Basic Laws of Object Modeling

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
Semantics-preserving model transformations are usually proposed in an *ad hoc* way because it is difficult to prove that they are sound with respect to a formal semantics. So, simple mistakes lead to incorrect transformations that might, for example, introduce inconsistencies to a model. In order to avoid that, we propose a set of simple modeling laws (which can be seen as bi-directional transformations) that can be used to safely derive more complex semantics-preserving transformations, such as refactorings which are useful, for example, to introduce design patterns into a model. Our laws are specific for Alloy, a formal object-oriented modeling language, but they can be leveraged to other object modeling notations. We illustrate their applicability by formally refactoring Alloy models with subtypes in order to improve the analysis performed by the Alloy Analyzer tool.

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

General Terms
Design, Verification

Keywords
Model Transformations, Refactorings, Formal Methods, Verification

1. INTRODUCTION
Laws of programming [14] are important not only to define the axiomatic semantics of programming languages but also to assist in the software development process. In fact, these laws can be used as the foundation for informal development practices, such as refactorings [7], widely adopted due to modern methodologies, in particular Extreme Programming [1].

Modeling laws might bring similar benefits, such as refactoring models, but with a greater impact on cost and productivity, since they are used in earlier stages of the software development process. However, semantics-preserving model transformations are usually proposed in an *ad hoc* way because it is difficult to prove that they are sound with respect to a formal semantics. So, simple mistakes lead to incorrect transformations that might, for example, introduce inconsistencies to a model.

In this paper, we propose a set of modeling laws [9] for Alloy [15], a formal object-oriented modeling language. Each law proposed here defines two small-grained model transformations that preserve semantics. We proved their soundness based on a denotational semantics for Alloy [9]. We regard them as basic because they are simple and able to derive more complex transformations. This set can be considered comprehensive, if compared to what have been proposed so far [6, 12, 18]. In addition, we propose an equivalence notion for Alloy models.

These laws can be useful to refactor Alloy models with subtypes in order to improve the analysis performed by the Alloy Analyzer tool. Moreover, these laws can be applied to derive refactorings [9], which are useful, for instance to introduce design patterns [8] into a model. Additionally, they can be applied to reason whether one component, annotated with Alloy, can be reused or substituted by another. Our laws can also be used for educational purposes in object modeling, since they clarify the meaning of a number of important constructs.

Related work [6, 18, 12] has proposed transformations for Unified Modeling Language (UML) [2] class diagrams. However, they do not offer a comprehensive set of modeling laws that can derive more complex transformations. In addition, some of them do not completely preserve semantics. We proposed laws for Alloy, rather than UML and the Object Constraint Language (OCL) [17], due to Alloy's simpler semantics, although expressive enough to model a broad variety of applications. Nevertheless, our laws can also be useful for reasoning about UML class diagrams, by provid-

[*]Supported by CNPq.
[†]Supported by CAPES.
[‡]Partially supported by CNPq (521994/96-9).

ing a semantics for UML class diagrams based on Alloy [20]. Similarly, the results can also be leveraged to other object modeling notations.

The remainder of this paper is organized as follows. Section 2 overviews the Alloy language. Section 3 presents some basic laws for Alloy. In Section 4, we show applications for the laws. The following section discusses some related work. Finally, Section 6 concludes the paper.

2. ALLOY

Alloy is formal object-oriented specification language that is strongly typed and assumes a universe of elements partitioned into subsets, each of which associated with a basic type. Alloy can be used for specifying, verifying and validating properties about object and component-based systems.

An Alloy model or specification is a sequence of *paragraphs* of two kinds: signatures that are used for defining new types, and formula paragraphs, namely facts and functions, used to record constraints. Each signature contains a set of objects (elements). These objects can be related by the relations declared in the signatures. A signature paragraph introduces a basic type and a collection of relations, called fields, along with the types of the fields and other constraints on the values they relate. Besides subtyping with signature extension, Alloy includes other important structures and operators such as modules, predicates and commands for analyzing the specification, which are discussed elsewhere [15].

