Formally Introducing Alloy Idioms

Rohit Gheyi[1], **Tiago Massoni**[1,2], **Paulo Borba**[1]

[1]Informatics Center – Federal University of Pernambuco

[2]Department of Computing Systems – Pernambuco State University
Recife – Brazil

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

Abstract. *Design patterns describe simple and elegant solutions to specific problems in software design. Usually designers have to perform ad hoc transformations in order to introduce them. Most of the time they do not want to introduce or change between patterns that result in a design with conflicting semantics. This is difficult to guarantee by performing ad-hoc transformations. In this paper, we use our catalog of semantics-preserving transformations for Alloy — proven to be sound in a theorem prover — to derive refactorings that formally introduce idioms (design patterns for Alloy). For instance, we derive a refactoring that allows us to change between idioms that declare a state locally or globally by using our catalog. Moreover, our catalog is useful for reasoning about models. For example, two models of a hotel room locking using two different idioms (operations are specified implicitly and explicitly) are stated to have the same state transition. However, we show that they are not equivalent by using our catalog.*

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. One way to evolve a program is by introducing design patterns [Gamma et al. 1994], which describe simple and elegant solutions to specific problems in software design. However, most of the time designers have to perform ad hoc transformations in order to introduce them. This is an error-prone activity. Usually designers do not want to introduce or change between patterns that result in a design with conflicting semantics. This is difficult to guarantee when performing ad-hoc transformations. In order to avoid that, program refactorings [Kerievsky 2004], which improve programs while maintaining their original behavior, for introducing a number of patterns have been proposed.

These refactorings focus on programs instead of models. To our knowledge, there is no adequate support for introducing design patterns in object models. An *object model refactoring*, which is applied to an object model instead of a program, is a transformation that improves model structure while preserving semantics. It is difficult to prove that refactorings are sound with respect to a formal semantics. Defining all conditions required for a transformation to be semantics-preserving is not an easy task. Even a number of object model transformations proposed in the literature, which are intended to be semantics-preserving, may lead to models with type errors or subtle semantic changes in some situations [Gheyi 2007]. In order to avoid that, in our prior work [Gheyi et al. 2004], we

proposed *laws*, which are bi-directional primitive semantics-preserving transformations, for Alloy [Jackson 2006], a formal object-oriented modeling language briefly discussed in Section 3.

In this paper, we show how our laws can be composed to derive object model refactorings that formally introduce Alloy idioms, which are design patterns [Gamma et al. 1994] proposed elsewhere [Jackson 2006]. For instance, we can declare a state using the global or local idioms. We propose a general transformation that allows us to change between them. Moreover, our catalog is useful for reasoning about models. For example, two models describing a hotel room locking using two different idioms, which specify operations implicitly and explicitly, are stated to have the same state transition [Jackson 2006]. Furthermore, one of the models uses an unusual style of frame condition. However, they do not have the same state transition because the unusual style of frame condition is not specified correctly. By using our laws we found that one of the models is *underconstrained*. Therefore, the contributions of this paper are the following:

- use our catalog to refactor the hotel room locking models (Section 4.3);
- show the importance of our laws for reasoning about models (Section 4.4);
- propose refactorings for introducing Alloy idioms (Section 5).

The transformations of our catalog are proven sound in the Prototype Verification System (PVS) [Owre et al. 2007], which encompasses a formal specification language and a theorem prover. Our previous work [Gheyi et al. 2005b] encodes a type system and semantics for Alloy in PVS, and shows in more details how we proved one of our laws in PVS. In this work, we focus on a novel application of our laws. By composing them, we show how they are useful for changing between design patterns. Since refactorings for idioms are derived from sound laws, they are also sound.

2. Alloy

An Alloy model declares a set of *paragraphs*: signatures that are used for defining new types, and constraints, such as facts. Each ***signature*** denotes a set of objects (similar to an UML class), which are associated to other objects by relations declared in the signatures. A signature paragraph introduces a type and a collection of ***relations*** along with their types and other constraints on their included values.

Next, we model in Alloy part of a hotel locking system. Each room has a set of possible keys (`keys`) that can open it and current key (`currKey`) at a given time. The `currKey` relation is a ternary relation. Relations with arity greater than two can use the **`one`** and **`lone`** qualifiers denoting total and partial functions, respectively. Moreover, a guest may hold a set of keys at a given time. Finally, the front desk, which is a singleton (**`one`**), records the last key (`lastKey`) that was issued for a room at a given time, and the occupants of each room at a given time. This example is used in Section 4.

```
sig Room {
  keys:  set Key,
  currKey:  keys one->Time
}
sig Guest {
  keys:  Key->Time
```

Table 1. Meaning of Alloy Operators and Keywords

Operator	**.**	**->**	**none**	**!**	**+**	**&**	**-**	**in**	**=>**	**all**	**some**
Meaning	join	product	$\emptyset$	$\neg$	$\cup$	$\cap$	$\setminus$	$\subseteq$	$\Rightarrow$	$\forall$	$\exists$

```
}
one sig FrontDesk {
  lastKey:  (Room->lone Key)->Time,
  occup:  (Room->Guest)->Time
}
sig Key, Time {}
```

