Type-safe Refactorings for Alloy

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. *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, not guaranteeing the absence of type errors or semantic changes. Consequently, developers using refactoring tools must rely on compilation and tests to ensure type-correctness and semantics preservation, respectively, which may not be satisfactory to critical software development. In this paper, we propose a static semantics for Alloy, which is a formal object-oriented modeling language, in Prototype Verification System (PVS). The static semantics' formalization can be useful for specifying and proving that transformations in general (not only refactorings) do not introduce type errors, for instance, as we show here.*

1. Introduction

Evolution is an important and demanding software development activity, as the originally defined structure usually does not accommodate adaptations, demanding new ways to reorganize software. Modern development practices, such as program refactoring [Fowler, 1999], improve programs while maintaining their original behavior, in order, for instance, to prepare software for change. Similarly, a *object model refactoring* is a transformation that improves design structure while preserving the semantics. They might bring similar benefits but with a greater impact on cost and productivity, since they are used in earlier stages of the software development process. For instance, model transformations can be used to improve the analysis performance of a tool [Gheyi et al., 2004].

In current practice, even using refactoring tools, programmers must rely on successive compilation and a good test suite to ensure that it does not introduce type errors and preserves the observable behavior, respectively [Fowler, 1999]. A test suite is able only to uncover errors, not to prove their absence. Moreover, usually, modifying the structure of a program, such as extracting a new class, implies updating the test suite in order to add new test cases for the new classes. Therefore, relying on a test suite is not a good way to guarantee behavior preservation. In case of structural model refactorings, most proposed transformations rely on informal argumentation because testing structural models are more difficult than programs. Refactorings are usually proposed in an *ad hoc* way because it is hard to prove that they are sound with respect to a formal semantics. Even a number of model transformations proposed in the literature, which are intended to be semantics-preserving, may lead to incorrect transformations that may introduce type errors in some situations, as we show in Section 2. This may be unacceptable, especially for developing critical software systems.

In this paper, we specify a static semantics for Alloy [Jackson, 2005], which is a formal object-oriented modeling language discussed in Section 4, in the Prototype Verification System (PVS) [Owre et al., 2005], which encompasses a specification language and a theorem prover (as discussed in Section 3). Therefore, the contributions of this paper are the following:

- a static semantics for Alloy in PVS, shown in Section 5;
- the experience of applying this static semantics in proposing and proving in PVS model transformations, such as refactorings, for Alloy, as shown in Section 6.

In a previous work [Gheyi et al., 2004], we propose model transformations for Alloy, showing a number of applications. In another work [Gheyi et al., 2005a], we give a dynamics semantics for Alloy and show how to prove that model transformations for Alloy preserve dynamic semantics. In this work, we focus on the static semantics and show how it can be used in proposing and proving (in PVS) that model transformations for Alloy do not introduce type errors.

One of the most difficult tasks for proposing refactorings is to define required enabling conditions. Proposing and proving refactorings in PVS helps identifying when transformations for Alloy do not introduce type errors. Even popular program refactoring tools, such as Eclipse [Eclipse.org, 2005], may introduce some simple errors, such as transforming a well-typed program into a ill-typed one. Next, we show a simple example describing part of a banking system in Java containing two kinds of accounts (savings and checking). `Account` declares a method `getBalance`, in which we would like to apply the *Push Down Method* refactoring [Fowler, 1999] to `ChAccount`. So, we choose in Eclipse that this refactoring should change `Account` and `ChAccount` classes. After applying this refactoring, the resulting program is ill-typed since the method `debit` uses the method `getBalance`, which is not defined in `SavAccount` anymore.

```
public class Account {
  float balance;
  public float getBalance() { return balance; } ...
}
public class SavAccount extends Account {
  void debit(float amount){ balance = getBalance()-amount; } ...
}
public class ChAccount extends Account {}
```

It is important to mention that it is a very difficult task to state all preconditions required for a transformation to be behavior-preserving. In our opinion, in this particular case the tool should at least warn the user that this transformation may introduce a type error, before applying the refactoring.

In case of structural model refactoring, this scenario is even worse since there are few model transformations proposed in the literature, most of them in an ad hoc way. Consequently, proposing refactorings following our approach can help improve tool support, adding reliability to software development. Although we demonstrate our approach for Alloy, we believe that it can be similarly applied to proving program refactorings. Moreover, the discussion in this paper can be useful for model transformations in general, not only semantics-preserving. So, it can be similarly applied to some transformations in Model Driven Architecture (MDA) [Kleppe et al., 2003]. For example, it is important to show that transformations between Platform Independent Models (PIM) do not introduce type errors.

2. Motivating Examples

In this section, we show how apparently structural semantics-preserving model transformations may introduce some type errors. These examples show that when proposing model transformations, we have to prove not only preservation of dynamic semantics, but also the absence of type errors.

