A Rigorous Approach for Proving Model Refactorings

Rohit Gheyi
rg@cin.ufpe.br

Tiago Massoni
tlm@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Brazil

ABSTRACT

Both model and program refactorings are usually proposed in an *ad hoc* way because it is difficult to prove that they are sound with respect to a formal semantics. In this paper, we propose guidelines on how to rigorously prove model refactorings for Alloy, a formal object-oriented modeling language. We use the Prototype Verification System (PVS) to specify and prove the soundness of the transformations. Proposing refactorings in this way can facilitate not only design, but also improve the quality of refactoring tools.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal Methods; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Design, Verification

Keywords

Model Refactoring, Theorem Proving

1. INTRODUCTION

Evolution is an important and demanding software development activity. Modern development practices, such as program refactoring [3], improve programs while maintaining their original behavior, in order, for instance, to prepare software for change. In current practice, even using refactoring tools, programmers have to rely on compilation and a good test suite to ensure that it preserves behavior [3]. In case of model refactorings, most proposed transformations rely on informal argumentation.

So, simple mistakes may introduce inconsistencies to a model or make a program ill-typed. This may be unacceptable, especially for developing critical software systems. To our knowledge, there are a few guidelines for formal proofs by refactoring designers, which are responsible for proposing new refactorings.

In this paper, we propose a rigorous approach to formalize and prove structural model refactorings for Alloy [6], a formal object-oriented modeling language. Our approach consists of proposing a formal semantics for a core language (Sections 2) and an equivalence notion (Section 3), before stating and proving any transformation for this language (Section 3). The Prototype Verification System (PVS) [7], which encompasses a formal specification language and a theorem prover, will be used to specify and prove that the model refactorings proposed for Alloy are sound.

Even popular program refactoring tools, such as Eclipse can introduce some simple errors, such as not making behavior-preserving transformation. In case of model refactoring tools, this scenario is even worse since there are a few model transformations proposed, and most of them proposed in an ad hoc way. Proposing refactorings following our approach can help to improve the tool support making software development more reliable.

2. LANGUAGE

In this paper, we show how to verify whether a model transformation is semantics-preserving. In order to do that, first we define a core language and a semantics of the language. We chose Alloy [6].

2.1 Core Language

The definition of a core language is very important in order to prove refactorings, since if we do not remove syntactic sugar constructs, the definition of semantics and proofs will increase in size and complexity.

An Alloy model consists of a set of modules. We do not consider all Alloy constructs; for instance we omit imports. Therefore, we consider that each model consists of a single module, which does not restrain the languages expressiveness. Moreover, functions and predicates are actually syntactic. Similarly, the assertion paragraph and commands are ignored since they are used by the Alloy Analyzer for performing analysis; they do not affect the meaning of a model. So we consider that an Alloy model contains a set of signatures and facts.

Regarding signatures, we do not consider syntactic sugar constructs, such as `abstract`. The constraints can be represented by formulae in facts. So each signature has a name, may extend a signature, and may declare a set of relations. Next, we declare part of the grammar considered in the core language.

```
module ::= (signature | fact)*
signature ::= sig sigName [extends sigName] {
                (relName: set sigName,)*
              }
fact ::= fact { (formula)* }
```

All relations must be declared with the `set` qualifier. The other qualifiers (`one` and `sole`) are syntactic sugar. Moreover, we cannot declare the right type of a relation `r` with an expression `exp`. The right type must be always a signature name. However, we can declare this constraint (the values given to the right type of `r` is a subset of the values given to `exp`) in a fact. Additionally, in order to facilitate semantic definitions, we consider that an Alloy model cannot declare two relations with the same name. It is important to mention that this constraint does not restrain expressiveness, since we can always rename a relation. We can propose a model refactoring for renaming an Alloy relation using our semantics-preserving model transformations [4].

Each fact may declare a set of formulae. We do not consider named facts since it does not alter the semantics of the language. Signature facts are syntactic sugar. There are subset, equality, negation, conjunction and universal quantification formulae. The other kinds of formulae, such as existential quantification and disjunction of formulae, can be derived from those. Moreover, we consider some binary (union, intersection, difference, join and product) and unary (transpose and transitive closure) expressions.

```
formula ::= expr in expr | expr = expr | not formula |
            formula and formula |
            (all var: sigName | formula)
expr ::= sigName | relName | var | none |
         expr binop expr | unop expr
binop ::= + | & | - | . | ->
unop ::= ~ | ^
sigName, relName, var ::= id
```

All constraints mentioned before do not simplify the expressiveness of the language. However, we follow two additional assumptions in the considered language. Although Alloy has a limited support for integer expressions (addition and subtraction) [6], we do not consider them. Moreover, all relations must be binary.

