Refactoring Alloy Specifications

Rohit Gheyi [1,2] Paulo Borba [3]

Informatics Center
Federal University of Pernambuco
Recife, Brazil

Abstract

This paper proposes modeling laws for Alloy, a formal object-oriented modeling language. These laws are important not only to define the axiomatic semantics of Alloy but also to guide and formalize popular software development practices. In particular, these laws can be used to formaly refactor specifications. As an example, we formally refactor a specification for Java types.

Key words: Formal Methods, Model Transformations, Refactoring, Model Checking

1 Introduction

Laws of programming are important not only to define the axiomatic semantics of programming languages but also to assist the software development process. In fact, besides being used as a foundation for informal development practices like refactorings [6], which is becoming popular due to modern methodologies like Extreme Programming [1], these laws are very helpful to motivate the practical use of formal development methods.

Although they have not been sufficiently studied yet, modeling laws, or laws of design models, might bring similar benefits, but with a greater impact on reliability and productivity, since they are used in earlier stages of the software development process. In order to explore that, this paper introduces and formalizes modeling laws. We also show how those laws can be used to refactor Alloy specifications. In particular, we focus on transformations of project models in Alloy [13], a formal object-oriented modeling language.

[1] We would like to thank the anonymous referees for making several suggestions. Special thanks go to Tiago Massoni and to the other members of the Software Productivity Group for their important comments. This work was supported by CNPq, a Brazilian research agency.

[2] Email: rg@cin.ufpe.br
[3] Email: phmb@cin.ufpe.br

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

We chose Alloy rather than the Unified Modeling Language (UML) [2] and the Object Constraint Language (OCL) [15], because Alloy has a simpler semantics and is not ambiguous, allowing then a consistent formalization. Another advantage of Alloy is the support for automatic specification analysis, which is an important aspect for using formal methods in practice. Furthermore, despite its simple semantics, Alloy is expressive enough to model a great variety of applications. Probably, our laws can also be useful for reasoning about UML models if we give the semantics of UML class diagrams using Alloy.

Besides clarifying aspects of Alloy's semantics, these basic laws serve as a basis for deriving more elaborate laws for practical applications of model transformations, such as introducing design patterns to a model. We will not show here due to the lack of space. Since we proved that the basic laws preserve semantics, we assure that these more elaborate laws derived, using the basic laws, also preserve semantics. The focus of these basic laws is to use them to derive more elaborate laws that restructure models for which we want to assure that the semantics is preserved. However, we may also want to make changes in the specifications that do not preserve semantics. In these cases, the basic laws proposed next are not suitable but this is trivially done by the user. Furthermore, it is a good practice to show that the set of laws that we proposed are complete in some sense. The standard approach is to show that the basic set of laws is sufficient to transform any arbitrary model into a normal form expressed in terms of a small subset of the language operators [9]. This, however, is beyond the scope of this article.

A similar work has been done for Refinement Object Oriented Language (ROOL) [3]. However, ROOL is less powerful for specifying structural properties among types. Whereas ROOL supports only attribute declarations, as in Java [8], Alloy supports the declaration of bidirectional relations with arbitrary arities and multiplicities, as in UML with OCL. Another difference is that in ROOL we cannot define global constraints, such as those involving cardinality (number of instances) of classes in the entire system. In Alloy and UML with OCL this is possible.

Related approaches [16,7,20,5] have proposed some transformations of UML models. However some of the transformations do not completely preserve semantics. Furthermore, they are different from ours since they do not propose a small set of small-grain transformations. They are much similar to design refactorings, which are large-grain transformations that preserve semantics and may improve readability, design structure and reusability of the specification.

Model transformations are useful because it is easier to transform models rather than code. Usually, we only refactor code. Nevertheless, if we build a tool to relate code and model transformations automatically, we only need to make changes in the models which are simpler, and it automatically maps changes to the code.

The remainder of this paper is organized as follows. Section 2 overviews the Alloy language. In Section 3, we present some basic laws for Alloy. The following section illustrates how the laws presented in the previous section can be used to formally refactor a design model for Java types. Finally, Section 5 discusses some related work and presents our conclusions.

2 Alloy

Alloy is a strongly typed language that assumes a universe of elements partitioned into subsets, each of which is associated with a basic type. An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures that are used for defining new types, and formulas paragraphs, like facts and functions, used to record constraints. Each signature has a set of objects (elements). These objects can be related by the relations declared in the signatures.

A signature paragraph introduces a basic type – if it does not extend another signature – and a collection of relations, called *fields*, along with the types of the fields and other constraints on the values they relate. If a signature extends another, then its type is the parent signature type. Besides signature extension, Alloy has other important structures and operators like modules, polymorphism, commands for analyzing the specification, that can be found elsewhere [13].