In the context of a simple banking application, Figure 1, which is an object model [Liskov and Guttag, 2001], shows a dynamic semantics-preserving transformation that may introduce a type error. A similar transformation is proposed for UML/OCL [Evans, 1998]. 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. An arrow with a closed head form, such as from `ChAcc` to `Account`, denotes a subtype relationship. The left-hand side (LHS) diagram states that accounts may be checking or savings. Each account is related to a set of bank cards. Moreover, there is an invariant (constraint) stating that savings accounts do not have a bank card.

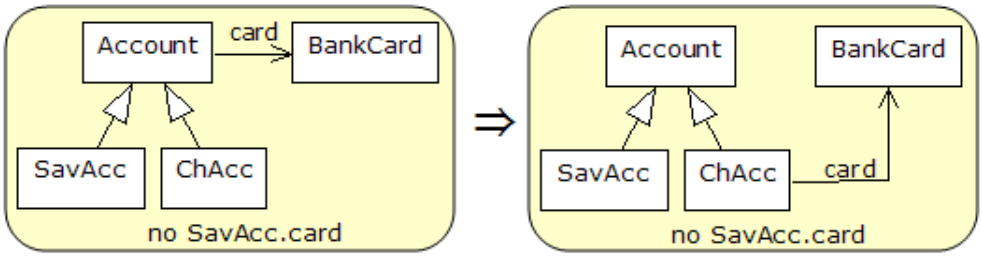

Figure 1: Push down relation

The join operator (`.`), in this case, denotes the standard relational composition. The `no` keyword, when applied to an expression, denotes that this expression has no elements. From this invariant and the fact that accounts can only be checking or savings, we conclude that only checking accounts are related to bank cards. So, we may push down the `card` relation to `ChAcc`, yielding the right-hand side (RHS) diagram. A deeper analysis shows that this transformation, although preserves dynamic semantics, introduces a type error in the refactored diagram. Since `card` is now declared in `ChAcc`, we cannot join `SavAcc` and `card`. So this constraint is no longer well-typed in object models, such as Alloy.

Another example shows a semantics-preserving model transformation for UML class diagrams [Booch et al., 1999] proposed elsewhere [Gogolla and Richters, 1998]. This transformation allows us to convert a generalization into an injective function. Figure 2 shows an example of this transformation. The LHS diagram states that a checking account is a type of account. Moreover, there is a constraint stating that `ChAcc` is a subset of `Account`. The `in` keyword, in this case, denotes the subset operator. Applying the proposed transformation results in a diagram where each checking account is related to exactly one account by the `acc` relation. The generalization is converted into `acc` and a constraint stating that `acc` is an injection. We do not show this constraint for simplicity. Since `ChAcc` and `Account` have different types in the refactored diagram, the constraint stating that checking account is a subset of account is ill-typed. Therefore, the transformation introduces a type error considering Alloy.

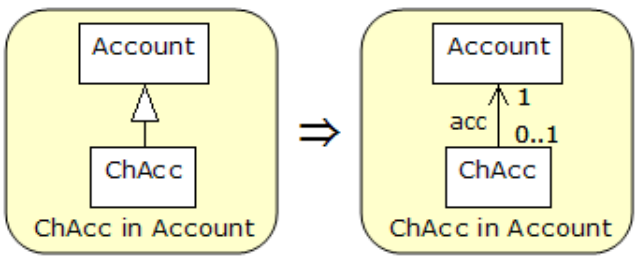

Figure 2: Converting a generalization into a relation

Since the authors [Evans, 1998, Gogolla and Richters, 1998] proposed their

transformation for UML class diagrams, and the Object Constraint Language (OCL) [Kleppe and Warmer, 1999] presents a complex type system [Edwards et al., 2004, Schürr, 2001], these examples may also introduce a type error. Therefore, the examples presented in this section suggest the importance of formally proving not only that a structural model transformation preserves dynamic semantics, but also that they do not introduce any type error. This is also useful for model transformations in general.

3. PVS Overview

The Prototype Verification System (PVS) provides mechanized support for formal specification and verification [Owre et al., 2005]. The PVS system contains a specification language, based on simply typed higher-order logic, and a prover. 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 and its balance, respectively.

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

In PVS, we can also declare function types. Next, we declare two functions types (mathematical relation and function, respectively). The first one just declares its name, parameters and result types, establishing that each bank relates to a set of accounts. The second function not only declares the withdraw operation, but also defines the associated mapping.

```
  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, which replaces the mapping for `acc` by a new tuple, if `acc` is originally in the function domain. 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)
```

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

4. Alloy

In this section, we give an overview of Alloy and propose a core language for it. An Alloy model or specification consists of a set of modules. Each module may contain some signature declarations and paragraphs (constraints and analysis). Signatures are used for defining new types, and constraint paragraphs, such as facts and functions, used to record constraints and expressions. Each signature comprises a set of objects (elements), which associate with other objects by relations declared in the signatures. A signature paragraph introduces a type and a collection of relations, called fields, along with the types of the fields and other constraints on the values they include.