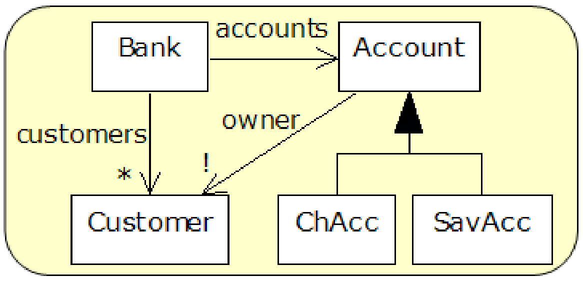

Figure 1: Bank System Object Model

Suppose that we want to model part of a banking system in Alloy, on which each bank contains a set of accounts and a set of customers. Each account is owned by a customer. Also, accounts may be checking or savings. Figure 1 describes the object model [19] of the system. Each box in an object model represents a set of objects. The arrows are relations and indicate how objects of a set are related to objects in other sets. For instance, the arrow labeled `owner` shows that each object from `Account` has a field whose value object is a `Customer` object.

An arrow with a closed head form, such as from `ChAcc` to `Account`, denotes a subset relationship. In this case, `ChAcc` is a subset of `Account`. If two subsets share an arrow, they are disjoint. For instance, `ChAcc` and `SavAcc` are disjoint. If the arrowhead is filled, the subsets exhaust the superset, so there are no members of the superset that are not members of one of the subsets. In the banking system, the subsets form a *partition*: every member of the superset belongs to exactly one subset.

The multiplicity symbols are as follows: `!` (exactly one), `?` (zero or one), `*` (zero or more) and `+` (one or more). Multiplicity symbols can appear on both ends of the arrow. If a multiplicity symbol is omitted, `*` is assumed. The following fragment introduces three signatures and three relations modeling part of the banking system.

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

In the field declaration of `Bank`, the `set` relation qualifier specifies that `accounts` maps each element in `Bank` to a set of elements in `Account`. When we omit the keyword, we specify a total function.

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` is a subset of the values given to `Account`.

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

Signature extension introduces a subtype in Alloy version 3, establishing that each subsignature is disjoint. In this case, `ChAcc` and `SavAcc` are disjoint. In Alloy, we can declare several signatures at once if they do not declare any relation, as showed in the previous fragment.

Facts are formula paragraphs. They are used to package 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.

```
fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
}
```

The first formula states that each account is owned by exactly one customer (the . operator can be seen as relational dereference), while the last one states that each account is a checking or savings account. The keyword `all` represents the universal quantifier. The `one` keyword, when applied to an expression, denotes that the expression has exactly one element. The operator `+` corresponds to the union set operator. In Alloy, the fact formulae are implicitly declared as a conjunction.

3. BASIC LAWS

In this section, we present a set of basic laws proposed for Alloy. These laws state properties about signatures, relations, facts and formulae. With a comprehensive set of simple basic laws [9], we aim to provide powerful guidance on the derivation of complex transformations, such as refactorings and optimizations. The models on the left and the right side of each law have the same meaning, since each law preserves semantics, as described elsewhere [9].

For instance, the laws can be useful to transform the model shown in Figure 2, which shows a transformation introducing a collection between `Bank` and `Account`. We establish that these models have the same semantics considering a set of signature and relation names that we call alphabet (Σ). The alphabet includes the element names which are considered to be relevant in a model. For instance, suppose that Σ contains the `Bank`, `Account` and `accounts` names. If these models have the same set of values to all names in the alphabet, they are equivalent under this equivalence notion. The other names (`col`, `Vector` and `elems`) are auxiliary, thus not taken into consideration.