Predicates (**pred**) are used to package reusable formulae. In the initial state, no guest has a key and the hotel (front desk) has no occupant. Moreover, the last key of the front desk and the current key of all rooms are synchronized, as declared next.

```
pred init[t:Time] {
  Guest.keys.t = none
  FrontDesk.occup.t = none
  all r:Room | r.(FrontDesk.lastkey.t) = r.currKey.t
}
```

The **all** and **none** keywords represent the universal quantifier and the empty set, respectively. The dot operator `.` is a generalized definition of the standard relational join operator. The join of `r.keys`, where `r` is a room and `keys` is a binary relation that relates the room to keys, yields all keys that can open a room. Table 1 summarizes the meaning of a number of Alloy operators and keywords used in this paper.

3. Laws

Next a set of sound semantics-preserving transformations for Alloy are presented [Gheyi et al. 2004]. First we define our refinement notion for object models.

3.1. A Refinement Notion for Object Models

Figure 1 depicts two object models of a banking system. Figure 1(a) shows a model stating that each bank is related directly to a set of accounts, whereas the model in Figure 1(b) establishes that each bank is related to a collection, which is directly related to a set of accounts.

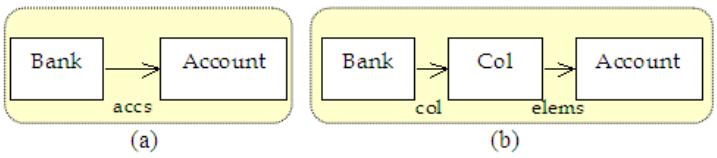


Figure 1. Part of Two Models of a Banking System

In our refinement notion, if a concrete object model has a subset of the instances of the abstract one, the concrete model refines the abstract one. Moreover, our approach compares the semantics of two object models only for a number of relevant model elements, abstracting away the values assigned to the others. The set of relevant elements names is called *alphabet* (Σ). The names that are not in the alphabet are auxiliar, or not relevant for the comparison. For instance, suppose that Σ contains only the `Bank` and `Account` names in Figure 1. Other names, such as `elems`, are regarded as auxiliary.

Sometimes we might have model elements that, although relevant, cannot be compared, since they are not part of both models. For instance, suppose that we include `accs` to Σ. In this case, we cannot compare the models in Figure 1, since `accs` is not part of the model in Figure 1(b). However, it can actually be expressed as the composition of `col` and `elems`. In those cases, our notion has a *view* (v), establishing how an element of one model can be interpreted using elements of another model. Views consist of a set of items such as $n \rightarrow exp$, where n is an element's name and exp is an expression. In Figure 1, we may choose a view containing the following item: $accs \rightarrow col.elems$. Now we can infer that the right-hand side (RHS) model refines the left-hand side (LHS) model.

Adding formulae to a model is a refinement in our notion [Gheyi 2007] (**Refinement 1**). More details about properties and the formalization of this notion in PVS can be found elsewhere [Gheyi et al. 2005a].

3.2. Modeling Laws for Alloy

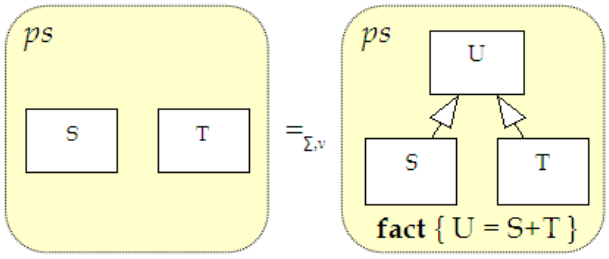
Next we present primitive laws for Alloy. Each law defines two fine-grained semantics-preserving model transformations. We state that two models are *equivalent* when there is a bidirectional refinement. They are primitive in the sense that they cannot be derived from other transformations.

Each law consists of two templates (patterns) of equivalent Alloy models, on the left-hand (LHS) and right-hand (RHS) sides. We can apply a law whenever the template is matched by an Alloy model. A matching is an assignment of all variables occurring in LHS/RHS models to concrete values. Each law may declare some meta-variables. We used ps to denote a set of signatures and facts, and $forms$ to denote a set of formulae. We write ($\rightarrow$), before a condition, to indicate that this condition is required when applying a law from left to right. Similarly, we use ($\leftarrow$) to indicate what is required when applying a law in the opposite direction, and we use ($\leftrightarrow$) to indicate that a condition is necessary in both directions.

Next, we show a law that allows us to introduce a generalization into a model (applying from left to right); similarly it can also be used to remove a generalization from a model (applying from right to left). This law establishes that we can always introduce a generalization declared with a fresh name. Since in Alloy a module cannot have two paragraphs with the same name, we have a condition stating that the new parent signature name does not appear in the module. It also indicates that we can remove a parent signature that is not being used.

