:{ vi:View | valid(vi,m1,a) ∧ valid(vi,m2,a) }): boolean =
  ∀ i1:semantics(m1) | ∃ i2:semantics(m2) |
      equivalentMappings(m1,m2,i1,i2,a,v)
```

We consider that the interpretations are equivalent with respect to an alphabet and a view, possibly abstracting names and elements, as shown next. Each `Interpretation` has a relation `map`, which maps names to values, as shown in the following PVS fragment. A value is modeled by the `Value` type.

```
Interpretation: TYPE = [# map: [Name->set Value] #]
```

The `semantics` function yields all valid interpretations of a model and it is defined according to the object modeling notation in use.

Actually, the interpretations are compared only with respect to the names in the alphabet. That is, both interpretations should assign the same values to each name in the alphabet.

```
equivalentMappings(m1,m2:Model,i1,i2:Interpretation,a:Alphabet,
    v:{ vi:View | valid(vi,m1,a) ∧ valid(vi,m2,a) }): boolean =
  ∀ n:names(a) | mappings(m1,a,n,i1,v) = mappings(m2,a,n,i2,v)
```

But when such a name is not in one of the models, it is certainly not mapped by the corresponding interpretation; so we cannot directly compare the assigned values. In those cases, we first apply to the interpretation the adequate view item in order to get the indirectly assigned values for that name. The following function applies a view to an interpretation. It receives an interpretation and yields the same interpretation with a mapping for the view item's source. So it evaluates the target expression in the interpretation received as a parameter, and maps it to the source of the item.

These values should then be the same assigned to that name by the other interpretation, otherwise they are not considered equivalent. This is formalized by the `mappings` function, which is described next, and yields the values for the `n` name within `m`, for the `i` interpretation. It computes the value of `n` in `i`, whether it belongs to the considered model or not. In the latter situation, the function applies a view before yielding the values for `n`.

```
mappings(m:Model, a:Alphabet, n:Name, i:Interpretation,
         v:{ vi:View | valid(vi,m,a) }): ℙ Value =
  if(n ∈ names(m)) then map(i)(n)
  else map(applyView(i,the(getViewItem(n,m,v))))(n)
```

The `applyView` function takes an interpretation and a view item and yields the same interpretation received as argument, but with an additional mapping for the name of the item received as a argument and its value. It actually extends an interpretation in such a way that it can be directly compared with another one.

```
applyView(i:Interpretation, v:ViewItem): Interpretation =
  i WITH [map := map(i) WITH
              [(source(v)) |-> (evalExpression(target(v),i))]]
```

The application of the view always works because we assume it is valid for each model. Therefore, there is exactly one valid item for each name that does not belong to the model in consideration.

5. Properties of the Equivalence Notion

First of all, we proved that our notion is an equivalence relation (for a fixed alphabet and view, it is reflexive, symmetric and transitive). The following theorem states that our equivalence relation is symmetric.

```
equivalentSymmetry: THEOREM
  ∀ (m1,m2:Model, a:Alphabet,
     v:{ vi:View | valid(vi,m1,a) ∧ valid(vi,m2,a) }) |
    equivalent(m1,m2,a,v) ⇒ equivalent(m2,m1,a,v)
```

After proving these properties, we proved other general properties using PVS. Those properties are specially useful when applying a sequence of semantics-preserving model transformations [Gheyi et al., 2004], which lead to a chain of equivalent models. This might be necessary, for instance, to introduce a design pattern [Gamma et al., 1994] into a model. In these situations, sometimes we might refactor models and only after a while notice that we have not chosen the appropriate alphabet and view. We might, for example, need an extra item in the view. One naive way to solve this problem is to restart the refactoring from scratch considering the correct alphabet and view. However, this is very time consuming, since it involves calculating the semantics of each model or applying several transformations again. The properties introduced in this section are a better solution to that.

