Formal Model-Driven Program Refactoring

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

[1] Department of Computing Systems, University of Pernambuco
tlm@dsc.upe.br
[2] Informatics Center, Federal University of Pernambuco
{rg,phmb}@cin.ufpe.br

Abstract. Evolutionary tasks, specially refactoring, affect source code and object models, hindering correctness and conformance. Due to the gap between object models and programs, refactoring tasks get duplicated in commonly-used model-driven development approaches, such as Round-Trip Engineering. In this paper, we propose a formal approach to consistently refactor systems in a model-driven manner. Each object model refactoring applied by the user is associated with a sequence of behavior preserving program transformations, which can be semi-automatically performed to an initially conforming program. As a consequence, this foundation for model-driven refactoring guarantees behavior preservation of the target program, besides its conformance with the refactored object model. This approach is described in detail, along with its formal infrastructure, including a conformance relationship between object models and programs. A case study reveals evidence on issues that will surely recur in other model-driven development contexts.

1 Introduction

During development and maintenance activities, software evolution is acknowledged as a demanding task. The original software structure usually does not smoothly accommodate adaptations or additions. Evolution is further complicated by the adoption of models, for instance object models, which contain domain related structures and constraints that must be followed by the implementation. In this scenario, it is useful that abstractions in models and source code evolve consistently, for documentation or even development purposes, as seen in model-driven methodologies [1].

When evolving programs or models, maintaining those artifacts consistent is usually hard, requiring manual updates, even in state-of-the-art tool support. As a consequence, most projects abandon models early in the life cycle, adhering to code-driven approaches. A popular technique for dealing with a number of evolution-related problems is *refactoring* [2, 3], which improves software structure while preserving behavior. Nevertheless, the same issues are observed when refactoring multiple artifacts.

We propose a formal approach to *semi-automatically refactor programs in a model-driven manner*. A sequence of formal behavior preserving program transformations is associated to each predefined model refactoring. Applying a model refactoring triggers the corresponding sequence of program refactorings, which (1) update code declarations as refactored in the model and (2) adapt statements according to the modified declarations. This is accomplished with *invariants*, which are assumed throughout all program's executions, through a conformance relationship. Although developer only applies refactorings to object models, both artifacts get refactored, avoiding manual updates on source code.

Our approach is a formal investigation of evolution tasks, showing evidence about issues with keeping object models and their implementations in conformance during refactoring. We establish our approach on formal languages relying on previous work in object modeling and program refinement. For instance, object models in Alloy [4], which includes structures for expressing objects, relations and invariants equivalent to the core concepts of UML class diagrams. For programs, we consider a core language for Java, developed for reasoning on object-oriented programming [5].

Our solution differs from related approaches as it argues the application of transformations only on object model, and programs are made consistent by semi-automatic, independent transformations, while still maintaining some abstraction gap. Round-trip engineering, in contrast, generates changes to programs from models, which results in the need for manual updates. Alternatively, Model-Driven Architecture (MDA) [1] allows generation of platform-specific programs from a modeled description of the system. In this case, model abstraction is significantly compromised, as platform-independent program logic must be included into models for a complete executable generation. We developed a case study with our approach for illustrating evidence on issues that will surely recur in other Model-Driven Development contexts.

2 Motivating Example

An Alloy model contains a sequence of *paragraphs*; one kind of paragraph is a *signature*, which are used for defining new types. Instances of these signatures can be related by the *relations* declared in the signatures. As an example, we use a simple object model of a file system in Alloy, as implemented in commonly used operating systems. The following fragment defines signatures for `FSObject` and `Name`. The keyword `sig` declares a signature with a name. Signature `Name` is an empty signature, while `FSObject` declares two relations. For example, every instance of `FSObject` is related to exactly one instance of signature `Name` by the relation `name` – the keyword `one` denotes an injective relation. Also, file system objects may have `contents`; it is optional, and maybe more than one instance, since it is annotated with the `set` keyword, which establishes no constraints on the relation. Also, signature may extend another, with the `extends` keyword, establishing that the extended signature is a subset of the parent signature – `File` and `Dir`. In addition, `Root` is a subtype of `Dir`.

```
sig Name {}
sig FSObject {
  name: one Name,
  contents: set FSObject }
sig File,Dir extends FSObject {}
sig Root extends Dir {}
```