S and T may declare relations, which are declared in ps. Applying Law 1 from left to right introduces a fact stating that U is an ***abstract signature*** indicating that all its elements belong to exactly one of its child signatures (subsignatures). We can propose a similar law for introducing a generalization between more than two signatures. Besides

Law 1 ⟨introduce generalization⟩

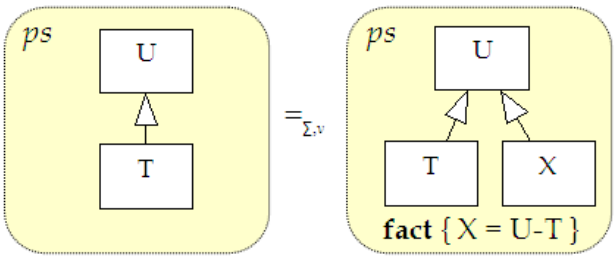


(↔) (1) if U belongs to Σ, v contains the $U \rightarrow S + T$ item; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) ps does not declare any paragraph named U;
(←) U does not appear in ps.

conditions for ensuring that the well-formedness rules of Alloy models are preserved, this law presents other conditions for semantics preservation. For instance, if U belongs to Σ then v must have the $U \rightarrow S + T$ item. In this way, both models have the same values for U. There is another condition applied to all names in the alphabet that are not in the resulting model, ensuring that v represent them unambiguously.

Law 2 allows us to introduce an empty subsignature declared with a fresh name. It makes U an abstract signature. A subsignature can be removed if it is not used elsewhere, and there is no expression in the model with its type, in order to avoid type errors. The exp variable denotes an expression. The $\leq$ operators denotes the subtype relationship. For instance, if X is direct or indirect subsignature of Y then X is a subtype of Y ($X \leq Y$). In order to preserve semantics, there is a condition when X belongs to Σ (v must have the item $X \rightarrow (U - T)$). We can propose a similar law when there are more than one subsignature extending U.

Law 2 ⟨introduce subsignature⟩



(↔) (1) if X belongs to Σ, v contains the $X \rightarrow (U - T)$ item; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) (1) ps does not declare any paragraph named X; (2) there is no signature in ps that extends U;
(←) (1) X does not appear in ps; (2) (2) there does not exist an expression exp, such that $exp \leq U$ and $exp \not\leq S$, in ps, $forms$ or v.

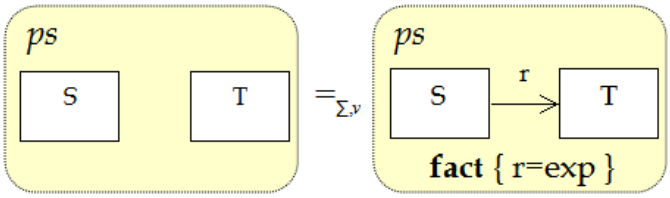
Besides laws for dealing with signatures, we also define laws for manipulating relations. Law 3 states that we can introduce a new binary relation along with its defini-

Table 2. Summary of Laws for Alloy

Law	Name	Law	Name
1	Introduce generalization	10	Introduce signature fact
2	Introduce subsignature	11	Separate signature declarations
3	Introduce relation	12	Remove relation qualifier
4	Introduce a deducible formula	13	Separate relation declarations
5	Remove abstract qualifier	14	Remove relation's type expression
6	Introduce empty signature	15	Introduce predicate
7	Introduce empty fact	16	Replace predicate
8	Split relation	17	Introduce one relation
9	Remove signature qualifier	18	Remove formula from signature fact

tion, which is a formula of the form $r = exp$. We can also remove a relation that is not being used. Law 3 can also be applied when S extends a signature. 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 similar law allows us to introduce a relation with an arity greater than two.

Law 3 ⟨introduce relation⟩



(↔) (1) if r belongs to Σ, r does not appear in exp and v contains the $r{\rightarrow}exp$ item; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
(→) (1) S's family in ps does not declare any relation named r; (2) r does not appear in exp, or exp is r; (3) $exp{\leq}r$ in the resulting model;
(←) r does not appear in ps.

We propose a similar law (Law 17) that allows us to *introduce one relation* (a total function). The conditions are the same except that the new relation cannot belong to the alphabet, and if the set representing the relation's domain is not empty, then the set representing the image must be non-empty. All primitive laws [Gheyi 2007] are summarized in Table 2.

4. Hotel Room Locking

Nowadays, a number of hotels issue disposable room keys using recodable locks. Suppose in the example presented in Section 2 that a new guest `g` is going to stay in the room `r`, whose current key is $key1$. The person at the front desk generates a new key $key2$, which is the successor of $key1$, and gives to `g`. Each room can be opened by its current key or

its successor. It is important to mention that the front desk and all room locks have the same algorithm generating the next key (successor). When g enters the room r for the first time, since g has a $key2$, which is the successor of the lock's current key ($key1$), the lock replaces its own current key with $key2$ and opens the door. The next time g enters the room, the lock finds that its current key is equal to the g's key ($key2$) and opens the door without recoding itself. The card is never modified by the lock. Later on, a new guest with a new key (successor of $key2$) enters into r, which automatically recodes the lock, and the previous guest g cannot enter anymore.