Next, we model part of the banking system in Alloy, on which each bank is related to sets of accounts and customers, and each account may have some owners. The following fragment declares three signatures and three relations. In Bank's declaration, the set qualifier specifies that accounts associates each element in Bank to a set of elements in Account. When we omit the keyword, we specify a total function.

```
sig Bank {
  accounts: set Account,
  customers: set Customer
}
sig Customer {}
sig Account {
  owner: set Customer
}
```

All top level signatures, which do not extend other, are implicitly disjoint. For instance, Account and Bank are disjoint. Moreover, accounts may be checking or savings. In Alloy, one signature can extend another, establishing that the extended signature is a subset of the parent signature. For instance, the values given to ChAcc form a subset of the values given to Account.

```
sig ChAcc, SavAcc extends Account {}
```

Signature extension introduces a subtype. Alloy supports single inheritance. The extensions of a signature are mutually disjoint. In this case, ChAcc and SavAcc are disjoint.

A fact is a kind of constraint paragraph. They are used to record formulae that always hold, such as invariants about the elements. The following example introduces a fact named BankConstraints, establishing general properties about the previously introduced signatures. It contains one formula stating that each account is related to exactly one customer by the owner relation. The all keyword represents the universal quantifier. The one keyword, when applied to an expression, denotes that the expression has exactly one element.

```
fact BankConstraints {
  all acc:Account | one acc.owner
}
```

4.1. Core Language

In this section, we propose a core language for Alloy. Its definition is important for facilitating reasoning and proof of refactorings, as syntactic sugar constructs increase complexity and size of static semantics' definitions.

Module header: We assume that each model consists of a single module. This constraint does not restrain the language's expressiveness [Jackson, 2005]. Each model may contain some signatures and facts. The other constructs are syntactic, as explained next.

```
model ::= (signature | fact)*
```

Signature declarations: Regarding *signatures*, we do not consider syntactic sugar constructs, such as `abstract`. An abstract signature partitions its subsignatures, so all elements of the abstract signature belong to exactly one of its direct subsignatures. This constraint can be represented by a formula in fact. So, each signature has a name, may extend a signature, in addition to possibly declaring a set of relations. Next, we declare part of the grammar considered in the core language.

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

We only consider binary *relations*, and all of them must be declared with the `set` qualifier, as the other qualifiers (`one` (total function), `some` and `lone` (partial function)) are syntactic sugar. In this core language, we cannot declare the right type of a relation `r` with an expression `exp`. The right type is always the name of a signature. However, we can declare this constraint — the values given to the right type of `r` are a subset of the values given to `exp` — in a fact. Additionally, in order to facilitate the semantic definition, 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. Indeed, a model refactoring can be defined for renaming an Alloy relation from four (introduce/remove relation and formula) of our semantics-preserving model transformations proposed elsewhere [Gheyi et al., 2004].

Constraint paragraphs: In Alloy, we have three kinds of constraint paragraphs: facts, functions and predicates. Regarding *facts*, we do not consider names since they do not alter the semantics of the language. Facts attached to signatures are also syntactic sugar. Our core language includes subset (`in`), equality, negation, conjunction and universal quantification *formulae*. The other kinds of formulae, such as existential quantification and disjunction, can be derived from those. Moreover, we consider binary (union (`+`), intersection (`&`), difference (`-`), join (`.`) and product (`->`)) and unary (transpose (`~`) and transitive closure (`^`)) *expressions*.

```
fact ::= fact { (formula)* }
formula ::= expr in expr | expr = expr | not formula |
            formula and formula | (all var: sigName | formula)
expr ::= sigName | relName | var | expr binop expr | unop expr
binop ::= + | & | - | . | ->
unop ::= ~ | ^
sigName, relName, var ::= id
```

Predicate and *function* paragraphs are used to package formulae and expressions, respectively. We do not include them in our core language because they are actually syntactic, as explained elsewhere [Jackson, 2005].

Analysis paragraphs: Alloy has some other constructs for performing analysis, such as *assertions* and *commands* (run and check). Due to its exclusive use for performing analysis [Jackson et al., 2000], they do not affect the meaning of a model. Therefore, we do not include in the core language. So, in our core language, an Alloy model may only contain signatures and facts.

All constraints mentioned before do not reduce the expressiveness of the language. Nevertheless, we have two assumptions in the considered language. Despite Alloy's limited support for integer expressions (addition and subtraction) [Jackson, 2005], we do not consider them. Moreover, we only consider binary relations.

5. Static Semantics

Before proposing a static semantics for Alloy, we must specify Alloy's syntax in PVS, not shown here due to its simplicity. It consists of introducing types, such as for models (`Model`) and signatures (`Signature`), and declaring relations, such as `sigs`, which represents all signatures of a model.

An Alloy model is well-formed (`wf`), if its signatures and relations are well formed (`wfSigRel`), and its formulae are well-typed (`wellTyped`), as stated next in PVS.

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

The `bool` keyword denotes the boolean type. We chose PVS since it has a theorem prover, which will allow us to verify whether a model transformation preserves the static semantics. Hereafter, besides mixing some well-known mathematical symbols with PVS keywords and functions, we make some small changes in PVS fragments in order to improve readability.