However, some models may not have all names considered in the alphabet. For instance, in Figure 2, `accounts` does not belong to the right-hand side model. In this case, relevant names must be represented by others names. Hence, we define a view (v), consisting of a set of items such as $n \rightarrow exp$, where n is a name and exp is an expression not containing n, and they have the same type. In the example of Figure 2, we can choose a v containing the following item: $accounts \rightarrow col.elems$. Now we can compare both models. Notice that `accounts` is defined in terms of two names that belong to the right-hand side model; hence we can compute the `accounts`'s value. So, a view allows a strategy for representing relevant names using an equivalent combination of other elements. There are some constraints when choosing a view, such as the items cannot be recursive. This is a general idea of the equivalence notion considered for our basic laws and it is described in more detail elsewhere [11].

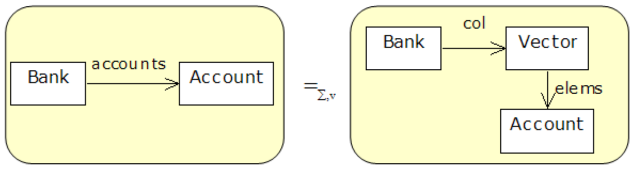

Figure 2: Equivalence Notion

In the proposed laws, we used ps to denote a set of signature and fact paragraphs. We do not consider other Alloy paragraphs since some of them are syntactic and others are used to perform analysis in the Alloy Analyzer.

3.1 Laws for Signatures

The first law states that we can introduce a generalization between two signatures when the name of the new parent signature is not previously used in the specification. We can also remove a generalization between them if the parent signature, the relations of its family and the subsignatures are not being used elsewhere. This proviso guarantees that there is no formula containing S, T and their relations, except the two formulae stating the partition. Since S and T have different types after the generalization removal, this proviso assures that the generalization removal does not introduce type errors.

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 then v contains the $U \rightarrow S + T$ item;
($\rightarrow$) ps does not declare any paragraph named U;
($\leftarrow$) U and the relations declared by its family, S and T do not appear in ps, rs, rs' and $forms$.

We write ($\rightarrow$), before the proviso, to indicate that this proviso 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 proviso is necessary in both directions. It is important to notice that each basic law, when applied in any direction, defines one transformation that preserves semantics.

Notice that both models of the law have the same names and constraints, except U, its definition and the implicit constraints stating that S and T are subset of U. Since the view has an item for U that is equivalent to its definition and it contains the union of the values given to S and T, the left side model yields the same value for U of right side model; hence the law preserves semantics.

The operator `&` denotes the intersection set operator. The keyword `no` when applied to an expression denotes that the expression has no elements. We write $forms$ and rs to denote a set of formulae and a set of relation declarations, respectively. We also propose trivial laws allowing us to introduce empty signatures [9].

3.2 Laws for Relations

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

Law 2. ⟨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
(↔) if r belongs to Σ, r does not appear in *exp* and v contains the $r{\rightarrow}exp$ item;
(→) The family of S in *ps* does not declare any relation named r; T is a signature name declared that does not extend other signatures;
(←) The r relation does not appear in *ps* and *forms*.

The *exp* expression can be either r or an expression having the same type of r and not containing r. It is important to stress 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. Moreover, constraints involving Σ and v must be carefully introduced. When introducing or removing a relation in Σ, we must guarantee that the $r{\rightarrow}exp$ item belongs to v and r does not appear in *exp* in order to avoid a recursive definition in v. The family of a signature is the set of all signatures that extend or are extended by it direct or indirectly. Alloy does not allow two relations with the same name in the same family.

A relation qualified as a `set` of T, declared in the S signature, indicates that every element in S relates to any number of T elements. Since it does not impose any constraint on the relation, we ensure that the previous law preserves the constraints, not introducing inconsistency. In contrast, due to its constraint, we cannot always introduce a relation declared with the `one` (stating a total function) qualifier since this constraint can contradict previous specification constraints. For instance, the introduction of the r relation with `one` in the previous law can introduce an inconsistency if there are constraints stating that T is empty and S has at least one element. After applying this transformation, r must relate every element of S to one element of T. However, S is not empty and T is empty, introducing an inconsistency.