The dynamic behavior of the hotel looking room system is described by three operations: *check in*, *entry* of a guest into a room and *check out*. We may specify those operations in Alloy using *two idioms*. In the first idiom, operations are represented by *predicates* (Section 4.1), and the second idiom specifies operations using *signatures* (Section 4.2). The specification used as an example in Section 2 is common for both idioms. We show how our laws can be used to formally refactor from one idiom to another (Section 4.3). Jackson [Jackson 2006] specifies similar models and declares that they have the same state transition. However, we show that they are not equivalent by using our laws (Section 4.4). Our laws show that one of the models from Jackson's book is underconstrained.

4.1. Implicit Operations

One way to specify operations in Alloy is to use *predicates*. This idiom specifies an operation implicitly since there is no value assigned to the operation. The Alloy semantics does not consider predicate names. It only gives values to signature and relation names. We have to reason in order to infer the performed operation based on the values given before and after the state transition accomplished by the operation.

Next we declare a predicate representing the entry operation. This predicate receives t and t' denoting the pre and post state, and the guest g, which wants to enter the room r with the key k. In order to enter the room, the guest's key must be one of the possible room's key (line 1). If it is not the first time that the guest enters the room, the current room key remains unchanged and the guest's key must be the same room's current key (line 2). However, if it is the first time, then the lock generates a new key (nextKey) and checks whether it is equal to the guest's key (line 3). If the guest's key is equals to the successor of the room's current key, then the lock is recoded.

```
pred entry[t,t':Time, g:Guest, r:Room, k:Key] {
1. k in g.keys.t
   let ck=r.currKey |
2.    (k=ck.t and ck.t'=ck.t) or
3.    (k=nextKey[ck.t,r.keys] and ck.t'=k)
```

In Alloy, we can specify the frame conditions in different ways. Next we specify the traditional style stating that nothing (current key and occupants) changes at the front desk (lines 4 and 5), and the current key of all rooms but r does not change (line 6). In the second idiom, we present a different way to specify frame conditions. Finally, the set of keys held by the guests does not change (line 7).

```
4. FrontDesk.lastKey.t=FrontDesk.lastKey.t'
5. FrontDesk.occup.t=FrontDesk.occup.t'
6. all rm:Room-r | rm.currKey.t=rm.currKey.t'
7. all gu:Guest | gu.keys.t=gu.keys.t'
}
```

The check in and check out operations are specified similarly. An Alloy ***fact*** (**fact**) packages formulae that always hold, such as invariants about the elements. The following fact states all possible traces that can be derived using those operations. After the initialization, the time transition can only be performed by three operations: `checkin`, `entry` and `checkout`. The values given to `Time` are ordered. We can specify this constraint by importing the `ordering` Alloy library. Therefore, we can apply the `first`, `next` and `last` functions. For instance, `first` yields the initial state.

```
fact TracesPred {
  init[first[]]
  all t:Time-last[] | let t'=next[t] |
    some g:Guest, r:Room, k:Key |
      entry[t,t',g,r,k] or checkin[t,t',g,r,k] or
      checkout[t,t',k]
}
```

4.2. Explicit Operations

Alloy does not have a fixed idiom for modeling state machines. We can also specify the previous hotel locking model using *signatures* representing operations instead of predicates. This idiom is called explicit because, when performing analysis on the Alloy Analyzer [Jackson et al. 2000], we can visualize the operations performed based on the values given to the signature representing operations. Representing operations by signatures produces nicer visualizations, and allows some properties to be written more succinctly and directly [Jackson 2006]. In this idiom, there are subtypes of *event* (`Event`) for each operation. In Alloy, one signature can extend another, establishing that the extended signature (***subsignature***) is a subset of the parent signature. For instance, `RoomKeyEvent` extends `Event`. `Event` is an abstract signature.

```
abstract sig Event {
  pre, post:  one Time,
  g:  one Guest
}
abstract sig RoomKeyEvent extends Event {
  r:  one Room,
  k:  one Key
}
```

Arguments of each predicate in the implicit idiom are mapped to relations in a signature. Another advantage of this idiom is that the signature hierarchy can be used to factor out common arguments. For instance, all three operations have a time before (`pre`) and after (`post`) the operation, and a guest (`g`) in common. Moreover, since the `checkin` and `checkout` operations have the key (`k`) and room (`r`) arguments in common, a subsignature (`RoomKeyEvent`) of `Event` is specified declaring those relations.