There are some constraints that define a well-formed signature or relation in Alloy. Next, we declare all well-defined constraints for our core language. For instance, a signature can only extend another 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, since we are dealing with binary relations, the right side type of a relation must be a signature name declared in the same model (`relType`). Finally, as previously mentioned, we add a constraint stating that we cannot have two relations with the same name (`uniqueRelName`) in a model.

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

5.1. Type System

In this section, we specify in PVS when formulae are well-typed in the core language. A formal type system for object models has been proposed [Edwards et al., 2004]. An extended version of it is used in Alloy, however it is not formalized. Firstly, we show an example that will be used in this section. Figure 3 describes the object model presented in Section 4.

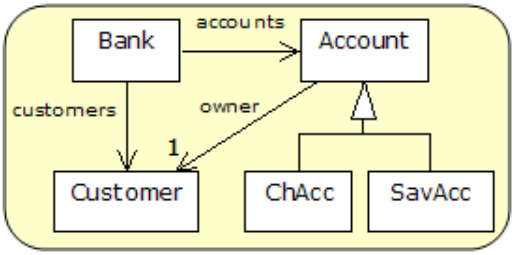

Figure 3: Banking System Application

One kind of type error in Alloy is an *arity error*. It is reported when we attempt to apply an operator to an expression of the wrong arity, or to combine expressions

of incompatible arity. For instance, the union of a signature name and a field, such as `Bank+owner`, is ill-typed since they have different arities. A *disjointness error* happens when two relations in an expression are combined in a way that always results an empty relation. For example, the expression `Bank&Account` yields a disjointness error, since this intersection is always empty.

Well-typed formulae: Next we describe when Alloy's formulae are well-typed. The following PVS recursive function states when a formula is well-typed in a model.

```
wellTyped(f:Formula,
          m:{mod:Model | wfSigRel(mod)},
          vars:set[Variable]): RECURSIVE bool =
```

In PVS, we can declare dependent types [Owre et al., 2005], such as the type of the model `m` in the previous PVS function. It establishes that `wellTyped` is only applicable to arguments for `m`, which has well-formed signatures and relations. The `RECURSIVE` keyword indicates that this function is recursive [Owre et al., 2005]. The universally quantified variables of the formula `f` are represented by `vars`.

An equality formula `e1=e2` is well-typed when both expressions are well-typed and have the same arity. Moreover, the types of `e1` and `e2` are not disjoint, as formalized next.

```
wellTypedExp(e1,m,vars) ∧ wellTypedExp(e2,m,vars) ∧
arity(e1)=arity(e2) ∧
typeExpr(e1,m,vars) ∩ typeExpr(e2,m,vars) ≠ ∅
```

The `typeExpr` relation yields all possible types of an expression. For example, the type of `Account` expression can be itself, `ChAcc` or `SavAcc` types. The constraints for subset formulae are the same for equality formulae. Negation and conjunction formulae are well-typed if their subformulae are well-typed.

A universal quantification formula `∀ x:T | f` is well-typed when the subformula `f` considering the variable `x` is well-typed. In nested quantifications, for simplicity, we do not allow in our core language two variables with the same name, which is possible in Alloy. This does not restrain expressiveness since we can always rename a variable. Finally, the model `m` must declare a signature named `T`.

```
wellTyped(f,m,vars∪{x}) ∧ x ∉ variables(f) ∧
∃ s:sigs(m) | name(s)=T
```

The `sigs` and `variables` relations yield all signatures of a model and all variables used in a formula, respectively.

Well-typed expressions: In Alloy, an expression is well-typed when it does not have arity (`arityWT`) and disjointness (`disjointnessWT`) errors, and all its names are declared in the model (`nameWT`), as described next.

```
wellTypedExp(e:Expression,
             m:{mod:Model | wfSigRel(mod)},
             vars:set[Variable]): bool =
arityWT(e,m,vars) ∧ nameWT(e,m,vars) ∧ disjointnessWT(e,m,vars)
```

Hereafter, for simplicity, we call an expression without arity or disjointness errors as arity and disjointness well-typed expressions, respectively.

Arity errors: For example, a union expression `e1+e2` is arity well-typed if each subexpression is arity well-typed, and both subexpressions (`e1` and `e2`) have the same arity, as formalized next.

```
arityWT(e1,m,vars) ∧ arityWT(e2,m,vars) ∧ arity(e1) = arity(e2)
```

For intersection and difference expressions, the constraints are the same. In case of join expressions, besides both subexpressions being arity well-typed, at least one of them must have an arity different than one. Since we are dealing with binary relations, product's subexpressions must have arity one. The closure and transpose operations are only defined for expressions with arity two.