Notice that both models of the law have the same names and constraints, except r and its definition. Since the view has an item for r that is equivalent to its definition, the left side model yields the same value for r of right side model; hence the law preserves semantics.

Next, we establish a law for pulling up relations. We can pull up a relation from a signature to its parent by adding a formula stating that this relation only maps elements of the subsignature. Similarly, we can push down a relation if the specification has a formula stating that the relation only relates elements of the subsignature.

Law 3. ⟨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
}
```

The operator `-` corresponds to the difference set operator. Notice that the values given to r, which is pulled up or down, are the only values that are subject to change. On the left side of the law, r relates elements from S to U. On the right side of the law, r relates elements from T to U. However, there is an explicit constraint indicating that r relates elements from S to U. Therefore, both modes have the same meaning. We have proposed other laws for relations, such as splitting a relation [9].

3.3 Laws for Facts and Formulae

Besides proposing some trivial laws for facts and formulae, we proposed a law establishing that we can add or remove a formula from a fact, as long as it can be deduced from other formulae in the specification.

Law 4. ⟨introduce formula⟩

```
ps
fact F {
  forms
}
```

$=_{\Sigma,v}$

```
ps
fact F {
  forms
  f
}
```

provided

(↔) The formula f can be deduced from the formulae in *ps* and *forms*.

Since f is derived from other formulae, we guarantee that both specifications have the same meaning. The constraints imposed by this formula are already imposed by the others. From predicate calculus, we infer `'P and Q'` from the `'P => Q'` and `'P'` formulae, where `P` and `Q` are arbitrary predicates. Therefore, this law is trivially valid. The laws presented here focus on Alloy structures, although relational [24] and predicate [22] calculi can also be applied to Alloy formulae.

Besides these laws, we proposed laws for syntactic sugar constructs [9]. We prove these laws using a denotational semantics for Alloy [9]. We aim at proposing simple small-grained transformations because it is easier to prove that they are semantics-preserving. Although they are simple, we can derive a number of complex large-grained transformations by composing them, which consequently preserve semantics. Examples of the use of the laws can be found in Section 4.

4. APPLICATIONS

The basic laws provide an axiomatic semantics for Alloy, clarifying the meaning of its constructs. In this section, we describe how we can use them to transform Alloy models with subtypes in order to improve the analysis performance by the Alloy Analyzer. As previously illustrated [5], analysis performance of Alloy models with subtypes can be increased by *atomization*. When performing analysis, the Alloy Analyzer internally transforms (atomizes) a model by pushing relations down to the lowest subtype level in order to improve its performance. Atomization applies a number of model transformations to remove a relation in a parent signature and introduce one relation to each subsignature. However, some transformations are not proved to be semantics-preserving [5], such as introducing and removing relations. Other transformations, such as deducing formulae, are semantics-preserving considering they are derived from relational calculus laws. Next, we show how the proposed laws, which are sound with respect to an Alloy denotational semantics, can formalize the atomization scheme.

Consider an addition to the signature `Account`, described in Section 2, which is partitioned by `ChAcc` and `SavAcc`. Each account relates to a customer by the `owner` relation. In order to improve the performance analysis, the atomization scheme removes `owner` from `Account` while introducing relations in `ChAcc` and `SavAcc`, called `chOwner` and `savOwner`, respectively.

Suppose that we consider an alphabet Σ containing all names declared in the initial specification, and a view v having the $owner \rightarrow chOwner + savOwner$ item. First of all, we can introduce the new relations into `Account` and their definitions by applying Law 2 from left to right. Since both relations do not belong to Σ, no item is needed to v. Next, we show the resulting specification. We consider that `ps` contains the `Bank` and `Customer` signatures.