Next we specify the subsignature `RoomKeyEvent` representing the entry operation. In Alloy, we can declare formulae attached to a signature instead of declaring in facts, as declared next. The only difference is that all formulae declared in a signature are implicitly universally quantified by all its elements. Notice that this signature has the same constraints of the `entry` predicate except for the frame conditions. This idiom uses a more unusual way of specifying frame conditions based on Ray Reiter idea [Reiter 1991]. The frame conditions are specified in the fact (`TracesEvent`) specifying all possible traces that can be derived from the events.

```
sig Entry extends RoomKeyEvent {} {
  k in g.keys.pre
  let ck=r.currKey |
    (k=ck.pre and ck.post=ck.pre) or
    (k=nextKey[ck.pre,r.keys] and ck.post=k)
  (Room-r).currKey.pre = (Room-r).currKey.post
}
```

The `Checkin` and `Checkout` signatures extend `RoomKeyEvent` and `Event`, respectively. They are specified similarly. They also specify all constraints of the corresponding predicates but the frame conditions. Next we specify in a fact all possible traces that can be derived from the initialization using those events. Notice that this fact is specified differently from `TracesPred`. All time transitions are related by at least one event or by skip, in which no variable in the state changes. Some event happens when the values given to a field change. For example, if the values given to `currKey` in a state transition changes, then the entry event happens. It is important to mention that both idioms have the same initialization.

```
fact TracesEvent {
  init[first[]]
  all t:Time-last[] | let t'=next[t] |
    some e:Event {
      e.pre=t and e.post=t'
      currKey.t!=currKey.t' => e in Entry
      occup.t!=occup.t' => e in Checkin+Checkout
      (lastKey.t!=lastKey.t' or keys.t!=keys.t') =>
        e in Checkin
    }
}
```

4.3. Refactoring between Implicit and Explicit Operations Idioms

Next we use our laws in order to refactor from the idiom, which represents operations using predicates, into another, which uses signatures to represent operations. We consider an alphabet containing all signature and relation names declared in Section 2. All names introduced in the following steps are auxiliary. Moreover, since all important names belong to both models, the view is empty. Therefore, all conditions related to the alphabet and view are satisfied. Figure 2 depicts the major steps.

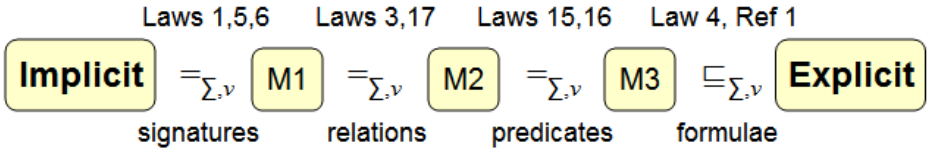


Figure 2. Refactoring between the Implicit and Explicit Operation Idioms

First of all, we apply three times Law 6 from left to right to introduce an empty signature (`Checkin`, `Checkout`, `Entry`) for each operation. After that we introduce a generalization (`RoomKeyEvent`) between `Checkin` and `Entry` by applying Law 1 from left to right. Finally by applying Law 1 from left to right, we add a generalization (`Event`) between `Checkout` and `RoomKeyEvent`. All previous steps can be applied because they do not introduce name conflicts. By applying Law 5 from right to left, we can make `Event` and `RoomKeyEvent` abstract by using formulae introduced by Law 1.

We have to add a relation for each operation's argument. In the hotel locking room, we should introduce five auxiliary relations (`pre`, `post`, `g`, `r` and `k`). This step can be performed by applying Law 17 from left to right, which allows us to introduce a one-relation (function). Since there is no constraint stating that `Event` is empty, we can always apply this law.

Now, by applying Law 16, we replace all predicates by their definition. They only appear in the `TracesPred` fact. So, we can inline all predicates. Since all predicates do not appear in the specification except in its declaration, we can apply Law 15 from right to left in order to remove them from the model. As a result, the resulting specification has the same structure (signature and relations) of the models using explicit idiom.

Finally, we should refactor their formulae in order to show the semantic equivalence. By applying Law 4, which allows us to introduce and remove formulae *deducible* from a model, we must show that the formulae declared in `TracesPred` resulted from the previous steps, and the formulae declared in `TracesEvent` and `Checkin`, `Checkout`, `Entry` are equivalent. We should prove an implication that is similar to the following one, in which `form1` and `form2` are formulae.

```
(some g:Guest, r:Room, k:Key | form1) =>
  (some e:Event | form2)
```

We cannot prove that both models are equivalent using our laws. The previous formula is not deducible from the model. We cannot always construct an event for any

guest, room, key or time. By applying Refinement 1, which states that adding a formula to a model is a valid refinement in our notion, we can add the previous implication and prove that the model with explicit operations refines the model using implicit operations. So, the idiom using the explicit operations is a refinement of the implicit operations idiom.

In order to prove the equivalence, the model using explicit operations must have a formula generating an event for each parameter of the operations. For instance, we may add a generator axiom stating that there is an event for all operations parameters. Another solution is to declare `g`, `r` and `k` in the `Event` hierarchy to be bijections. We should add those bijections carefully because it is easy to introduce inconsistencies. A bijection introduces an implicit formula stating that the domain and image must have the same set cardinality. We need to propose new laws with semantic conditions in both situations, since most of our current laws [Gheyi 2007] have simple syntactic conditions.