Disjointness errors: In case of disjointness errors, a difference expression `e1-e2` is ill-typed when `e1` and `e2` have disjoint types, as declared next.

```
typeExpr(e1,m,vars) ∩ typeExpr(e2,m,vars) ≠ ∅ ∧
disjointnessWT(m,vars,e1) ∧ disjointnessWT(m,vars,e2)
```

For instance, the expression `Bank-Account` is ill-typed since the types of bank and account are disjoint. These constraints are similar for intersection expressions. In Alloy, we can make a union of two expressions of different types, such as `Bank+Account`. Therefore, there is no other restriction, except that each subexpression must be disjointness well-typed. The same holds for product and transpose expressions. For join expressions `e1.e2`, at least the right type of `e1` must be a super or subtype of the left type of `e2`. In case of closure expressions `^e1`, at least the left and right types of `e1` must have a subtype in common.

Type of expressions: In order to avoid subtypes comparisons in Alloy's type system, Alloy models are reduced to a canonical form [Edwards et al., 2004]. This canonical form eliminates each type that has a subtype in favour of *atomic types*. The atomic types are a fine-grained set of types that partition the same universe of objects. For every signature `S` that has a subtype, we create a *remainder type* named `$S` containing its direct instances that belong to no subtype. Since each atomic type created is disjoint, we eliminate subtype comparisons in favor of exact matching. For example, the canonical form of our banking system contains a checking, savings and remainder accounts (`$Account`), hence eliminating the account's parent signature. `$Account` contains all accounts that are not checking or savings, hence it is disjoint from checking and savings accounts.

When eliminating subtype comparisons, each expression may have a set of possible types. The following relation declares the types of an expression. Notice that models must have well-formed signatures and relations.

```
typeExpr(e:Expression,
         m:{mod:Model | wfSigRel(mod)},
         vars: set[Variable]): RECURSIVE set[Type] =
```

For instance, in case a signature has a subtype, its type is the set of the subsignature types, united to the type of its remainder type. So, in the previous banking system example, the type of `Account` is the union of `SavAcc`, `ChAcc` and `$Account` types, which is the remainder type. In case a signature, such as `Bank`, does not have a subsignature, its type is the singleton set with its name as unique element. The type of a relation name expression is the Cartesian product of its left and right types. For instance, `owner`'s type is the product of `Account` and `Customer` types.

For the other expressions, this simplification (canonical form) in the type system allows the use of relational operators in type calculation [Edwards et al., 2004]. Thus the type of a union expression `e1+e2` is the union of each subexpression's type, as declared next.

```
typeExpr(e1,m,vars) ∪ typeExpr(e2,m,vars)
```

Another example, the type of a join expression is the join of each subexpression's type. Moreover, the type of a product expression is the product of each subexpression's type. This general idea holds for all expressions in our core language, except for difference

expressions, where the type of `e1-e2` is `e1`'s type, and it is not the difference of each subexpression's type.

5.2. Discussion

Besides arity and disjointness errors, Alloy presents an additional type error: the *irrelevance error*. This error is reported when an expression is redundant, usually within union expressions. For instance, the expression `(ChAcc+Bank)&Account` is ill-typed since `Bank` and `Account` types are disjoint. If we replace `Bank` by the empty relation, the entire expression's meaning is not changed.

However, this typing rule implies that equational reasoning in Alloy may introduce type errors when using Alloy Analyzer [Jackson et al., 2000], which is a tool used to perform analysis on Alloy models. For example, the following model shows part of a banking system stating that all accounts are checking or savings. Moreover, some (checking) account is related to some primary savings accounts. It is important to mention that in Alloy, differently from Java, a parent signature can refer to some of its subsignatures' relations [Jackson, 2005], such as in the last formula of the `BankConstraints` fact.

```
sig Account {}
sig ChAcc extends Account {
  primary: set SavAcc
}
sig SavAcc extends Account {}
fact BankConstraints {
  Account = ChAcc+SavAcc
  some Account.primary
}
```

The `some` keyword, when applied to an expression, denotes that the expression yields at least one element. In the last constraint, replacing `Account` by the union of checking and savings accounts generates the following model, which has the same semantics as the previous one.

```
sig Account {}
sig ChAcc extends Account {
  primary: set SavAcc
}
sig SavAcc extends Account {}
fact BankConstraints {
  Account = ChAcc+SavAcc
  some (ChAcc+SavAcc).primary
}
```

Although `Account` has the same type as `ChAcc+SavAcc`, this transformation introduces an irrelevance error since `SavAcc` and the domain of `primary`, which is `ChAcc`, types are disjoint. This shows that this kind of equational reasoning is unsound considering the Alloy Analyzer. Therefore, we do not consider this kind of error. In our formalization, the resulting constraint is well-typed since the type of `ChAcc+SavAcc` and the left type of `primary` are not disjoint, all names are declared in the model and at least one expression (`primary`) has an arity two in the join. Moreover, both expressions (`ChAcc` and `SavAcc`) have the same arity in the union.

All type errors (arity, disjointness and irrelevance) reported in Alloy Analyzer are sound. However, reports regarding *ambiguous reference* may generate a false alarm [Edwards et al., 2004]. In Alloy, two disjoint signatures can declare relations with

the same name. Ambiguous reference report is not considered a type error in Alloy Analyzer [Jackson, 2005]. Since it is not considered a type error, our core language does not allow two relations with the same name, hence avoiding ambiguous reference reports.