The properties considered here are general laws about our equivalence notion, establishing how we can manipulate views and alphabets while preserving the equivalence. For instance, Figure 6 shows the banking system models described in Section 2. Suppose that both models on the top are equivalent considering a view $v1$ containing only the mappings *accounts*→*col*.*elems* and *Account*→*ChAcc* + *SavAcc* (the '+' operator denotes set union), and an alphabet Σ containing only the `Bank`, `Account` and `accounts` names. Suppose that we noticed that we have not chosen the appropriate view, since *Account*→*ChAcc* + *SavAcc* is not used; it would be desirable to remove this item from $v1$.

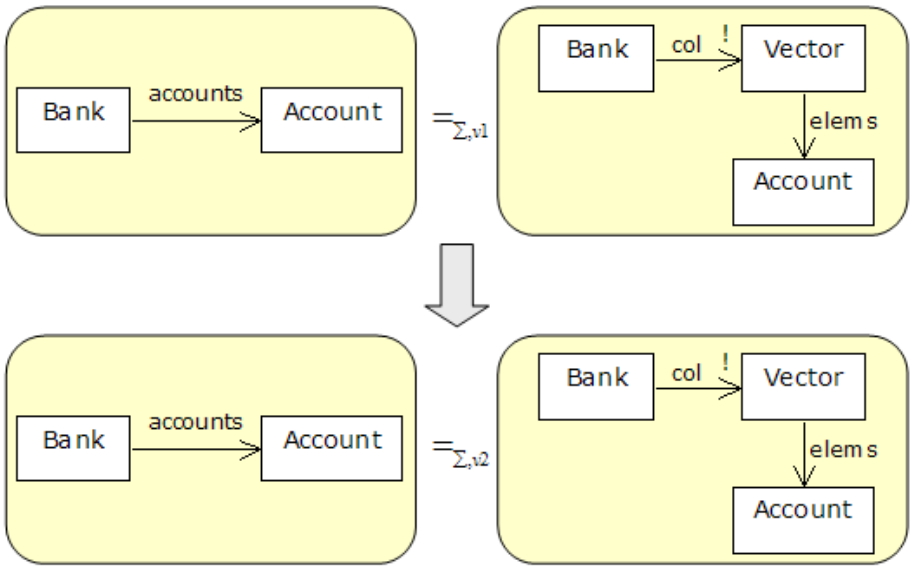

Figure 6: Changing a View

In order to assure that both models are also equivalent with the reduced view, we need to check that the new view is still valid for both models and that each model refines the other. In order to $v2$ be valid, it cannot be recursive. Since `M1` and `M2` are equivalent in $v1$ and Σ, $v1$ must be a valid view for both models; hence all mappings are not recursive. So, $v2$ is not recursive because its mappings are all in $v1$.

The second property of a valid view makes sure that the view only maps elements in the alphabet. Since $v1$ is a valid view, it only maps mappings in the alphabet. This property is preserved when removing an item of the view. Finally, the last property of a valid view ensures that there is exactly one valid item for each name of each alphabet that does not belong to the models. Since $v1$ is a valid view for `M1` and `M2`, it already has mappings for all names of `M1` and `M2` that are not in the alphabet. Therefore, we must guarantee that if the name mapped by `it`, which is the item to be removed, does not belong to `M1` or `M2`, then the expression mapped by this name cannot contain all names of `M1` or `M2`. This constraint ensures that `it` is not used when comparing interpretations, hence preserving the property that the view has only one valid item for each name in a model.

Now we must check that the refinement relations hold with respect to $v2$ too, so that we can be sure that the models are equivalent with $v2$. Notice that the refinement relation just compares values of the names in the alphabet. Since the previous constraints ensure that the removed item is not used by the comparison, we guarantee that the refinement relations hold with $v2$ as well. This is the corresponding theorem proved in PVS.