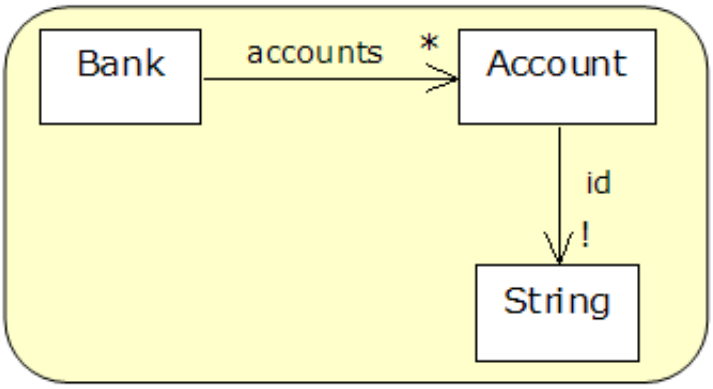

Fig. 1. Bank System Object Model

Suppose that we want to model in Alloy part of a bank system where each bank has a set of accounts and each account has an identifier that is unique. Figure 1 describes the object model [17] of the system. Each box in an object model represents a set of objects. The arrows are relations, sometimes called associations, and they indicate how objects of a set are related to objects in other sets. For instance, the arrow labeled `id` from `Account` to `String` shows that each object from `Account` has a field whose value object is a `String` object.

The multiplicity symbols are: ! (exactly one), ? (zero or one), * (zero or

more) and + (one or more). Multiplicities annotations can appear on both ends of the arrow. If a multiplicity symbol is omitted, * is assumed. The following piece of code introduces the three signatures and two relations that models part of the bank system.

```
sig Bank {
  accounts: set Account
}
sig Account {
  id: String
}
sig String {}
```

In the field declaration of `Bank`, the keyword `set` specifies that `accounts` maps each element in `Bank` to a set of elements in `Account`. When we omit the keyword, as in the declaration of `id`, we specify that all accounts have one identifier (total function).

One of the formula paragraphs is called a fact. It is used to package formulas that always hold, such as invariants about the elements of certain types and sets. The following example introduces a fact named `Restrictions`, establishing general properties about the previously introduced signatures and relations.

```
fact Restrictions {
  Account = Bank.accounts
  all disj acc1,acc2:Account | (acc1.id) != (acc2.id)
}
```

The first formula states that there is no account that is not a bank account and the other one states that different accounts have different identifiers. The keyword `all` is the universal quantifier and we use `disj` before `acc1` and `acc2` to indicate that they are disjoint. In Alloy, the fact formulas are implicit declared as a conjunction of them.

Functions are parameterized formulas that can be applied by binding its parameters to expressions whose types match the declared parameter types. They are especially useful for specifying the behavior of the system operations. The subsequent example declares a function that adds an account to a specific bank and returns a bank with the new account added.

```
fun Bank::addAccount(acc:Account):Bank {
  result.accounts = this.accounts + acc
}
```

The operator + corresponds to the union operator. The keywords `this` and `result` are anonymous variables that contain the receiver and result arguments. The receiver and result arguments represent the pre and post state, respectively, of the `addAccount` function.

The join `p.q` of relations `p` and `q` is the relation you get by taking every combination of an element in `p` and an element in `q`, and including their join,

if it exists. The relations `p` and `q` may have any arity, as long as they are not both unary. The right type of `p` must match the left type of `q`. This is a generalized definition of the standard join operator. For instance, the join of `this.accounts`, where `this` is a bank and `accounts` is a binary relation that relates banks to accounts, returns the set of elements which are the relational image of `this` in `accounts`. This expression yields all accounts of the receiver bank.

The semantics of an Alloy specification is all the possible assignments of values to the signature names and field names that satisfy the implicit and explicit constraints of the specification. In the previous example, it consists of all valid assignments to the signature names `Bank`, `Account` and `String` and to the relation names `accounts` and `id`. These assignments have to satisfy all implicit and explicit constraints of the specification. For example, all elements in all assignments given to `Account` have to be a bank account.

Models in Alloy are concise, analyzable, declarative and structural. None of these features is new in itself. Alloy is different because it is declarative and analyzable at the same time. It has been assumed for a long time that a high-level specification cannot be analyzable and declarative at the same time. Functional programs are both declarative and executable, but they do not have the same level of abstraction of Alloy's specifications, which are more abstract. In Alloy, it is possible to automate different kinds of simulations and analysis [12,10,11,14].

3 Basic Laws

In this section, we present some basic laws proposed for Alloy. These laws state properties about signatures, formulas and relations. The first law establishes that we can always introduce an empty signature declared with a fresh name. It indicates that we can also remove an empty signature that is not used. We used *ps* to denote a set of paragraphs. Since in Alloy we cannot have two paragraphs with the same name then we have a proviso stating that the name of the fresh signature does not appear in *ps*.