In Alloy it is possible to have `none` expressions, which denote the empty set relation. It has the same type of some ill-typed expressions [Edwards et al., 2004]. For example, considering the formula `Account=none` expressing that there is no account. Although `Account` and `none` have disjoint types, this formula is well-typed in Alloy Analyzer. Therefore, we need some additional cases in some parts of our formalization for dealing with `none` expressions. Since it is a syntactic sugar construct, we do not consider it here, hence making our formalization more uniform. As a consequence, the size of model transformations soundness proofs is decreased.

6. Model Refactorings

In this section we show the importance of the static semantics presented in Section 5, in proposing and proving model transformations for Alloy in PVS. We have proposed a set of primitive laws for Alloy, stating properties about signatures, relations, facts and formulae [Gheyi et al., 2004]. Each primitive law defines two fine-grained structural semantics-preserving model transformations.

Next, we show a law that allows us to introduce a relation and its definition, which is a formula of the form $r = exp$, into a model (applying from left to right); similarly it can also be used to remove a relation from a model (applying from right to left). Each law defines two templates of equivalent models [Gheyi et al., 2005b] on the left and the right side. This law establishes that we can always introduce a relation declared with a fresh name. It also indicates that we can remove a relation that is not being used. We used ps, rs and $forms$ to denote a set of signature and paragraphs, a set of relation declarations and a set of formulae, respectively.

Law 1 ⟨introduce relation and its definition⟩

ps
sig S {
rs
}
fact F {
$forms$
}
	$=_{\Sigma,v}$	ps
		sig S {
		rs,
		r : **set** T
		}
		fact F {
		$forms$
		r=exp
		}

provided
(→) (1) The family of S in ps does not declare any relation named r; (2) T is a signature name declared in ps or is S; (3) $exp \leq r$ in ps and $forms$; ...
(←) r does not appear in ps and $forms$; ...

The $\leq$ operator denotes the subtype relationship. This law can also be applied when the signature S extends a signature. Moreover, since we are focusing on the static semantics, we do not show other provisos required to preserve the dynamic semantics. We write (→), before the condition, to indicate that it is required when applying this law from left to right. Similarly, we use (←) to indicate what is required when applying the law in the opposite direction. Each primitive law, when applied in any direction, defines two semantics-preserving transformations.

When introducing r, its name must not be previously declared in the family of S (all signatures that extend or are extended by it) in order to preserve `uniqueRelName` valid. It is important to mention that `uniqueRelName` is the only property of `wfSigRel` that is different in full Alloy from our core language. This property states that a model cannot have two relations with the same name, which is very restrictive. In full Alloy, two relations of a signature's family cannot have the same name. Sometimes, the conditions of a law for a core language have to be relaxed when considering full Alloy.

Moreover, T must be a signature name declared in the model in order to preserve `relType`. The other constraints of `wfSigRel` are related to signatures. When removing r, all properties are valid. Since r is a new relation or does not appear in the model, except in its definition, all formulae are well-typed. In the definition of r, exp must be a subtype of r. Since their types are not disjoint ($exp \leq r$), the equality formula is well-typed. It is implicitly assumed that exp is well-typed. The expression exp cannot be a super type of r since it may introduce an inconsistency in the model.

Although these transformations are simple and localized, it is important to formally prove that they are sound. After formalizing a static semantics for Alloy, now we are able to state both transformations defined by Law 1 in PVS and verify whether they preserve the static semantics. Similarly, we have to prove that the dynamic semantics is also preserved, as described elsewhere [Gheyi et al., 2005a]. So, each model transformation must transform a well-formed model into another well-formed model.

Next, we show how to prove in PVS that the introduction of a relation preserves the static semantics (applying Law 1 from left to right). First, we describe the syntax of the template models in the transformation, as stated next. We consider that `m1` and `m2` represent the left and right side models of the law, respectively. In the previous law, the two models have the same formulae, except for $r = exp$ (for readability, `g` is the surrogate for $r = exp$). There are two signatures (`s1` and `s2`) named S, one on each side of the law. They are equivalent except that one of them declares a relation r. In general, we map each construction in the law to each corresponding element in the semantics.

```
syntaxIntRel(m1,m2:Model, s1,s2:Signature, r:Relation): bool =
  formulae(m2)= formulae(m1)∪{ g } ∧
  name(s1)=name(s2) ∧ extends(s1)=extends(s2) ∧
  relations(s2) = relations(s1)∪{ r } ∧ sigs(m1)(s1) ...
```

The `formulae`, `extends` and `relations` relations denote the set of formulae of a model, the set of extensions of a signature and the set of relations of a signature, respectively.