```
ps
sig Account {
  owner: set Customer,
  chOwner: set Customer,
  savOwner: set Customer
}
sig ChAcc extends Account {}
sig SavAcc extends Account {}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
}
```

The notation `->` represents the product operator that combines every element in the left operand with every element in the right operand. When applied to sets, this operator represents the standard Cartesian product.

Our aim is to pull down `chOwner` and `savOwner` to `ChAccount` and `SavAccount`, respectively. In order to do that, we first derive the `no (Account - ChAcc).chOwner` formula, by applying some relational calculus properties, from the `chOwner` definition. Similarly, we can derive a formula for `savOwner`, and introduce both formulae, by applying Law 4 from left to right, which results in the following specification.

```
ps
sig Account {
  owner: set Customer,
  chOwner: set Customer,
  savOwner: set Customer
}
sig ChAcc extends Account {}
sig SavAcc extends Account {}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
  no (Account - ChAcc).chOwner
  no (Account - SavAcc).savOwner
}
```

Next we apply Law 3 from right to left and pull down both relations, as shown next.

```
ps
sig Account {
  owner: set Customer
}
sig ChAcc extends Account {
  chOwner: set Customer
}
sig SavAcc extends Account {
  savOwner: set Customer
}

fact BankConstraints {
  all acc: Account | one acc.owner
  Account = ChAcc + SavAcc
  chOwner = owner & (ChAcc->Customer)
  savOwner = owner & (SavAcc->Customer)
}
```

Our aim is to derive the `owner` definition in order to replace it by its definition and eventually remove it from the specification. Applying some predicate and relational calculus properties, which are within brackets to justify every step in the derivation, we deduce that:

```
(chOwner + savOwner =
  owner & (ChAcc->Customer) +
  owner & (SavAcc->Customer))
  [(P&Q + P&R) = (P&(Q+R))] =
(chOwner + savOwner =
  owner & ((ChAcc->Customer) + (SavAcc->Customer)))
  [(P->R) + (Q->R) = (P+Q)->R)] =
(chOwner + savOwner =
  owner & ((ChAcc+SavAcc)->Customer))
  [Account = ChAcc + SavAcc] =
(chOwner + savOwner = owner & (Account->Customer)) =
(owner = chOwner + savOwner)
```

Since this formula is deduced from formulae in the specification, using Law 4 from left to right, we can introduce this formula in the specification. After that, we can replace `owner` by its definition. It is important to notice that from every formula containing `owner`, except its definition, we can derive a new formula replacing `owner` by its definition, which is inserted to the specification applying Law 4 from left to right. Consequently, these new formulae can also derive the formulae with `owner`. Next, we can remove all formulae that contain `owner`, except its definition, from the specification by applying Law 4 from right to left.

Finally, since `owner` does not appear in the model, except in its definition, we can remove this relation and its definition using Law 2 from right to left. Since `owner` belongs to Σ, we have to check whether v has the $owner \rightarrow chOwner + savOwner$ item. Moreover, the third and fourth formula of `BankConstraints` can be deduced from the model. Therefore, we can remove them from the specification applying Law 4 from right to left. The final specification is described next.

```
ps
sig Account {}
sig ChAcc extends Account {
  chOwner: set Customer
}
sig SavAcc extends Account {
  savOwner: set Customer
}