Law 1 ⟨introduce empty signature⟩

ps	=	*ps* **sig** *S* `{}`

provided
(→) *ps* does not declare any paragraph named *S*;
(←) *S* does not appear in *ps*.

We write (→) before the proviso to indicate that this proviso is only required for applications of this law from left to right. Similarly, we use (←) to

indicate that it is only for applying the law from right to left, and we use ($\leftrightarrow$) to indicate that the proviso is necessary in both directions. It is important to notice that each basic law, when applied from left to right or right to left, defines one transformation that preserve semantics.

We consider that two models in Alloy are equal if all possible assignments to the names of one model are equivalent to the other model's assignments in respect to all names that are common to the models.

A similar law holds for facts, but as Alloy facts cannot be referred from other paragraphs, the elimination of an empty fact is always possible.

Law 2 ⟨introduce empty fact⟩

ps	=	*ps* **fact** *F* {}

provided
($\rightarrow$) *ps* does not declare any paragraph named *F*.

The next law establishes that we can add or remove a formula to a fact as long as it can be deduced from the other formulas of the specification.

Law 3 ⟨introduce formula⟩

ps
fact *F* {
  forms
}
	=	*ps*
		fact *F* {
		  forms
		  f
		}

provided
($\leftrightarrow$) The formula *f* can be deduced from the formulas in *ps* and *forms*.

Since *f* is derived from other formulas, we guarantee that both specifications have the same meaning. The constraints imposed by this formula are already imposed by the others. We write *forms* to denote a set of formulas. From predicate calculus, we infer `'A and B'` from the `'A => B'` and `A` formulas. Therefore, we can apply this law. These laws presented here focus only on Alloy structures. However, relational operator properties and predicate calculus properties can also be applied to Alloy formulas. Another law indicates that we can always move a formula from one fact to another.

Law 4 ⟨move formula⟩

ps
fact *F* {
  forms
  f
}
fact *G* {
  forms′
}
	=	*ps*
		fact *F* {
		  forms
		}
		fact *G* {
		  forms′
		  f
		}

We can move a formula from one fact to another because in Alloy it does not matter if one formula is declared in one fact or in another since the semantics of Alloy specifications treat them as a conjunction of all fact formulas.

We can think about Laws 1 and 2 as a neutral element. Similarly, Law 3 states the idempotence property of the conjunction logical operator, whereas Law 4 states the associative property of the conjunction logical operator.

Besides those laws for dealing with facts, we also have laws for manipulating relations. The next law states that we can introduce a new relation along with its definition, that is a formula in the form $r = exp$, establishing a value for the relation. We can also remove a relation that is not used.

Law 5 ⟨introduce relation and its definition⟩

ps
sig *S* {
  rs
}
fact *F* {
  forms
}
	=	*ps*
		sig *S* {
		  rs,
		  r : **set** *T*
		}
		fact *F* {
		  forms
		  r=*exp*
		}

provided
(→) The family of S does not declare any relation named r in ps, and T is either S or a signature name declared in ps;
(←) The r relation does not appear in ps and $forms$.

The family of the S signature is the set of all signatures that extend and are extended by S including itself. The precondition stating that the fresh relation name does not appear in the S signature family guarantees that we do not have conflict names. Alloy does not allow two relations with the same name in a family signature.

Notice that the previous law can be used to simply introduce a relation, without any definition. We have just to take *exp* as being r itself, introducing a tautology, which can be trivially eliminated with Law 3.

A relation qualified as a **set** of T, declared in the S signature, indicates that every element in S can be related to any number of T elements. Since it does not impose any constraint in the mapping, we assure that the previous law preserves the implicit and explicit constraints. Therefore, we do not introduce an inconsistency. The type of the r relation can also be qualified as **scalar** or **option** of T. The first qualifier defines a total function and the last one a partial function from S to T, in this case. The default qualifier is **scalar**. Due to the implicit constraint introduced by the **scalar** and **option** qualifiers, we cannot introduce a relation declared with these qualifiers since the implicit constraints can contradict previous implicit and explicit constraints of the specification.

In this context, we can only introduce a relation with the **set** qualifier. Nevertheless, if we derive from the implicit and explicit constraints that this new relation is a total or a partial function then we can apply Law 3 from left to right and introduce this formula. After that, we can change the type of the relation by applying the laws that we show next. Hereafter we use r instead of $S\$r$ because we assume, for simplicity, that there is no other relation with the same name in the specification. The notation $S\$r$ alternatively denotes a reference to the r relation of the S signature.

The following two laws allow us to change the type of the relations from **set** to **scalar** or **option**. The first of them establishes that we can change the type of a relation from **set** to **scalar**, or vice-versa, if r is a total function.