4.4. Reasoning about Models

We can use our laws for reasoning about models. For example, Jackson specifies the previous hotel room locking models [Jackson 2006] but his model does not declare the last formula declared in the signature `Entry`. He informally argues that both models have the same state transition. Notice that we use our equivalence notion to compare the models, which is different from the notion used by Jackson [Jackson 2006] (same state transitions), in the previous section. We propose a new refinement notion because objects models are usually abstract and may not have operations. We relate the Z and our refinement notions elsewhere [Gheyi 2007].

We used the same approach presented in Section 4.3 to check whether Jackson's models have the same state transition. We successfully performed all refactorings steps in Figure 2 but the last, which checks whether the models have equivalent formulae. The Alloy Analyzer generates a counterexample when we check whether both models have equivalent formulae. It shows a scenario in which a guest enters into his/her room and modifies the current key of another guest's room. The formulae are not equivalent

The counterexample describes a valid transition of the explicit idiom. However, it is not valid in the implicit idiom because the constraint declared in the `Entry` signature (implicit operation) does not mention what happens to the current keys of the other rooms different from `TracesPred`, which explicitly states that they do not change. So, the models do not have the same state transition different from stated by Jackson. His model using events is underconstrained. We need to add the last formula declared in the `Entry` signature in his model. This mistake was introduced when specifying the Reiter's frame condition style.

5. Refactorings

In this section, we present a number of refactorings that can be derived from our laws presented in Section 3. In Alloy, we can specify the state of models describing behavior in two ways. We can specify one global signature containing all relations of the model's state. This idiom is called *global state* [Jackson 2006]. Another idiom called *local state* [Jackson 2006] allows different signatures to contain the state locally. Those idioms are important because they give flexibility for designers to choose the right idiom in each situation.

For example, the hotel locking key example presented in Section 2 declares each relation *locally* by extending it with an additional column corresponding to states (time). For example, `currKey` is declared in the `Room` signature instead of the global state (`State`). The name `State` would be more appropriate than `Time` in this case. This style is reminiscent of object-oriented programming: it packages together all the static and dynamic aspects of a single object, and it allows objects to be classified using signature extension [Jackson 2006]. Another way is to group all relations that are important for a given state in the `State` signature. Next we specify part of the hotel locking room example declaring one relation (`currKeyGlobal`), which maps the current key of a room in each state. This relation is declared in the global state.

```
sig Room { keys:  set Key }
sig Time {
  currKeyGlobal:  Room->Key, ...
}
```

Its advantage is that it separates static and dynamic aspects more cleanly, and supports a style of modeling (common in Z) in which the state is grown incrementally [Jackson 2006]. By using Laws 3, 4 and 7, we derive a refactoring that can change between the local and global idioms. The refactoring is stated using a textual notation different from our laws. The rs keyword denotes a set of relations.

Refactoring 1 ⟨change global to local state⟩

$ps\ [s \rightarrow expS]$
sig S {
  $rs1$
}
sig $State$ {
  $rs2$,
  r : S->T
}
	$=_{\Sigma,v}$	$ps\ [r \rightarrow expR]$
		sig S {
		  $rs1$,
		  s : T->$State$
		}
		sig $State$ {
		  $rs2$
		}

provided
($\leftrightarrow$) (1) if r and s belong to Σ, v contains the $r \rightarrow expR$ and $s \rightarrow expS$ items, respectively; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it;
($\rightarrow$) S's family in ps does not declare any relation named s;
($\leftarrow$) $State$'s family in ps does not declare any relation named r.
where

```
expR = { z:State, x:S, y:T |
  some st:State | (s.st!=none) and ((z->x->y) in st->s.st)
}
expS = { x:S, y:T, z:State |
  some st:State | (st.r!=none) and ((x->y->z) in st.r->st)
}
```

The $ps[x \rightarrow exp]$ notation states that x is replaced by exp in ps. Refactoring 2 allows us to collapse two related relations and declare one relation in the parent signature when applying from left to right. Applying from right to left, we can split a relation and declare one relation in each subsignature.

Refactoring 2 ⟨collapse relation⟩

$$
\begin{array}{|l|}
\hline
ps\ \ [r \rightarrow expR] \\
\textbf{abstract sig}\ S\ \{ \\
\quad rs \\
\} \\
\textbf{sig}\ T\ \textbf{extends}\ S\ \{ \\
\quad rsT, \\
\quad x:\ \textbf{set}\ V \\
\} \\
\textbf{sig}\ U\ \textbf{extends}\ S\ \{ \\
\quad rsU, \\
\quad y:\ \textbf{set}\ V \\
\} \\
\hline
\end{array}
\quad =_{\Sigma,v} \quad
\begin{array}{|l|}
\hline
ps[x \rightarrow expX,\ \ y \rightarrow expY] \\
\textbf{abstract sig}\ S\ \{ \\
\quad rs, \\
\quad r:\ \textbf{set}\ V \\
\} \\
\textbf{sig}\ T\ \textbf{extends}\ S\ \{ \\
\quad rsT \\
\} \\
\textbf{sig}\ U\ \textbf{extends}\ S\ \{ \\
\quad rsU \\
\} \\
\hline
\end{array}
$$