We may further constrain this object model with invariants, for establishing more complex domain rules concerning the declared signatures and relations. For this purpose, an Alloy model can be enriched with formula paragraphs called *fact*, which is used to package formulas that always hold for the model. The first formula in the following fact states that there can be exactly one `Root` instance in every file system, by the `one` keyword. Next, the second formula defines that from all file system objects, only directories may present contents. In the expression `(FSObject-Dir).contents`, the join operator (.) represents relational dereference (in this case, yielding contents from the set of instances resulting from `FSObject` instances that are not directories; the `-` symbol is equivalent to set difference). The `no` keyword establishes that the expression that follows results in an empty set, which gives the constraint the following meaning: only directories have contents in the file system.

```
fact {
  one Root
  no (FSObject-Dir).contents }
```

The following code fragment shows a direct implementation for this file system in a formal Java-like language [5], which is used as basis for the formal investigation showed in this paper. Classes declare fields, methods and constructors. The `main` method, within the `Main` class reads a file system object from the user (with the input/output field `inout`), and in the case of a directory, adds a new `File`; otherwise, none is added. The keyword `is` corresponds to Java's `instanceof`. Also, a constructor can be declared with the **constr** keyword. The **set** modifier establish a collection variable (set of objects). Keywords **result** and **self** denote, respectively, method return and the current object. Following the invariant from the object model, only directories have contents.

```
class FSObject{
  pri Name name;
  pub Dir set contents;
  set Dir getContents() { result:= self.contents; }
  void setContents(set FSObject c) { self.contents:= c } }
class Dir extends FSObject{..}
class File extends FSObject{ constr {.. self.contents:= ∅ }
class Main{
   void main(){
     File f:= new File ,currentFSObj:= null in ..
        currentFSObj:= (FSObject)self.inout;
        if (currentFSObj is Dir)
          then currentFSObj.setContents({f})
          else currentFSObj.setContents(∅) } }
```

The source code shows a situation for likely refactoring: there is a field in the superclass, `contents`, which is only useful in one of the subclasses – a "bad smell" [2]. The refactoring can be accomplished by moving `contents` down to `Dir`. Additional changes are needed for correctness, such as updating accesses to `contents` with casts to `Dir`. Issues with evolution arise as the object model is updated for conformance. Round-trip Engineering (RTE) [6] tools are popular choices for automation, applying reverse engineering for recovering object models from code. In this case, usually models become rather concrete, as *visualizations of source code, losing abstraction*.

On the other hand, the mentioned "bad smell" can be detected in the object model. The relation is then pushed down from `FSObject` to `Dir`. Likewise, conformance of the source code is desirable; in the context of RTE, a usual technique for automatic updates to the source code marks previously edited code fragments as immutable, in order to avoid loss of non-generated code that is marked by the tool. However, if the field is simply moved to `Dir`, *these immutable statements become incorrect*; for instance, accesses to the field with left expressions not exactly typed as `Dir`, as within `FSObject` and `File`, showed next.

```
class FSObject{ ..
  set Dir getContents() { result:= self.contents; }
  void setContents(set FSObject c) { self.contents:= c } }
class File extends FSObject{...
   constr {...self.contents:= ∅...} }
```

Due to the representation gap between object models and programs, the immutable code may rely on model elements that were modified, showing a recurring evolution problem in RTE-based tools. Manual updates must be applied, making evolution more expensive.

An alternative is employed in tools following the *Model-driven architecture*(MDA) [1]. Models in MDA include programming logic, written in a platform-independent fashion – for instance, action semantics in executable UML [7]. Tool support then generates source code for that logic in a specific implementation platform. Under such approach the model refactoring in this section's example would also include changes to the programming logic attached to models, maintaining source code up-to-date [8]. Although the MDA approach might be promising, it lacks a more abstract view of the domain given by object models, which is still useful for overall understanding. Our approach investigates model-driven refactorings in this context.