Law 6 ⟨change relation type: from set to scalar⟩

```
ps
sig S {
  rs,
  r : set T
}
fact F {
  forms
  all s : S | one s.r
}
```

=

```
ps
sig S {
  rs,
  r : scalar T
}
fact F {
  forms
}
```

A similar law is proposed to change the type of a relation from **set** to **option** or vice-versa. In this case, we need a formula that establishes that r is a partial function.

Law 7 ⟨change relation type: from set to option⟩

ps
sig S {
rs,
r : **set** T
}
fact F {
$forms$
all $s : S$ | **sole** $s.r$
}
	=	*ps*
		sig S {
		rs,
		r : **option** T
		}
		fact F {
		$forms$
		}

It is important to notice that Laws 6 and 7 are also valid if the S signature extends other signatures. We have proved the soundness of these laws and proposed other laws, such as pull up relation, but these are beyond the scope of this article. We proved the soundness of these laws based on the denotational Alloy's semantics that we propose using Alloy. We prefer to propose small-grain transformations because it is easier to prove that they preserve semantics. Composing them, we can derive large-grain transformations, which consequently preserve semantics. Examples of the use of the laws can be found in the following section.

4 Refactoring Java Types Specification

Maintaining an object-oriented program often requires structural changes, such as changing the relations between classes. This is useful for software evolution, when designers are interested in restructuring the specification for improvement, or to introduce new requirements in a more adequate way.

In this section, we show how we can refactor a simple but non-trivial Alloy specification by simply using the presented laws. This illustrates that the laws can be useful to refactor models. We consider here a specification that models part of a Java type-checker specification. This model was written by Daniel Jackson [4], with a few syntax differences, and is part of the Alloy Analyzer distribution package [5].

4.1 Initial Specification

The specification of Java types describes the basic notions of typing in Java as depicted in Figure 2. An arrow with a closed head form, like from `Object` to `Class`, denotes a subset relationship. In this case, `Object` is a subset of `Class`. If two subsets share an arrow, they are disjoint. For instance, `Class` and `Interface` are disjoint. If the arrowhead is filled, the subsets exhaust the

[4] http://sdg.lcs.mit.edu/~dnj
[5] http://alloy.mit.edu

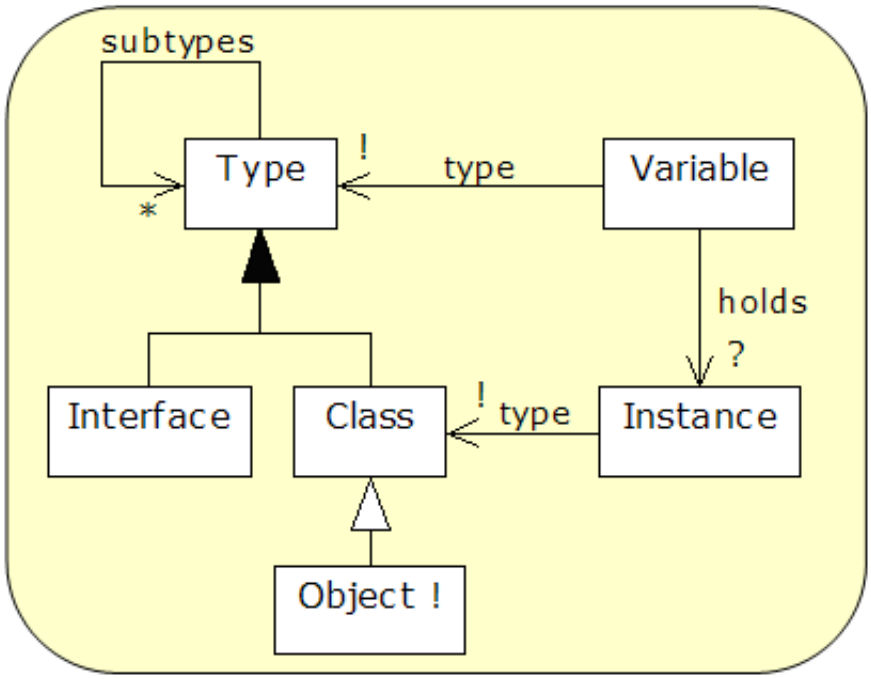

Fig. 2. Object Model of Daniel Jackson's Java Types

superset, so there are no members of the superset that are not members of one of the subsets. In this case, the subsets form a *partition*: every member of the superset belongs to exactly one subset.

The ! notation in the `Object` box indicates that the set this box represents contains one element. This model ignores primitive types and `null` references. Each `Type` may have a set of subtypes. Types are partitioned into `Class` and `Interface` types and `Object` is a particular class. The following Alloy's fragment describes the object model in Figure 2.

```
sig Type {
  subtypes: set Type
}
sig Class, Interface extends Type {}
```

In Alloy, we can declare several signatures at once if they do not declare any relation, as showed in the declaration of `Class` and `Interface`. One signature can extend another one establishing that the extended signature is a subset of the parent signature. This does not introduce another basic type, as mentioned before. The extended signature has the same type of the parent signature.

```
static sig Object extends Class {}
```

The `static` keyword denotes that the signature has exactly one element. In this example, the `Object` signature has only one element.