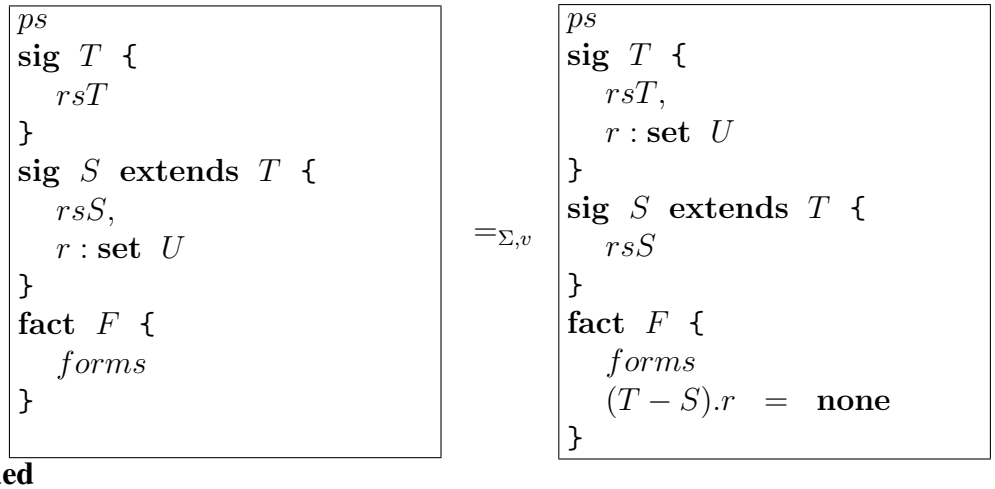
provided
(↔) (1) if r, x and y belong to Σ, v contains the $r \rightarrow expR$, $x \rightarrow expX$ and $y \rightarrow expY$ items, respectively; (2) for all names in Σ that are not in the resulting model, v must have exactly one valid item for it; (3) there is no signature extending S in ps;
(→) S's family in ps does not declare any relation named r;
(←) T and U's family in ps do not declare any relations named x and y, respectively.
where
`expR = x + y and expX = r&(T->V) and expY = r&(U->V)`

Refactoring 2 is derived from Laws 3, 4 and 7. It can be applied similarly to more than two subsignatures and relations with arity greater than two. If S is not abstract, Law 2 can be applied from left to right in order to make it abstract. Our laws can be useful for optimizations. For instance, the analysis performance of Alloy models with subtypes can be increased by *atomization* in the Alloy Analyzer 3 [Edwards et al. 2004]. When performing analysis, the Alloy Analyzer 3 internally transforms (atomizes) a model by splitting and pushing relations down to the lowest subtype level in order to improve its performance. Atomization applies a number of model transformations for removing a relation in a parent signature and introducing one relation to each subsignature. Applying Refactoring 2 from right to left, we can formally transform Alloy models with subtypes in order to improve the analysis performance. We have to apply this transformation, until there is no signature with a relation, except for the lowest subtype level. Formalizing this process is very important to guarantee the analysis quality, since it must be a completely sound transformation.

As another example, composing Laws 3, 4 and 7, we can formally derive the *Pull Up Field* and *Push Down Field* refactorings [Fowler 1999]. We can pull up a relation if it does not introduce name conflicts. Similarly, we can push down a relation if it does not introduce type errors. Refactoring 3 is different from the previous one because it does not

split a relation. It pulls up or pushes down the same relation.

Refactoring 3 ⟨pull up relation⟩



provided
(→) T's family in ps does not declare any relation named r;
(←) there does not exist an expression $exp.r$, such that $exp \leq T$ and $exp \not\leq S$, in ps or $forms$ or any valid item in v.

We derive other refactorings [Gheyi 2007], such as *Move Relation* refactoring by using Laws 3, 4 and 7. The *Extract Subclass* refactoring can be derived from Law 2 and Refactoring 3. Using Laws 3, 4, 7 and 8, we can formally *introduce a collection*. Since these refactorings are derived from sound laws, they are also sound.

6. Related Work

Banerjee et al. [Banerjee et al. 1987] propose a set of primitive transformations for object-oriented database schemas. These schemas can be represented by a subset of UML class diagrams. They propose well-formedness rules for schemas and argue that their transformations preserve it. They have transformations for adding and removing signatures, relations, inheritance and methods. We have proposed similar transformations, but we also focus on semantics preservation. So, some of their transformations do not preserve semantics. Moreover, we consider formulae differently from their work. It is more difficult to remove an element (signature or relation) in our approach than theirs, since we have to replace all element occurrences by its definition in all formulae before removing it. However, it is not always possible to find a definition for an element.

Bergstein [Bergstein 1991] proposes five primitive *object-preserving* class transformations. This work only considers models containing signatures and relations. Moreover, inheritance is only allowed with abstract signatures and invariants are ignored. The equivalence notion is restrictive. Two models can only be compared if they have the same base signatures, which are signatures that are not extended by others. So, using this notion, the models in Figure 1 cannot be compared. The set of transformations is shown to be complete under his notion. His work does not prove that the transformations preserve the well-formedness rules. Semantics preservation is informally argued. The conditions for each transformation are not precisely defined as our work.