```
decreasingView: THEOREM
  ∀ (m1,m2:Model, a:Alphabet, v1,v2:View, it:ViewItem) |
    (equivalent(m1,m2,a,v1) ∧
     mappings(v1) = mappings(v2) ∪ { it } ∧
     (source(it) ∉ names(m1) ⇒
        ¬ (expNames(target(it)) ⊆ names(m1))) ∧
     (source(it) ∉ names(m2) ⇒
        ¬ (expNames(target(it)) ⊆ names(m2)))) ⇒
           equivalent(m1,m2,a,v2)
```

This law (property) can then be applied when dealing with sequences of transformations. Observe that we have syntactic conditions that can be easily automated by a tool when decreasing a view. Moreover, these constraints just involve the item to be removed. Similarly, we proved a similar theorem for increasing a view. We just have to check the same properties of decreasing a view, and two other constraints verifying that the new item cannot be recursive and maps an element of the alphabet. Observe that these constraints imply that we can only introduce an item that is not used to compare the corresponding models.

Figure 7 shows the same banking system models described before with the same view and alphabet. Now suppose that we do not want to consider `Bank` in the alphabet. When decreasing an alphabet, we have just to make sure that the v still relates mappings in the alphabet. Since we do not change the view, it is not recursive, and has only one valid item for each name that is not in `M1` and `M2`. Since both models are equivalent in a larger alphabet, intuitively they are still equivalent in a subset of it; the more we abstract the more we equate.

```
decreasingAlphabet: THEOREM
  ∀ (m1,m2:Model, a1,a2:Alphabet, v:View) |
    equivalent(m1,m2,a1,v) ∧
    names(a2) ⊆ names(a1) ∧
    hasOnlyItemsForAlphabet(a2,v) ⇒
```

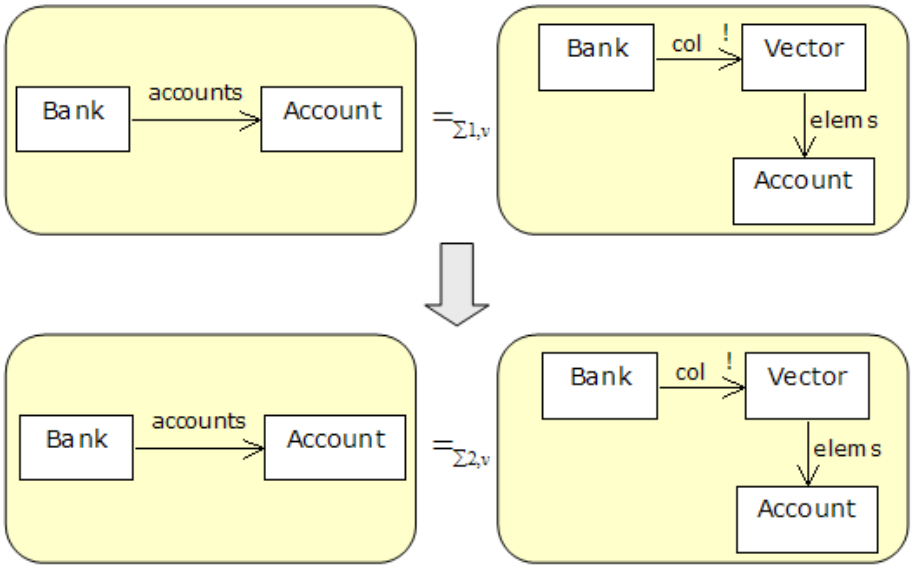

Figure 7: Changing an Alphabet

```
equivalent(m1,m2,a2,v)
```

Considering the view and alphabet mentioned before, both models are still equivalent. However, if we choose to remove `Account` from the alphabet, we cannot remove it since the view contains an item for it. In this case, we first have to decrease the view, removing the item for it; then we can decrease the alphabet. It is important to observe that the conditions for decreasing a view are also syntactic. For increasing an alphabet, we need some semantic conditions.

These properties are important in a chain of equivalent models while refactoring them since there is no need to compute the semantics of each model again in the chain. We have just to check some syntactic conditions in the view and alphabet involved. As a future work, we intend to prove other properties of the equivalence notion, such as compositionality.