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 relation, as established in Law 1, stating that there exists a signature named `T` in `m1`. The `right` function denotes the right type of a relation.

```
condIntRel(m1,m2:Model, s1,s2:Signature, r:Relation): bool =
  ∃ s:sigs(m1) | name(s) = right(r) ...
```

Since we are dealing with fine-grained transformations that have syntactic conditions, the previous two functions are easily specified. Now we are able to state the theorem in order to prove that the introduction of a relation preserves static semantics. So, we have to prove that a transformation takes a well-formed model to another well-formed model, as declared next.

```
staticSemanticsIntRel: THEOREM
  ∀ m1,m2:Model, s1,s2:Signature, r:Relation:
    syntaxIntRel(..) ∧ condIntRel(..) ∧ wf(m1) ⇒ wf(m2)
```

Following a similar approach, we can prove that removing a relation and other model transformations preserves the static semantics. An additional purpose of fine-grained laws is their possible composition, in order to derive model refactorings, such as one for pulling up a relation (applying from left to right), as described next. This refactoring presents one proviso when we push down a relation r, stating that there is no expression using r with a type that is subtype of T but not subtype of S. Since we are decreasing its type, we have to make sure that the transformation does not introduce a type error, such as in the transformation depicted in Figure 1. When pulling up a relation, we have to make sure that it does not introduce name conflicts.

Refactoring 1 ⟨pull up relation⟩

ps
sig T {
rs
}
sig S **extends** T {
rs',
r : **set** U
}
fact F {
$forms$
}
	$=_{\Sigma,v}$	ps
		sig T {
		rs,
		r : **set** U
		}
		sig S **extends** T {
		rs'
		}
		fact F {
		$forms$
		no $(T - S).r$
		}

provided
(→) The family of T in ps does not declare any relation named r;
(←) $E.r$, where $E \leq T$ and $E \not\leq S$, does not appear in ps or $forms$ or any valid item in v.

E denotes any expression. This derived law can also be applied when T extends a signature. Since both transformations have the same signatures and relations, preserving the hierarchy, this law preserves all properties of `wfSigRel`. Considering full Alloy, `uniqueRelName` has to make sure that pulling up a relation does not introduce name conflicts. As the law just changes r's type when we decrease its type, we have a proviso in all formulae containing it. The other formulae are well-typed.

6.1. Discussion

So far, we have proposed a set of 20 primitive laws for Alloy. All of them are proven sound in PVS. Although they define small transformations, we can compose them and derive interesting coarse-grained transformations, such as Refactoring 1. All refactorings derived using these sound laws are also type safe and do not need to be proven in PVS.

Proving some laws in PVS shows us the importance of defining fine-grained transformations. In our opinion, coarse-grained transformations would be far more difficult to prove. The PVS prover helps us by performing several proofs or part of those lemmas automatically. In order to do that, experience with PVS is needed, for deciding when to apply the appropriate proof command.

This proof experience in PVS is very important not only to understand the reasons for enabling conditions, but also for proposing conditions to other model refactorings, which is a difficult task. Proving that a transformation preserves the static semantics increases the knowledge about incorrect transformations. Moreover, during the proofs we detected some problems in our initial static semantics specification. For example, the type of all binary expressions is the type of each expression applied to its binary operator. For

instance, the type of `exp1+exp2` is the union of each subexpression's type. We followed this approach in the difference of two expressions. However, this is not correct. During the proofs in PVS, we realized that the type of `exp1-exp2` is the type of `exp1`.

7. Related Work

Related work proposes a formal dynamic semantics for a core language of Alloy [Edwards et al., 2004, Jackson, 2002]. The authors take into consideration the same expressions and formulae as ours, but, in contrast, they consider signatures as a syntactic sugar construct for sets. Moreover, there is no subtyping in their language. Nevertheless, by using our laws, we can transform our core language into an equivalent language. Moreover, they do not formally state when an Alloy model is well-formed. In one of the approaches [Edwards et al., 2004], authors proposed a type system for object models, which an extended version is used in Alloy. They formalize the type inference rules for object models but they do not show all rules for Alloy.

A related approach defines a formal semantics for a previous version of Alloy [Frias et al., 2004, Frias et al., 2005], where there is no notion of subtyping. A subsignature has the same type of its parent signature. They follow a similar approach of the previous work, considering signatures as syntactic sugar; hence they do not show when an Alloy model is well-formed. Differently from our work, they extend Alloy to include other constructs for expressing dynamic properties, and propose a complete calculus for this extension by translating relational logic to the equational calculus of fork algebras.

Related work [Gogolla and Richters, 1998, Evans, 1998, Sunyé et al., 2001, Lano and Bicarregui, 1998] has been carried out on transformation of UML class diagrams. These approaches do not formally state in which conditions a transformation can be applied. Therefore, some transformations do not preserve semantics and the static semantics in some situations. These transformations do not preserve semantics because some of them use a semi-formal UML semantics. Others partly define a semantics for UML but do not verify soundness of the transformations, or do not consider OCL constraints. None of them prove that each transformation preserves the static semantics.