2.2 Semantics

Before proposing its semantics, we must state Alloy's syntax, not shown here due to its simplicity. In this section, we describe Alloy's semantics. First, we introduce properties of well-formed Alloy models. An Alloy model is well-formed if the signatures and relations are well formed (`wfSigRel`), and the formulae and expressions are well typed (`wellTyped`), as stated next in PVS. We make some small changes in PVS fragments in order to improve readability.

```
wf(m:Model): bool = wfSigRel(m) ∧ wellTyped(m)
```

There are some constraints that define a well-formed signature and relation in Alloy. Next, we declare all well-defined constraints for our core language. For instance, a signature can only extend signatures declared in the same module (`extSigsFromModel`) and an Alloy model cannot have two signatures with the same name (`uniqueSigName`). Moreover, a signature cannot extend itself direct or indirectly (`noRecExtension`). Additionally, the right side type of a relation must be the name of a signature declared in the same model (`relType`). Finally, we add a constraint stating that in a model we cannot have two relations with the same name (`uniqueRelName`).

```
wfSigRel(m:Model): bool =
  extSigsFromModel(m) ∧ uniqueSigName(m) ∧
  noRecExtension(m) ∧ relType(m) ∧ uniqueRelName(m)
```

The formalization of well formed signatures and relations is also another contribution of this paper, which is informally described [6] and not considered [2]. Similarly, we declare functions for verifying whether formulae and expressions are well-typed based on Alloy's new type system [2]. Since these constraints are very similar to the other languages, we do not show them here.

The dynamics semantics of an Alloy model is the set of instances that satisfy all implicit and explicit constraints of a model, as stated next. An instance is a binding of values to signature and relation names.

```
semantics(m:{ md:Model | wf(md)}): set[Instance] =
  { i:Instance | impConsts(m,i) ∧ expConsts(m,i)}
```

Notice that this relation is only applicable to arguments for `m` that are well-formed models [7]. In our core language, there is just one implicit constraint, regarding the `extends` clause, since we eliminate all syntactic sugar constructs. The `impConsts` predicate certifies that values given to a subsignature form a subset of the parent signature and the immediate subtypes of a signature are disjoint. The `expConsts` predicate certifies whether `i` satisfies all module's formulae.

3. MODEL REFACTORINGS

In this section we show an approach for proving that a transformation for an Alloy model preserves semantics. Thus, we need to state a notion establishing when two models are equivalent. There are some simple and intuitive equivalence notions, however not widely applicable. Next we formalize our notion for object models.

3.1 Equivalence Notion

Figure 1 describes two object models (UML-like class diagrams [1]) of a banking system, subject to refactoring. 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.

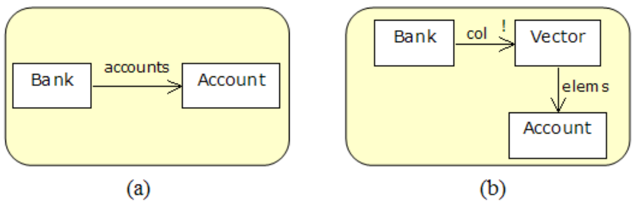

Figure 1: Two Models for a Simplified Banking System

The common notion states that two object models 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`, or with different forms of representing the same concept, such as `accs` in Figure 1(a). The models in Figure 1 are intuitively equivalent, but they are not equivalent using this notion.

In order to compare models in such scenario, we propose a flexible notion. Our approach compares the semantics of two object models only for a number of relevant model elements (signatures and relations), abstracting away the values assigned to the others. The set of relevant element names is called alphabet (Σ). The names that are not in the alphabet are 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 models have the same instances for those names, they are considered to be equivalent under this equivalence notion. Other names, such as `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 models. For instance, suppose that we include `accs` to Σ. In this case, we cannot compare the models in Figure 1, since `accs` is not part of the model in Figure 1(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 concepts. For instance, in Figure 1(b), `accs` 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. In the previous example, we may choose a view containing the following item: $accs {\rightarrow} col.elems$. Now we can infer that both models are equivalent under this notion.