6. Related Work

Related work [Nuka and Woodcock, 2004] has been carried out giving a formal semantics and laws to the Alphabetized Relational Calculus (ARC), which adds a theory of alphabets to relational calculus [Tarski, 1941]. Each ARC specification contains an associated alphabet that is equivalent to our notion of alphabet. They used an equivalence notion for comparing models in the laws, stating that two ARC models are equivalent if they have the values for all names in the specification. Their equivalence notion can be used to compare ARC models with different names. In order to do that, they give any possible values to the names that are not in each model; in fact, they only compare the values that are in common of both models.

A similar approach [Borges, 2004] proposes and formalizes a simplified equivalence notion for Z specifications [Spivey, 1989]. They show how to refine a model with an association by another with an attribute, where both models have the same names. They prove this refinement using the Z/EVES theorem prover [Saaltink, 1997]. Our equivalence notion can also be seen as a traditional refinement of Z specifications, as proposed by the authors. However, our refinement relation must be symmetric, differently than the proposed by them [Borges, 2004].

Transformation rules [Gogolla and Richters, 1998] has been proposed for UML class diagrams [Booch et al., 1999]. They state when two class diagrams are equivalent. One distinction from our work is that the equivalence notion is necessarily to be symmetric. Further, some of the rules compare models with different names. Nevertheless, they do not define a general equivalence notion stating when two class diagrams are equivalent. This notion is based on an informal UML semantics. Therefore, a few transformation rules may not preserve semantics in some situations, as previously described [Gheyi and Borba, 2004].

A previous version of the equivalence notion proposed here is described for comparing Alloy models [Gheyi and Borba, 2004], which may have different elements. However, this previous notion, which uses an implicit alphabet, is limited since it actually compares only elements that belong to both Alloy models. Moreover, our notion is defined for any object modeling notation, differently than the related approaches mentioned in this section.

A similar work proposes laws of programming for Refinement Object-Oriented Language (ROOL) [Borba et al., 2003], which is a language similar to Java [Gosling et al., 1996]. This related work is similar to ours in the sense that they propose an equivalence notion for programs. They state that two programs are equivalent if each program refines the other. Moreover, this refinement relationship is based on the weakest precondition semantics [Cavalcanti and Naumann, 2000] of the main commands. Our equivalence notion deals with abstract object models, rather than programs.

Another work proposes refactorings [Fowler, 1999] for Java [Gosling et al., 1996] programs. Each refactoring changes the internal structure of the software to make it easier to understand and cheaper to modify without affecting its observable behavior of the software [Fowler, 1999]. The author guarantees that two programs have the same behavior if they do not have failures in a test suite. Our equivalence notion deals with models instead of code. Moreover, our notion can be used to compare models with respect to the relevant structure, similarly proposed by the author which compares the observable behavior of the software.

7. Conclusions

In this paper, we propose an abstract equivalence notion for object models. It supports abstraction from names and elements when comparing models. Moreover, it could also be useful for comparing parts of models. Our equivalence notion compares only semantic interpretations for model elements in an alphabet, and elements from this alphabet do not need to be in the considered model, since they are expressed in terms of other model elements by the views. We also show some useful properties, proved by means of the PVS prover, which can syntactically check whether previous equivalent models are still equivalent when alphabets and views are changed.

Previously, we proposed an equivalence notion described in Alloy, first only for Alloy models [Gheyi, 2004]. Since the Alloy Analyzer [Jackson et al., 2000], which is a tool used to analyze Alloy models, is not a theorem prover, we manually proved that our notion is actually an equivalence relation. However, using Alloy as the framework for specifying any modeling language can be particularly interesting, since we are able to use the tool before proving a property. We specified each property of the equivalence notion in a logical assertion, then asking the tool to verify whether the assertion is valid for a predefined scope. The counterexamples generated by the tool help us understanding the equivalence notion. Only after the tool does not give any counterexample, we started to prove manually, later by using PVS.