Nipkow [Nipkow 2006] proved in Isabelle/HOL that both specifications of a hotel room locking based on the implicit and explicit idioms have the same state transition. His specifications are slightly different from Jackson's models. In our approach, we already proved a catalog of sound transformations. By composing them, we check whether two models are equivalent based on syntactic conditions in almost all steps but formulae equivalence instead of using theorem provers. Since we do not have the full completeness result, we cannot prove all equivalences by using our laws. It is better to use a theorem prover. On the other hand, it is easier for designers in practice to reason based on syntactic conditions than using a theorem prover.

Sunyé et al. [Sunyé et al. 2001] present a set of class diagrams refactorings for adding, removing and moving features. Enabling conditions are informally presented, but some of them are not feasible in practice to be implemented in a tool. Gogolla and Richters [Gogolla and Richters 1998] show some transformations for class diagrams and OCL constraints. Both aproaches do not propose a formal semantics for class diagrams and an equivalence notion. Both of them do not guarantee the type system preservation. So, OCL constraints can become ill-typed by applying some transformations. In some situations, some of the transformations proposed do not preserve semantics. Evans [Evans 1998] proposes deductive transformations for a subset of UML class diagrams. Semantics is proposed for a subset of UML class diagrams. He proposes five transformations, such as the *Pull Up Attribute*. These transformations can introduce type errors when considering OCL constraints. None of the previous approaches propose refactorings for introducing design patterns.

7. Conclusions

We use our comprehensive catalog of semantics-preserving transformations for Alloy to derive refactorings that allow us to formally introduce design patterns in Alloy models. For instance, we derived refactorings that can change between the global and local state idioms. This is very important since manual updates in order to introduce design patterns are an error-prone activity. Our catalog is a tool for reasoning about object model transformations, as we show that the two models, which are stated to be equivalent, describing the hotel room locking using two different idioms do not have the same state transition. Although we focus on equivalences, more interesting transformations can be derived using our refinement notion.

We have specified two theories [Gheyi et al. 2006] in Alloy of feature models (FM) [Czarnecki and Eisenecker 2000], which represent the common and the variable features of concept instances and the dependencies between the variable features of a Software Product Line. One theory can check more properties than the other theory, which is much more efficient for checking FM refactorings. We used our laws to show that both theories are equivalent with respect to an alphabet and a view. Additionally, we evaluate our catalog by refactoring the object model of Java AWT and Swing. Since Swing was created by extending AWT in order to avoid breaking the previous AWT clients, it introduced duplications [Bergel et al. 2005]. We used our catalog to remove some of them, such as attribute duplication. As a future work, we intend to propose Alloy idioms and derive refactorings for them.

References

Banerjee, J. et al. (1987). Semantics and implementation of schema evolution in object-oriented databases. In *Int. Conf. on Management of Data*, pages 311–322.

Bergel, A., Ducasse, S., and Nierstrasz, O. (2005). Classbox/j: controlling the scope of change in java. In *20th OOPSLA*, pages 177–189.

Bergstein, P. (1991). Object-preserving class transformations. In *OOPSLA*, pages 299–313.

Czarnecki, K. and Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Edwards, J. et al. (2004). Faster constraint solving with subtypes. In *ISSTA*, pages 232–242.

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

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

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

Gheyi, R. (2007). *A Refinement Theory for Alloy*. PhD thesis, Federal University of Pernambuco.

Gheyi, R., Massoni, T., and Borba, P. (2004). Basic laws of object modeling. In *3rd SAVCBS*, pages 18–25.

Gheyi, R., Massoni, T., and Borba, P. (2005a). An abstract equivalence notion for object models. *Electronic Notes in Theoretical Computer Science*, 130:3–21.

Gheyi, R., Massoni, T., and Borba, P. (2005b). A rigorous approach for proving model refactorings. In *20th Automated Software Engineering*, pages 372–375.

Gheyi, R., Massoni, T., and Borba, P. (2006). A theory for feature models in alloy. In *Alloy Workshop*, pages 71–80.

Gogolla, M. and Richters, M. (1998). Equivalence rules for UML class diagrams. In *UML*, pages 87–96.

Jackson, D. (2006). *Software Abstractions: Logic, Language and Analysis*. MIT press.

Jackson, D. et al. (2000). Alcoa: the alloy constraint analyzer. In *ICSE*, pages 730–733.

Kerievsky, J. (2004). *Refactoring To Patterns*. Addison-Wesley.

Nipkow, T. (2006). Verifying a hotel key card system. In *ICTAC*, volume 4281 of *LNCS*. Springer.

Owre, S. et al. (2007). PVS language reference. At http://pvs.csl.sri.com.

Reiter, R. (1991). The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artificial Intelligence and the Mathematical Theory of Computation*, pages 359–380.

Sunyé, G. et al. (2001). Refactoring UML models. In *4th UML*, pages 134–148.