So, two models are equivalent with respect to an alphabet and a view (it is an indexed relation), given the view is valid for both models and both models are well-formed. More details about this and the formalization of this equivalence notion in PVS can be found elsewhere [5]. It is important to formalize it because we will use it when proving that a transformation preserves semantics. The discussion in this section might give an intuition of the problems involved in this definition. There is a tradeoff between the applicability and simplicity of the equivalence notions.

3.2 Transformations

In this section, we propose some primitive structural model transformations that can be used to derive model refactorings, and show how we can define and prove them in PVS. We propose a number of laws, which define two fine-grained localized semantics-preserving transformations that deals with one language construct each time.

Next, we show a law that allows us to introduce a generalization into a model (applying from left to right); similarly can also be used to remove a generalization from a model (applying from right to left). Each law defines two templates of equivalent models on the left and the right side. This law establishes that we can always introduce a generalization declared with a fresh name. It also indicates that we can remove a parent signature that is not being used. Since in Alloy a module cannot have two paragraphs with the same name, we have a condition stating that the new parent signature name does not appear in *ps*. We used *ps*, *rs* and *forms* to denote a set of paragraphs, a set of relation declarations and a set of formulae, respectively.

Law 1. ⟨introduce generalization⟩

```
ps
sig S {
  rs
}
sig T {
  rs'
}
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
sig U {}
sig S extends U {
  rs
}
sig T extends U {
  rs'
}
fact F {
  forms
  U = S + T
}
```

provided
($\leftrightarrow$) if U belongs to Σ, v contains the $U {\rightarrow} S+T$ item;
($\rightarrow$) (1) *ps* does not declare any paragraph named U; (2) all names in Σ that are not on the right side model, v has exactly one valid item for it;
($\leftarrow$) (1) U does not appear in *ps*, *rs*, *rs'* and *forms*; (2) all names in Σ that are not on the left side model, v has exactly one valid item for it.

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 what is required when applying the law in the opposite direction, and we use ($\leftrightarrow$) to indicate that the condition is necessary in both directions. It is important to notice that each primitive law, when applied in any direction, defines one transformation that preserves semantics.

Besides conditions for ensuring that the static semantics is preserved, the law has other constraints in order to preserve the dynamic semantics. For instance, if U belongs to Σ then v must have the $U {\rightarrow} S+T$ item in order to have the same semantics of the right side model. The $+$ operator denotes the union set operator. Moreover, there are conditions for all names in the alphabet that are not in the model. In this case, these constraints assure that v must have an ambiguous way to represent them. It is important to observe that all conditions are syntactic. So, it is straightforward to implement tool support to verify those automatically.

Although this transformation is simple and localized, it is important to prove that it is sound. After formalizing the syntax, semantics and the equivalence notion for the language, now we are able to state a transformation in PVS and verify whether it is sound. A model transformation preserves semantics if it preserves the static and dynamic semantics. Moreover, due to the equivalence notion, we must ensure that each transformation preserves the validity of the view. So we must prove three theorems for each transformation. Next, we will show how to prove that the introduction of a generalization transformation preserves semantics.

First, we describe the syntax of the template models in the transformation, as stated next. We consider that `m1` and `m2` are the left and right side models of the law, respectively. In the previous law, the two models have the same formulae, except for $U = S + T$ (for readability, `g` is the surrogate for $U = S + T$). There are two signatures (`s1` and `s2`) named S one in each side of the law. They are equivalent except that one of them extends U. Additionally, `m2` declares an empty signature `U`. The signatures named T (`t1` and `t2`) and the other elements of the law are declared similarly. We should proceed similarly for other transformations. We map each construction in the law to each corresponding element in Alloy semantics proposed in PVS.

```
syntax(m1,m2:Model, s1,s2,t1,t2,u:Signature): bool =
  formulae(m2)=formulae(m1) ∪ { g } ∧
  name(s1)=name(s2) ∧ relations(s1)=relations(s2) ∧
  extends(s2)={ name(u) } ∧ sigs(m1)(s1) ...
```

Each refactoring includes a number of enabling conditions for ensuring that it preserves semantics. Next we declare a function describing some of the conditions for introducing a generalization, as stated in the law. For simplicity, the `it` item denotes the $U \rightarrow S + T$ item.

```
conditions(m1,m2:Model, s1,s2,t1,t2,u:Signature,
           a:Alphabet, v:View): bool =
  (name(u) ∈ a) ⇒ items(v)(it) ∧
  name(u) ∉ sigNames(m1) ...
```