The following fact declaration states that every type is subtype of `Object` but is not a subtype of itself. Additionally, it specifies that every type is a

subtype of at most one class. It also declares that there exists no type that is both class and interface and any type is exclusively a class or an interface. The operators $*$, $\hat{}$ and $\tilde{}$ correspond, respectively, to reflexive transitive closure, transitive closure and transpose of a relation.

```
fact TypeHierarchy {
  Type in (Object.*subtypes)
  no t: Type | t in (t.^subtypes)
  all t: Type | sole ((t.~subtypes) & Class)
  no (Class & Interface)
  Type = (Class + Interface)
}
```

The operator & corresponds to the intersection operator. The keyword `no` is a quantifier in the second formula. But, when applied to an expression, as in the fourth formula, defines a predicate that there is no element in the expression. The keyword `in` in Alloy can denote a subset operator, like in the first formula of the previous fact, and a membership operator as in the second formula declared before. The keyword `sole`, when applied to an expression, defines a predicate. For instance, `sole` (`t. subtypes` & `Class`) is true if the expression (`t. subtypes` & `Class`) has at most one element.

So far, we have specified types in Java. Now we describe part of a simple type checker of some expressions, for simplicity. The following signature declarations express that every object has a type (its creation type) that is a class. A variable may hold an instance, and has a declared type.

```
sig Instance {
  type: Class
}
sig Variable {
  holds: option Instance,
  type: Type
}
```

Finally, the `TypeSoundness` fact states that all instances held by a variable have types that are direct or indirect subtypes of the variable's declared type.

```
fact TypeSoundness {
  all v: Variable | (v.holds.type) in (v.type.*subtypes)
}
```

4.2 Refactored Specification

The previous model describes Java types in terms of the `subtypes` relation. However, we want to refactor this model to describe Java types in terms of supertype relations such as `extends`[6] and `implements`. In order to guaran-

[6] It is important to observe that no relation in Alloy can be named `extends` because it is a keyword. Nevertheless, we use it here only to improve readability.

tee that the resulting model is equivalent to the original one, we should not proceed in an ad hoc way, making arbitrary changes to the model. In fact, we can use Laws 2, 3, 5 and 7 to change the specification in a step-by-step way, as illustrated next, preserving its semantics. The desired object model is described in Figure 3.

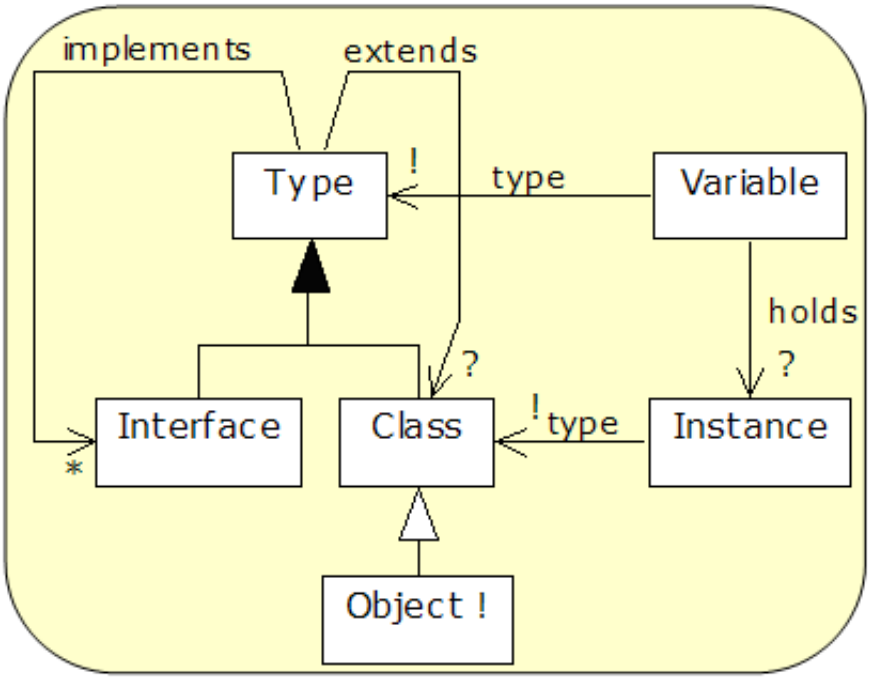

Fig. 3. Object Model of Java Types Refactored

To simplify the following refactoring steps, we consider that `ps` contains the signatures and facts declared before, except the `Type` signature. At the end of the section, we show the complete specification. In order to introduce the new relations, we can use Law 5, but as this law requires a fact, we first apply Law 2 from left to right to introduce an empty fact.