3 Model Transformations

As model refactoring must preserve semantics, and the task is usually accomplished in an *ad hoc* way, unexpected results are usually observed. We can improve safety by founding refactoring over semantics-preserving *primitive transformations*, which can be composed into refactorings preserving semantics by construction. Examples of these primitive transformations are showed in Section 3.1, while refactoring composition is explained in Section 3.2.

3.1 Primitive Transformations

For the Alloy language [4], a catalog of primitive transformations has been proposed [9]. Although the transformations are language specific, they can be leveraged to other object modeling notations, such as UML class diagrams, as we previously investigated [10].

These primitive transformations are presented as equivalences, which delimit two transformations; they are regarded as primitive since they cannot be derived from other transformations. In addition, syntactic conditions are defined as requirements for a correct application. Such conditions ensure that the transformation preserves the semantics of the object model, while maintaining its well-formedness. Soudness, completeness and compositionality have been proved for the catalog [9]. The equivalences use a flexible equivalence notion [11] as basis for comparing object models.

For instance, the catalog includes an equivalence for introducing and removing relations. Equivalence 1 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 (a set of pair of objects). Likewise, we can remove a relation that is not being used, by applying the reverse transformation.

Equivalence 1 ⟨introduce relation and its definition⟩

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

provided
(→) (1) *S*'s hierarchy in *ps* does not declare any relation named *r*; (2) *r* is not used in *exp*, or *exp* is *r*; (3) *exp* is a subtype of *r*;
(←) *r* is not used in *ps*.

We use *ps* to denote the remainder of the model (signatures, relations and facts). Each equivalence defines two templates of models on the left and the right side, with syntactic conditions. We write (→), before the condition, to indicate that this condition is required when applying the left-to-right (L-R) transformation. Similarly, we use (←) to indicate what is required when applying the opposite transformation (R-L).

Equivalence 2 introduces transformations for adding or removing a subsignature from an existing hierarchy. We can add an empty subsignature if declared with a fresh name. After this transformation, the supersignature becomes abstract (defining no direct instances), as denoted by the resulting constraint (`S=U-T`). Similarly, the subsignature can be removed if it is not being used elsewhere and there is no formula in the model with its type, in order to avoid type errors – $\leq$ denotes subtype.

Equivalence 2 ⟨introduce subsignature⟩

```
ps
sig U { rsU }
sig T extends U{ rsT }
fact F { forms }
```

=

```
ps
sig U { rsU }
sig T extends U{ rsT }
sig S extends U{}
fact F {
  forms AND S = U − T
}
```

provided
(→) (1) *ps* does not declare any signature named S; (2) there is no signature in *ps* that is subsignature of U;
(←) (1) S does not appear in *ps*; (2) *exp*, where $exp \leq U$ and $exp \nleq T$, does not appear in *ps*.

3.2 Model Refactorings

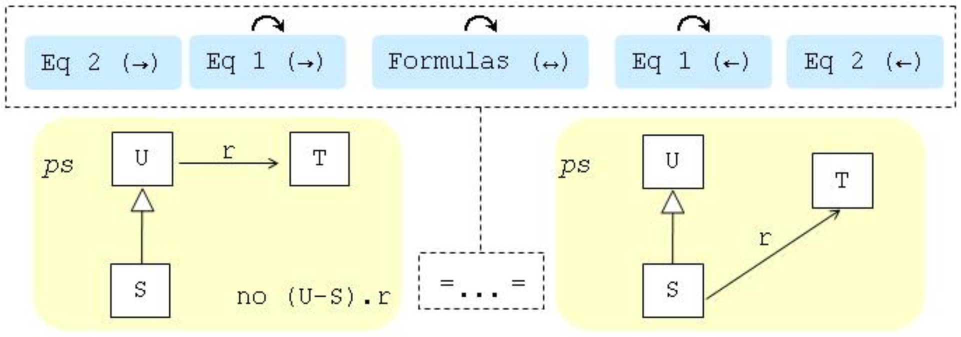
The primitive Alloy transformations can be used as basis for several applications that require semantics-preserving transformations, for instance *model refactorings*. Since the primitive transformations are simpler – dealing with a few language constructs – they can be more easily proved to be semantics-preserving. By construction, a composition of correct primitive transformations is also correct, providing safe refactorings for object models [9].