Now we are able to state three theorems in order to prove that the introduction of a generalization preserves semantics. First we have to prove that a transformation takes a well-formed model to another well-formed model. Next we declare a theorem to verify whether the introduction of a generalization preserves the static semantics.

```
staticSemanticsIntGen: THEOREM
  ∀ m1,m2:Model, s1,s2,t1,t2,u:Signature,
    a:Alphabet, v:View:
      syntax(...) ∧ conditions(...) ∧ wf(m1) ⇒ wf(m2)
```

Moreover, we must guarantee that each transformation preserves the validity of the view, due to our equivalence notion. This `validViewIntGen` theorem is stated similarly. Finally, the last theorem must ensure that the transformation must preserve the dynamic semantics, which uses the equivalence notion proposed before. Notice that since our equivalence notion is defined for well-formed models and the view must be valid for both models, this theorem must use the other two previous theorems. In our case, since we propose and prove laws (bidirectional transformations), the following theorem is slightly different. We prefer to state it in this way in order to improve readability.

```
dynamicSemanticsIntGen: THEOREM
  ∀ m1,m2:Model, s1,s2,t1,t2,u:Signature,
    a:Alphabet, v:View:
      syntax(...) ∧ conditions(...) ∧ wf(m1) ∧
      validView(m1,a,v) ⇒ equivalent(m1,m2,a,v)
```

For other semantics-preserving model transformations for Alloy, such as removing a parent signature, we can state the three theorems and prove them similarly. Notice that once proposed a semantics and an equivalence notion for Alloy, refactoring designers can use them to state and prove in PVS whether any Alloy transformation is semantics-preserving. Following our approach, they just have to define the `syntax` and `conditions` functions for each transformation before proving it. In order to facilitate tool support, the refactoring designers should propose transformations with syntactic conditions.

So far, we have proved 14 semantics-preserving transformations in PVS, laws for introducing a signature, relation, formula and subsignature, and pulling up a relation. Proving some laws in PVS shows us the importance of defining fine-grained transformations. In our opinion, coarse-grained transformations will be more difficult to prove. This experience of proving transformations in PVS was very important not only to understand why an enabling condition is required or not, but also to propose conditions for other model refactorings, which is a difficult task. Proving that a transformation preserves semantics increases the knowledge about other transformations that do not preserve.

4. CONCLUSIONS

In this paper, we provide some guidelines on how to prove model refactorings, such as for Alloy. To our knowledge, there is no attempt to aid refactoring designers in formally proving their refactorings. In order to do that, we propose a formal semantics and an equivalence notion for Alloy. We use PVS to encode the Alloy's transformations, and prove them using its prover. This approach can be used to make more reliable and cost effective model refactoring tools.

The Alloy Analyzer tool, which performs analysis in Alloy specifications, transforms each user model containing subtypes into an optimized version, in order to improve the performance analysis. This process is transparent to the user. Composing our laws proved in PVS, we can formally transform Alloy models in order to optimize the performance analysis [4] of the Alloy Analyzer. Formalizing this process is very important to guarantee the analysis quality, since the user does not want to perform analysis on an optimized model with conflicting semantics. Following the idea of proposing fine-grained transformations, advanced model and program refactoring tools will allow the user not only to use the standard refactorings [3], but also to propose their own. Moreover, proving the refactorings based on a formal semantics allow the development of more reliable refactoring tools, since developers know exactly which conditions are necessary for applying each transformation.

Acknowledgments

We would like to thank all anonymous referees, whose appropriate comments helped improving the paper.

5. REFERENCES

[1] G. Booch et al. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] J. Edwards, D. Jackson and E. Torlak. A type system for object models. In *12th Foundations of software engineering*, pages 189–199, 2004.

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

[4] R. Gheyi, T. Massoni, and P. Borba. Basic Laws of Object Modeling. In *3rd Specification and Verification of Component-Based Systems*, pages 18–25, 2004.

[5] R. Gheyi, T. Massoni, and P. Borba. An Abstract Equivalence Notion for Object Models. *Electronic Notes in Theoretical Computer Science*, 130:3–21, 2005.

[6] D. Jackson, I. Shlyakhter and M. Sridharan. A micromodularity mechanism. In *9th Foundations of software engineering*, pages 62–73, 2001.

[7] S. Owre et al. PVS System Home Page. At http://pvs.csl.sri.com.