Next, we introduce, using Law 5 from left to right, the relations `extends` and `implements`, in `Type`, and their definitions in the `Definitions` fact.

```
ps
sig Type {
  subtypes: set Type,
  extends: set Class,
  implements: set Interface
}
fact Definitions {
  implements = (~subtypes & (Type->Interface))
  extends = (~subtypes & (Type->Class))
}
```

The notation $->$ represents the product operator that combines every element in the left operand with every element in the right operand and con-

catenates them. These elements can also be a tuple of elements. When applied to sets, this operator represents the standard Cartesian product. Notice that we can apply the law because no `extends` and `implements` relations are declared in the `Type`'s family. Observe also that the types of the new relations are valid in the initial specification.

Our aim is to derive the `subtypes` definition in order to replace it by its definition and eventually remove it from the specification. To derive its definition, notice that `extends` and `implements` have the same type (`Type -> Type`). We use some predicate calculus and set theory properties [19] and some formulas in the specification, within brackets, to justify every step in the derivation. Since they have the same type, we can derive that their union is:

```
(extends + implements =
  (~subtypes & (Type->Class)) + (~subtypes & (Type->Interface)))
  [A&B + A&C = A&(B+C)]
<=> (extends + implements =
    ~subtypes & ((Type->Class) + (Type->Interface)))
  [(A->B) + (A->C) = (A->(B+C))]
<=> (extends + implements =
    ~subtypes & (Type->(Class + Interface)))
  [Type = (Class+Interface)]
<=> (extends + implements = ~subtypes & (Type->Type))
<=> (extends + implements = ~subtypes)  [(A=B) => (~A=~B)]
<=> (~(extends + implements) = ~~subtypes)  [~(A+B) = (~A+~B)]
<=> (~extends + ~implements = ~~subtypes)  [~~A = A]
<=> (~extends + ~implements = subtypes)  [(A=B) => (B=A)]
<=> (subtypes = ~extends + ~implements)
```

Since this formula is deduced from `Definitions`, using Law 3 from left to right, we can introduce this formula in the specification.

```
ps
sig Type {
  subtypes: set Type,
  extends: set Class,
  implements: set Interface
}
fact Definitions {
  implements = (~subtypes & (Type->Interface))
  extends = (~subtypes & (Type->Class))
  subtypes = (~extends + ~implements)
}
```

From the third formula of the `TypeHierarchy` fact, we can deduce, as explained next, `all t : Type | sole (t.extends)`.

```
all t: Type | sole ((t.~subtypes) & Class)
  [subtypes = ~extends + ~implements]
<=> all t: Type | sole ((t.~(~extends + ~implements)) & Class)
```

```
  [~(A+B) = ~A + ~B]
<=> all t: Type | sole ((t.(~~extends + ~~implements)) & Class)
  [r.(A+B) = r.A + r.B]
<=> all t: Type | sole ((t.~~extends + t.~~implements) & Class)
  [~~A = A]
<=> all t: Type | sole ((t.extends + t.implements) & Class)
  [(A+B)&C = A&C + B&C]
<=> all t: Type | sole (((t.extends) & Class) +
      ((t.implements) & Class))  [no (Class & Interface)]
<=> all t: Type | sole (((t.extends) & Class) + {})  [A + {} = A]
<=> all t: Type | sole ((t.extends) & Class)
<=> all t: Type | sole (t.extends)
```

Then we can apply Law 3 from left to right introducing in the `Definitions` fact the `all t : Type | sole (t.extends)` formula since we derive a formula stating that `extends` is a partial function.

```
ps
sig Type {
  subtypes: set Type,
  extends: set Class,
  implements: set Interface
}
fact Definitions {
  implements = (~subtypes & (Type->Interface))
  extends = (~subtypes & (Type->Class))
  subtypes = (~extends + ~implements)
  all t:Type | sole (t.extends)
}
```

Now we can change the type of the `extends` relation from `set` to `option` applying Law 7 from left to right.

```
ps
sig Type {
  subtypes: set Type,
  extends: option Class,
  implements: set Interface
}
fact Definitions {
  implements = (~subtypes & (Type->Interface))
  extends = (~subtypes & (Type->Class))
  subtypes = (~extends + ~implements)
}
```

Next, we replace the references to the `subtypes` relation by its definition. Notice that from every formula, except its definition, containing the `subtypes` relation, we can derive a formula replacing the `subtypes` relation by its definition and insert it to the specification applying Law 3 from left to right. Consequently, these new formulas can also derive the formulas with `subtypes`. Next,

we can remove all formulas that contain `subtypes` from the specification applying Law 3 from right to left. The `ps(subtypes = ~extends+~implements)` notation denotes that we replace every occurrence of `subtypes` in `ps` by `~extends+~implements`.

```
ps (subtypes = ~extends + ~implements)
sig Type {
  subtypes: set Type,
  extends: option Class,
  implements: set Interface
}
fact Definitions {
  implements = (~(~extends + ~implements) & (Type->Interface))
  extends = (~(~extends + ~implements) & (Type->Class))
  subtypes = (~extends + ~implements)
}
```