Refactoring 1 depicts a formal rule for pushing down a relation, which can be done if the model presents an invariant stating that the relation's range is within subsignature `S`. Similarly, we can pull up a relation by adding this constraint to the model. This refactoring is derived by successive applications of transformations from the Equivalences 1 and 2. It must be clear that this derivation is carried out only once by a refactoring designer; once done, the refactoring is ready to be applied.

4 Laws of Programming

For establishing a formal basis for program refactoring, we use *laws of programming* [12]. An extensive set of laws of object-oriented programming has been defined for ROOL [13], a formal subset of Java. These laws, proved to be behavior-preserving according to a formal semantics, provide a formal basis for defining program refactorings. These laws have been proved, however, for a copy semantics, in which objects are *values*, not referenced by pointers. This decision simplified the semantics to the point in which certain laws – in special laws based on class refinement [13] – require simpler proofs, as *aliasing* can be ruled out, allowing direct modular reasoning. However, it is rather inconvenient to define refactoring on a language that is disconnected from the practice of OO programming. Therefore, we transferred the catalog to a reference-based language defined in the work of Banerjee and Naumann [5], since it guarantees modular

Refactoring 1 ⟨Push Down Relation⟩



(→) $exp.r$, where $exp \leq U$ and $exp \not\leq S$, does not appear in ps;
(←) S's family does not declare any relation named r.

reasoning by using *ownership confinement* as a requirement for programs; therefore, we require confinement in the programs to be refactored. Laws related to class refinement can be proved by showing a correct simulation, as showed in their work. Most laws used in our work are clearly still sound in the presence of reference semantics; even though, we developed proofs for a demonstrative number of laws in the new language.

For instance, the following law eliminates or introduces classes. Classes can be removed when they have no use anywhere in the program. Similar to model equivalences, laws denote two transformations (from left to right and vice-versa). The class to be eliminated or introduced is denoted by cd_1 – CT is a *class table*, the context of class declarations.

Law 1 ⟨*class elimination*⟩
$CT \; cd_1 \;=\; CT$

provided
(↔) $cd_1 \neq$ `Main`;
(→) cd_1 is not used in CT.
(←) (1) cd_1 is a distinct name; (2) field, method and superclass types within cd_1 are declared in CT.

In addition to equivalence laws, reasoning about classes usually requires a notion of class refinement, for changes of representation like addition and removal of private fields. The application of refinement changes the bodies of the methods in the target class, using a *coupling invariant* that relates the old and the new field declarations [13].

5 Model-Driven Refactoring

In this section, we present our model-driven approach to refactoring. The catalog of primitive model transformations is used for model refactorings; the applied primitive transformations form the basis for defining semi-automatic refactoring in programs. The approach is delineated in Section 5.1, and the required notions of conformance and confinement are showed in Section 5.2. The approach is explained in detail in Section 5.3, as applied to the file system example.

5.1 The Approach

Our solution considers the context in which object models can be refactored using the formal primitive transformations, directly or indirectly (in the latter case, by using refactorings derived from primitive transformations). To each primitive model transformation from the catalog, we associate a sequence of program transformations – a *strategy* –, to be applied to a program, with no user intervention, guided by laws of programming.

The approach is depicted in Figure 1, where OM represents an object model, and P a program. In (a) we partially repeat the sequence of model transformations applied in Refactoring 1; after the model refactoring, all primitive transformations are *recorded* (for instance, in a CASE tool), for later association with the strategies. We associate each primitive model transformation (depicted as “corresponds” in Figure 1) with a specific strategy, which will semi-automatically refactor the program in (b), resulting in a program consistent with the refactored program. Strategies are constructed as the automatic application of laws of programming – ensuring behavior preservation –, denoted as L_k.