This equivalence notion can be useful in several contexts, such as formalization of the atomization process in Alloy's analysis [Edwards et al., 2004] (along with modeling laws), derivation of formal refactorings and introduction of design patterns [Gamma et al., 1994] into models, and comparison of components' specifications [Zaremski and Wing, 1997]. Although our equivalence notion was initially defined in terms of the Alloy modeling language, allowing the proof of modeling laws for Alloy [Gheyi et al., 2004], this notion is abstract enough to be applicable to other object modeling languages, such as the UML [Booch et al., 1999]. In fact, proofs in PVS show the independence of the notion with respect to the semantics of the modeling language.

Considering future work, we intend to derive and prove a number of additional properties of this equivalent notion, in order to ensure that our notion is widely applicable and make the proof process more intuitive to modelers. For instance, it is desirable to prove the compositionality property of the equivalence notion. In addition, we aim to relate this equivalence notion to the classic notion of data refinement, as employed by Woodcock and Davies [Woodcock and Davies, 1996].

Acknowlegments

We would like to thank Augusto Sampaio, Daniel Jackson and members of the Software Design Group at MIT, Carlos Pombo, and members of the Software Productivity Group at UFPE, for their important comments. This work was supported by CNPq and CAPES (Brazilian research agencies).

References

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

[Borba et al., 2003] Borba, P., Sampaio, A., and Cornélio, M. (2003). A refinement algebra for object-oriented programming. In *17th European Conference on Object-Oriented Programming, ECOOP'03*, pages 457–482, Darmstadt, Germany.

[Borges, 2004] Borges, R. (2004). Integrando UML e métodos formais. Technical Report, Federal University of Pernambuco, Informatics Center.

[Cavalcanti and Naumann, 2000] Cavalcanti, A. and Naumann, D. (2000). A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 26(8):713–728.

[Edwards et al., 2004] Edwards, J., Jackson, D., Torlak, E., and Yeung, V. (2004). Faster constraint solving with subtypes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 232–242. ACM Press.

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

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Gheyi, 2004] Gheyi, R. (2004). Basic laws of object modeling. Master's thesis, Federal University of Pernambuco.

[Gheyi and Borba, 2004] Gheyi, R. and Borba, P. (2004). Refactoring alloy specifications. In Cavalcanti, A. and Machado, P., editors, *Electronic Notes in Theoretical Computer*

Science, Proceedings of the Brazilian Workshop on Formal Methods, volume 95, pages 227–243. Elsevier.

[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.

[Gogolla and Richters, 1998] Gogolla, M. and Richters, M. (1998). Equivalence rules for UML class diagrams. In *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France*, pages 87–96.

[Gosling et al., 1996] Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. Addison-Wesley.

[Jackson, 2004] Jackson, D. (2004). Alloy 3.0 reference manual. At http://alloy.mit.edu/beta/reference-manual.pdf.

[Jackson et al., 2000] Jackson, D., Schechter, I., and Shlyahter, H. (2000). Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 730–733. ACM Press.

[Liskov and Guttag, 2001] Liskov, B. and Guttag, J. (2001). *Program Development in Java*. Addison-Wesley.

[Nuka and Woodcock, 2004] Nuka, G. and Woodcock, J. (2004). Mechanising the alphabetised relational calculus. In Cavalcanti, A. and Machado, P., editors, *Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods*, volume 95, pages 209–225. Elsevier.

[Owre et al., 2001a] Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (2001a). PVS language reference. At http://pvs.csl.sri.com.

[Owre et al., 2001b] Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (2001b). PVS prover guide. At http://pvs.csl.sri.com.

[Saaltink, 1997] Saaltink, M. (1997). The Z/EVES system. In Bowen, J., Hinchey, M., and Till, D., editors, *ZUM'97: Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85. Springer-Verlag.

[Spivey, 1989] Spivey, J. (1989). *The Z Notation: A Reference Manual*. C. A. R. Hoare Series Editor. Prentice Hall.

[Tarski, 1941] Tarski, A. (1941). On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89.

[Woodcock and Davies, 1996] Woodcock, J. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*. Prentice Hall.

[Zaremski and Wing, 1997] Zaremski, A. and Wing, J. (1997). Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369.