The resulting first and second formulas of `Definitions` are trivially valid formulas. We can deduce using some set theory properties and formulas in the specification, that they are equivalent to `implements` = `implements` and `extends` = `extends`, respectively. Therefore, we can remove them applying Law 3 from right to left.

```
ps (subtypes = ~extends + ~implements)
sig Type {
  subtypes: set Type,
  extends: option Class,
  implements: set Interface
}
fact Definitions {
  subtypes = (~extends + ~implements)
}
```

Since `subtypes` does not appear in `ps`, because we replaced it by its definition, we can remove this relation and its definition using Law 5 from right to left. After that, the `Definitions` fact becomes empty and can be removed applying Law 2 from right to left. This is the last step for replacing the `subtypes` relation by the `extends` and `implements` relations. The resulting specification is shown next:

```
sig Type {
  extends: option Class,
  implements: set Interface
}
part sig Class, Interface extends Type {}

static sig Object extends Class {}
```

```
fact TypeHierarchy {
  Type in (Object.*(~extends + ~implements))
  no t: Type | t in (t.^(~extends + ~implements))
  all t: Type | sole ((t.~(~extends + ~implements)) & Class)
  no (Class & Interface)
  Type = (Class + Interface)
}

sig Instance {
  type: Class
}
sig Variable {
  holds: option Instance,
  type: Type
}

fact TypeSoundness {
  all v: Variable |
    (v.holds.type) in (v.type.*(~extends + ~implements))
}
```

Now the specification models the Java types in terms of supertype relation. However, the resulting model present some formulas that are not so readable, like some formulas in the `TypeHierarchy` fact. We can replace them by other formulas. For instance, we can replace the third formula of this fact by `all t : Type | sole (t.extends)` using some set theory and predicate calculus properties and formulas of the specification. In fact, we can even remove this formula. First, applying Law 7 from right to left derives the same formula. Next, using Law 3 from right to left we can remove one of them. Finally, applying Law 7 from left to right, result a specification without the `all t : Type | sole (t.extends)` formula.

We have refactored other specifications and have derived other refactorings such as introducing a generalization and moving a relation. However, due to the lack of space, we will not show them here.

5 Conclusions

In this paper we propose basic laws for Alloy. These laws are important not only to define the axiomatic semantics of the modeling language but also to refactor specifications, as illustrated for the Java type's model.

The laws presented here have been proved to be sound with respect to a formal semantics for Alloy. Consequently, they should serve as tools for carrying out model transformations. One immediate application of the basic laws is to define an interface from which one can derive more elaborate laws such as refactorings. As illustrated in Section 4, they can also be used to

refactor designs in a step-by-step way. We do not need to prove that the initial and final Java types specifications have the same semantics because the basic laws, that we used, guarantees that each one preserve semantics. Therefore, the composition of them also preserves semantics.

The main focus of the basic laws is the architects. The architects will use them to derive refactorings (large-grain transformations) that will be available for the designers. The designers will use only the refactorings derived by the architects. We have deduced some refactorings and apply three of them to a bank system's specification. In this case, the designer only needs to apply three refactorings rather than refactoring the bank system's specification in around twenty steps applying the basic laws, as we did in the Java type's specification. Formalizing further basic laws, like one stating in which conditions we can introduce a generalization, refactorings, and applying them to other realistic case studies are some topics for further research.

Most of the basic laws are very simple to apply since their pre-conditions are simple syntactic conditions. However, some laws like the one to introduce a formula (Law 3) might require, in order to avoid errors and to help the architect, the use of an automated theorem prover or a proof assistant to verify if two formulas are equivalent, for example. Alloy can also be used in these cases. However, we can only guarantee in the Alloy Analyzer, that the assertions are valid for a pre-defined scope. We extended the Alloy Analyzer tool to include the implementation of most of the basic laws presented here, where the user does not need to verify the pre-conditions and apply them. The user only requires informing the parameter values of the transformations. Another approach that we can use to automate the implementation of them is to use the Maude [18] term rewriting system.

Although we already have a set of basic laws for Alloy, some of them we show here, we still need to prove a reduction theorem stating that our set of laws is complete in the sense of allowing reduction of arbitrary Alloy specifications to a normal form expressed in a small subset of the language operators following approaches adopted for ROOL [4], for imperative languages [9], among others. Another further research topic is to check the relation between modeling laws and programming laws; for instance, the laws of ROOL. In particular, we need to investigate if the design refactorings have corresponding code refactorings.