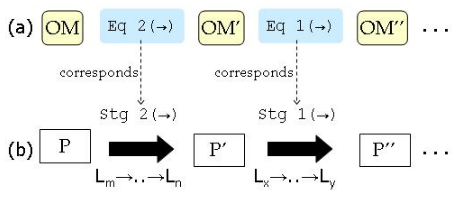


Fig. 1. Model-driven refactoring with strategies

A strategy performs program refactoring following a sequence of transformations on the assumption of conformance. Therefore, it is especially conceived to exploit the model invariants that are known to be met by the program; hence, program transformations can be more powerful with high-level assumptions about program declarations, such as classes and fields. In this context, a strategy must rewrite programs for updating correspondent declarations that were refactored in the object model and preserve program behavior. In an iterative software process, this approach could be applied when refactoring models during analysis and design activities, and some source code is already implemented from previous iterations. The user would have to ensure the conformance relationship between the source code and the object model, which is the precondition for applying the approach, by other conformance checking tool, since it is beyond the scope of this paper.

Regarding tool support, this approach could be implemented primarily as a modeling tool, which would include a built-in catalog of primitive model transformations, which could be composed into refactorings. The application of a refactoring could be recorded in terms of the primitive transformations composing this refactoring. Next, strategies correspondent to each applied primitive are applied to the attached source code, in the same order as the primitives were applied to the model. With such a supporting tool, our approach differs from Round-trip Engineering as it avoids generation of source code from models (or models from code) completely. In fact, object model and program refactorings take place independently. Similarly, MDA-based tools are different as they include programming logic information, which yields more concrete models.

5.2 The Required Conformance Relationship

In order to establish which changes each strategy is expected to carry out, we define assumptions that constrain programs that are amenable to model-driven refactoring. We assume *conformance* before applying transformations, and strategies guarantee its preservation. In this context, a definition of conformance must be established for defining which programs may be refactored by strategies associated to a model transformation.

Semantically, a program is in conformance with a object model if its states – heaps of linked objects – meet the modeled invariants. In the file system model in Section 2, a conforming program must create a single `Root` directory in the system, for example, for all possible execution states. Additionally, we assume a syntactic conformance between models and programs in which model structures (signatures and relations) strictly correspond to program structures (classes and fields). Also, relations with multiplicity (0..1) or (1) are implemented as single fields; differently, unconstrained multiplicities are implemented by set fields. Signature hierarchies must be maintained by classes, although additional abstract classes can be declared between two modeled classes. A deep discussion on other forms of implementing object models can be seen in our previous work on formal conformance between object models and programs [14].

Also, for ensuring that program transformations that employ some form of class refinement are correct, aliasing is partially restricted; for example, class invariants may be invalidated if other classes share references to internal elements of this class (representation exposure). Therefore we enforce that programs to be refactored with our approach present syntactic properties that guarantee ownership confinement. A related work presents some syntax directed static analysis rules for confinement [5], which we direct apply in our work.

5.3 Applying the Approach

Regarding the file system example, the object model invariant guarantees that the `contents` relation will be empty for any instance of `FSObject` subsignatures, except for `Dir`. In the required conformance relationship, the invariant is always valid in the program. This allows strategies to be semi-automatically applied for each primitive transformations carried out in the object model.

After the application of Refactoring 1, a number of primitive model transformations, have been applied, as showed in Section 3.2. First, Equivalence 2 was applied, from left to right; it is correspondent to the strategy *introduceSubclass*, according to Table 1. This table also presents all strategies from our approach.