For example, a model transformation similar to our Refactoring 1 applying from right to left is proposed [Evans, 1998] for UML/OCL without any enabling condition. Moreover, some transformations remove elements, such as classes, disregarding whether some constraints using them are declared in the model. Both transformations may turn the resulting model ill-typed. It is easy to make a small change in a model and make it inconsistent or introduce a type error. Another work [Gogolla and Richters, 1998] proposes some syntactic sugar semantics-preserving transformations for UML/OCL. Since OCL's type system is not formalized, some of them may introduce type errors, as discussed in Section 2.

A related solution [McComb and Smith, 2004] proposes three refactoring rules for Object-Z, for adding, copying and removing structures from a specification. They show how they can be combined in order to formally introduce design patterns. These model transformations are proven to be valid refinements. As a future work, we aim at relating our equivalence notion [Gheyi et al., 2005b] to the traditional notion of data refinement.

A work on program refactorings proposes a set of program refactorings for a subset of sequential Java [Borba et al., 2004]. A set of primitive laws is also defined, and proved that they are behavior preserving based on weakest preconditions semantics [Borba et al., 2004]. However, they did not formally prove that each transformation preserves the static semantics of the language. We believe that our approach can be simi-

larly used for proving their laws.

A closely related approach was developed by Tip et al. [Tip et al., 2003]. They realized that some enabling conditions and modifications to source code for refactorings involving generalization in Java, for automation in Eclipse [Eclipse.org, 2005], depend on relationships between types of variables. These type constraints enable the tool to selectively perform transformations on source code, avoiding type errors that would otherwise prohibit the overall application of the refactoring. They manually proved that these refactorings preserves the type system of the language. In our approach, we can prove that any model transformation preserves static semantics, not only dealing with generalization.

8. Conclusions

In this paper, we formalize a static semantics for Alloy in PVS. Moreover, we use PVS to specify some model transformations for Alloy, and prove them with respect to the static semantics proposed using PVS's theorem prover. Additionally, we show the importance of dealing with fine-grained transformations. This approach can be used to make more reliable and cost effective model refactoring tools.

One of the most difficult tasks for proposing refactorings is to define the enabling conditions. This is the most important part for refactoring tools developers. They rely on these conditions to automate refactorings. We can waste a great amount of time trying to prove something that cannot be accomplished. The experience of proving some laws in PVS shows us that it becomes easier (dealing with fine-grained transformations) to identify which conditions are necessary for a specific transformation.

So far, we have proposed and proved, with respect to the static and dynamic semantics of Alloy, 14 semantics-preserving transformations in PVS, such as laws for introducing a signature, generalization, formula and subsignature. As a future work, we intend to propose more laws, and compose them to derive more refactorings.

Acknowlegments

We would like to thank all anonymous referees, Daniel Jackson, 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., 2004] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52:53–100.

[Eclipse.org, 2005] Eclipse.org (2005). Eclipse project. At http://www.eclipse.org.

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

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

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

[Frias et al., 2005] Frias, M., Galeotti, J., Pombo, C., and Aguirre, N. (2005). Dynalloy: Upgrading alloy with actions. In *27th International Conference on Software Engineering*, pages 442–451. ACM Press.

[Frias et al., 2004] Frias, M., Pombo, C., and Aguirre, N. (2004). An equational calculus for alloy. In *6th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, pages 162–175. Springer-Verlag.

[Gheyi et al., 2004] Gheyi, R., Massoni, T., and Borba, P. (2004). Basic Laws of Object Modeling. In *3rd Specification and Verification of Component-Based Systems, 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. In *20th IEEE/ACM International Conference on Automated Software Engineering*. To appear.

[Gheyi et al., 2005b] Gheyi, R., Massoni, T., and Borba, P. (2005b). An Abstract Equivalence Notion for Object Models. *Elsevier's Electronic Notes in Theoretical Computer Science, Proceedings of Brazilian Symposium on Formal Methods*, 130:3–21.

[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.

[Jackson, 2002] Jackson, D. (2002). Alloy: a lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290.

[Jackson, 2005] Jackson, D. (2005). Alloy 3.0 Reference Manual. At http://alloy.mit.edu.

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

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

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained*. Addison-Wesley.

[Lano and Bicarregui, 1998] Lano, K. and Bicarregui, J. (1998). Semantics and transformations for UML models. In *1st Unified Modeling Language*, pages 97–106.

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

[McComb and Smith, 2004] McComb, T. and Smith, G. (2004). Architectural design in object-z. In *Australian Software Engineering Conference*, pages 77–86.

[Owre et al., 2005] Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (2005). PVS Language Reference. At http://pvs.csl.sri.com.

[Schürr, 2001] Schürr, A. (2001). New type checking rules for OCL expressions. In *Modellierung 2001, Workshop der Gesellschaft für Informatik e. V.*, pages 91–100. GI.

[Sunyé et al., 2001] Sunyé, G., Pollet, D., Traon, Y., and Jézéquel, J.-M. (2001). Refactoring UML models. In *4th Unified Modeling Language*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag.

[Tip et al., 2003] Tip, F., Kiezun, A., and Baumer, D. (2003). Refactoring for Generalization Using Type Constraints. In *18th Object-oriented programing, systems, languages, and applications*, pages 13–26. ACM Press.