Related work [20,5,7,16] has been carried out on transformation of UML design models. However, some transformations [20,7,16] do not preserve semantics. For instance, it is argued that creating a generalization between classes preserve semantics. However, the constraints in the specification can become inconsistent by introducing a generalization (Figure 4). For instance, we cannot extend the `S` class with the `T` class when we have a explicit constraint in the specification stating that `S` has more elements than `T`. The introduction of a generalization in this case will make the specification inconsistent since the generalization introduces an implicit constraint that `T` contains `S`. Therefore,

we deduce, from this implicit constraint, that `T` has the same elements or more elements than `S`, which contradicts the explicit constraint in the specification.

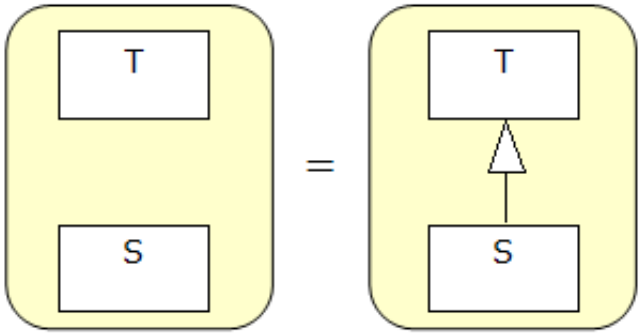

Fig. 4. Introduce Generalization

In addition, another work [16] states that logically strengthening a class invariant is a refinement transformation, although adding a formula can sometimes turn the specification inconsistent. Another work [5] removes classes and relations without checking if there are some constraints about them in the specification, which is not acceptable in a formal setting. These transformations do not preserve semantics because some of them use an UML semantics that is ambiguous. Others define part of UML semantics but do not verify the transformations soundness or do not consider OCL constraints.

References

[1] Beck, K., "Extreme Programming Explained," Addison-Wesley, 2000.

[2] Booch, G., I. Jacobson and J. Rumbaugh, "The Unified Modelling Language User Guide," Addison-Wesley, 1999.

[3] Borba, P. and A. Sampaio, *Basic laws of ROOL: an object-oriented language*, Revista Brasileira de Informática Teórica e Aplicada **VII** (2000 - A shorter version appeared in *Third Brazilian Workshop on Formal Methods*, pages 33–44, João Pessoa, Brazil, October 2000.), pp. 49–68.

[4] Borba, P., A. Sampaio and M. Cornélio, *A refinement algebra for object-oriented programming*, in: *17th European Conference on Object-Oriented Programming, ECOOP'03*, Darmstadt, Germany, 2003, pp. 457–482.

[5] Evans, A., *Reasoning with UML class diagrams*, in: *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA* (1998), pp. 102–113.

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

[7] Gogolla, M. and M. Richters, *Equivalence rules for UML class diagrams*, in: *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, 1998, pp. 87–96.

[8] Gosling, J., B. Joy and G. Steele, "The Java Language Specification," Addison-Wesley, 1996.

[9] Hoare, C., J. Spivey, I. Hayes, J. He, C. Morgan, A. W. Roscoe, J. Sanders, I. Sorenson and B. Sufrin, *Laws of programming*, Communications of the ACM **30** (1987), pp. 672–686.

[10] Jackson, D., *Object models as heap invariants*, in: *Essays on Programming Methodology*, Springer-Verlag, 2000 .

[11] Jackson, D. and A. Fekete, *Lightweight analysis of object interactions*, in: *Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, 2001.

[12] Jackson, D., I. Schechter and I. Shlyakhter, *Alcoa: the alloy constraint analyzer*, in: *International Conference on Software Engineering*, Limerick, Ireland, 2000, pp. 730–733.

[13] Jackson, D., I. Shlyakhter and M. Sridharan, *A micromodularity mechanism*, in: *ACM SIGSOFT Conference on Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC'01)*, Vienna, Austria, 2001, pp. 62–73.

[14] Jackson, D. and K. Sullivan, *COM revisited: Tool assisted modelling and analysis of software structures*, in: *ACM SIGSOFT Conference on Foundations of Software Engineering*, San Diego, USA, 2000, pp. 149–158.

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

[16] Lano, K. and J. Bicarregui, *Semantics and transformations for UML models*, in: *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, 1998, pp. 97–106.

[17] Liskov, B. and J. Guttag, "Program Development in Java," Addison-Wesley, 2001.

[18] Meseguer, J., *A logical theory of concurrent objects and its realization in the maude language*, in: *Object Oriented Programming*, MIT Press, 1993 pp. 314–390.

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

[20] Sunyé, G., D. Pollet, Y. Traon and J.-M. Jézéquel, *Refactoring UML models*, in: *The Unified Modeling Language, UML'01 - Modeling Languages, Concepts, and Tools. Fourth International Conference, Toronto, Canada, October 2001, Proceedings*, LNCS **2185** (2001), pp. 134–148.