fact BankConstraints {
  all acc: Account | one acc.(chOwner + savOwner)
  Account = ChAcc + SavAcc
}
```

Notice that our laws deal with equivalent models; hence the atomization process can be reversed, similarly. This process can be generalized and we can state the atomization semantics-preserving transformation similarly to the laws. Figure 3 summarizes the order of the application of the laws. Each box has the direction and number of the law to be applied. The filled arrow denotes the next law to be applied. Some boxes have filled arrows on top of it indicating that this law can be applied repeatedly.

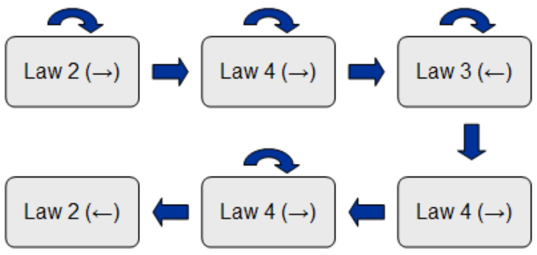

Figure 3: Atomization

In the banking system, every account is a checking or savings account. The atomization process also considers parent signatures that are not partitioned by the subsignatures. In this case, we have to create a new subsignature, extending the parent signature. We do not have a law that allows us to introduce a subsignature in a parent signature already declared. We regard this law as future work.

The basic laws proposed here can also be useful to refactor models [10, 9]. For instance, we refactored a simple but non-trivial Java types specification. Applying the laws, a model describing Java types in terms of subtyping relations can be transformed into another in terms of supertypes, having the same semantics.

We can also use our laws to derive complex large-grained transformations (refactorings) by composing them [9]. Since these refactorings are derived using semantics-preserving laws, they also preserve semantics. We have derived large-grained transformations such as the extract interface refactoring, introducing a collection, and move and reverse a relation [9]. These refactorings can be useful, for example, to introduce design patterns [8] into a model. However, we will not show these here due to the lack of space. Furthermore, by using the laws, we can verify whether two models have the same meaning.

5. RELATED WORK

Zaremski and Wing [25] determines whether two software components are related by a specification matching process. This can be useful, among other things, to reuse components and substitute one component by another without affecting the observable behaviour. To verify whether the specification of one component matches the other, the authors use a theorem prover. We believe that our set of laws can be useful in this case. Suppose that each Java component is annotated with Alloy, similarly as described elsewhere [16]. Since we already proved that our laws are semantics-preserving [9], applying the laws, we just have to check syntactically whether one specification component is equivalent to another, instead of proving it. However, since we do not prove that our set of laws is complete, we may not use them always.

Related work [23, 6, 12, 18] has been carried out on transformation of UML class diagrams. They do not state in which conditions a transformation can be applied. Therefore, some transformations do not preserve semantics in some situations. For instance, creating a generalization between classes not always preserve semantics (Figure 4). Given the constraints in a specification, it can become inconsistent by introducing a generalization. For instance, we cannot declare the `S` class to extend the `T` class when a explicit constraint in the specification states that `S` has more elements than `T`. The introduction of a generalization in this case makes the specification inconsistent, since the generalization constrains `T` to include `S`. Therefore, we can deduce that `T` has the same number or more elements than `S`, which contradicts the explicit constraint in the specification. We cannot apply Law 1 to introduce a generalization in Figure 4, since `T` is already declared.

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. We conclude that it is important to prove the soundness of the transformations, in order to guarantee that a transformation preserves semantics. It is easy to make a small change in a model and make it inconsistent.

A similar work proposes basic laws for Refinement Object-

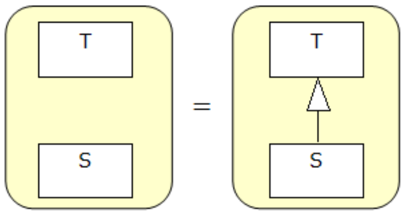

Figure 4: Introduce Generalization

Oriented Language (ROOL) [3]. ROOL is less powerful for specifying structural properties among types compared to Alloy. Whereas ROOL supports only attribute declarations, as in Java [13], Alloy supports the declaration of bidirectional relations with arbitrary arities and multiplicities, as in UML. Another difference is that we cannot define global constraints in ROOL. This related work is similar to ours in the sense that they propose basic laws that are used not only for giving the axiomatic semantics of the language, but also for deriving refactorings.

Laws for top level design elements of UML-RT (Real Time) [4] have also been proposed [21]. Our laws do not deal with refinements, as theirs. Moreover, their work does not intend to propose basic laws, as ours. They propose laws not only for structural constructs, as our laws, but also laws for behavioural constructs, such as laws for capsules. They assume that relationships are directed and predicates involve only relationships as attributes, as in ROOL. Additionally, the authors consider implementation-oriented models. Moreover, their proposed laws rely on the absence of global constraints on the model, such as those involving cardinality (number of instances) of classes in the entire system. Our laws also work for models containing global constraints.

6. CONCLUSIONS

In this paper, we propose basic laws for Alloy and show how they can be used for deriving complex transformations, such as refactorings and optimizations. In contrast to model transformations usually defined in an *ad hoc* way, these laws describe semantics-preserving transformations. Additionally, we propose an equivalence notion for Alloy models.

The laws presented here have been proven sound with respect to a formal semantics for Alloy [9]. Consequently, they should act as a tool for carrying out model transformations. One immediate application of the basic laws is to define an interface from which one can derive more complex transformations, as illustrated in Section 4, and to refactor specifications [10]. Although our laws are specific to Alloy, they can be leveraged to object modeling in general. For instance, we can leverage them to UML class diagrams giving a precise semantics for it in Alloy [20].

All basic laws are very simple to apply since their preconditions are simple syntactic conditions. Nevertheless, these laws can be used as powerful guidance for deriving complex transformations. The law for introducing a formula that is deduced from the model also has syntactic conditions, if we consider relational and predicate calculi. We extended the Alloy Analyzer tool to include the implementation of a number of the basic laws, in such a way that the user does not need to verify the preconditions and apply the laws [9]. The user is only required to inform the parameter values for the transformations. Furthermore, our laws can be used for educational purposes in object modeling, since they clarify the meaning of a number of important constructs. Additionally, they could be useful to verify, with syntactic conditions, whether the specification of one component is equivalent to another. In case they are equivalent, we can substitute a component by another.

Although we have a comprehensive set, relative to what have been proposed so far, of basic laws for Alloy, 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. We need more laws such as for introducing an empty subsignature. This normal form is expressed in a small subset of the language operators, following approaches adopted for ROOL [3] and imperative languages [14], among others. We also intend to study and formalize the relationship between modeling and programming laws. In particular, we need to investigate whether model refactorings have corresponding program refactorings. This might be useful for implementing tools that apply model and code refactorings in a synchronized way.

7. ACKNOWLEDGMENTS

This work benefited from the discussions during the Software Productivity Group's meetings. In addition, we would like to thank all the anonymous referees, whose appropriate comments helped improving the paper. This work was partially funded by CAPES and CNPq, grant 521994/96-9.

8. REFERENCES

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

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

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

[4] B. Douglass. *Real Time UML - Developing Eficient Objects for Embedded Systems.* Addison-Wesley, 1998.

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

[6] A. Evans. Reasoning with UML class diagrams. In *Second IEEE Workshop on Industrial Strength Formal Specification Techniques, WIFT'98, Boca Raton/FL, USA*, pages 102–113. IEEE CS Press, 1998.

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

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

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

[10] R. Gheyi and P. Borba. Refactoring alloy specifications. In A. Cavalcanti and P. Machado, editors, *Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods*, volume 95, pages 227–243. Elsevier, 2004.

[11] R. Gheyi, T. Massoni, and P. Borba. An equivalence notion of object models. Technical Report, 2004.

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

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

[14] C. Hoare, J. Spivey, I. Hayes, J. He, C. Morgan, A. Roscoe, J. Sanders, I. Sorenson, and B. Sufrin. Laws of programming. *Communications of the Association for Computing Machinery*, 30(8):672–686, 1987.

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

[16] S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–245. ACM Press, 2002.

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

[18] K. Lano 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*, pages 97–106, 1998.

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

[20] T. Massoni, R. Gheyi, and P. Borba. A UML class diagram analyzer. In *Third International Workshop on Critical Systems Development with UML (CSDUML), affiliated with UML Conference*, Lisbon, Portugal, 2004.

[21] A. Sampaio, A. Mota, and R. Ramos. Class and capsule refinement in UML for real time. In A. Cavalcanti and P. Machado, editors, *Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods*, volume 95, pages 23–51. Elsevier, 2004.

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

[23] G. Sunyé, 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*, volume 2185 of *LNCS*, pages 134–148. Springer-Verlag, 2001.

[24] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(9):73–89, 1941.

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