Table 1. Strategies Corresponding to Model Equivalences

Model Equivalence	**Strategy →**	**Strategy ←**
Introduce Signature	*introduceClass*	*removeClass*
Introduce Generalization	*introduceSuperclass*	*removeSuperclass*
Introduce Subsignature	*introduceSubclass*	*removeSubclass*
Introduce Relation	*introduceField*	*removeField*
Remove Optional Relation	*fromOptionalToSetField*	*fromSetToOptionalField*
Remove Scalar Relation	*fromSingleToSetField*	*fromSetToSingleField*
Split Relation	*splitField*	*removeIndirectReference*

We formalize strategies using *refinement tactics*, based on the ArcAngel language [15], in order to add preciseness to the description for easier implementation in the transformation language of choice. We present the main constructs of the language while showing the *introduceSubclass* strategy. Law applications are denoted by the keyword **law**. Alternation (`|`) establishes that if the first application fails, the second one is executed; in the following strategy, we consider that the program may already have a class with the same name of the introduced subclass (`C`, in the example `X`), so we rename the existing class (`C’`). If it fails, the name is new; with **skip**, nothing else happens in this case. After introducing `X`, with `FSObject` as its superclass, we replace all commands of type `x:= new FSObject` by `x:= new X`, using a law of programming called *new superclass* [13]. Equivalence 2 establishes on the right-hand side the superclass as an abstract class, thus objects of `FSObject` may no longer exist.

```
Tactic introduceSubclass(C:Class,SC:Class)
   applies to program do
     (law rename(C,"C’") | skip);
     setExtends(C,SC);
     law classElimination(C,←);
     law newSuperclass(SC,C,→);
end
```

Then for each recorded model transformation, strategies are semi-automatically applied following the correspondence in Table 1. For instance, auxiliary fields `contentsDir`, `contentsFile` and `contentsX` are introduced with strategy *Introduce Field*, with invariant (**self**`.contentsFile =`**self**`.contents` ∧ (**self is** `File`)). Writings to **self**`.contents` are then extended with additional commands, as follows. In the outcome, the desired invariant is enforced.

```
class File { ..
      self.contents:= ∅;
      if (self is File) then ((File)self).contentsFile:= self.contents }
```

Deduction of invariants, such as `contents = contentsDir` + `contentsFile` + `contentsX`, do not affect the source code, therefore no strategies are associated with these model transformations. The subsequent strategy removes the `contents` field based on a given invariant definition for the corresponding relation in the model; this strategy uses class refinement for removing a field after adjusting the code for replacing its occurrences with the equivalent expression from the invariant.

After removing the auxiliary subclass `X` and renaming the new `contentsDir` field to `contents`, a partial view of the resulting program is showed in the following program fragment.

```
class FSObject{ ..
  set Dir getContents() {
    if (self is Dir) then result:= ((Dir)self).contents }
  void setContents(set FSObject c) {
    ((Dir)self).contents:= c } }
class Dir extends FSObject{ set Dir contents;.. }
class File extends FSObject{ constr {..} }
class Main{ ..
  void main(){ ..
      if (currentFSObj is Dir)
        then currentFSObj.setContents({f})
        else currentFSObj.setContents(∅) .. }
```

We previously mentioned that strategies are *semi-automatic*, as their automation is limited by a number of issues, mainly problems in strategies with class refinement steps, whose proofs are not automatic. These strategies involve changes in the internal representation of a class or hierarchy. For instance, this is crucial in strategies that introduce or change fields, implicating in changes to the methods internal to this class (as exemplified by strategy *introduceField*). First, the generation of a coupling invariant for the refinement is not straightforward, and even harder it is to prove the simulation for an arbitrary class. This problem is identified in this work, albeit we do not intend to provide a solution at this point.

For validating the correctness of our approach, we developed *soundness proofs* for each strategy, ensuring that they preserve program behavior, do not break confinement and the refactored program is in conformance with the refactored model. These proofs are not showed here due to space restrictions.

6 Case Study

In this section we apply our approach in a case study contemplating a recurring example for refactoring, taken from Fowler's book on refactoring [2]. From an initial object model of the video rental domain, we established three (3) refactorings; these refactorings are formed of primitive model transformations from the catalog. Next, program strategies can be semi-automatically applied to an original conforming program. Figure 2(a) shows the initial object model subject to refactoring, by means of a UML class diagram. The invariant states that a rental is never registered to both a customer and one of its dependents (# refers to set cardinality). The refactored model is depicted in Figure 2(b).

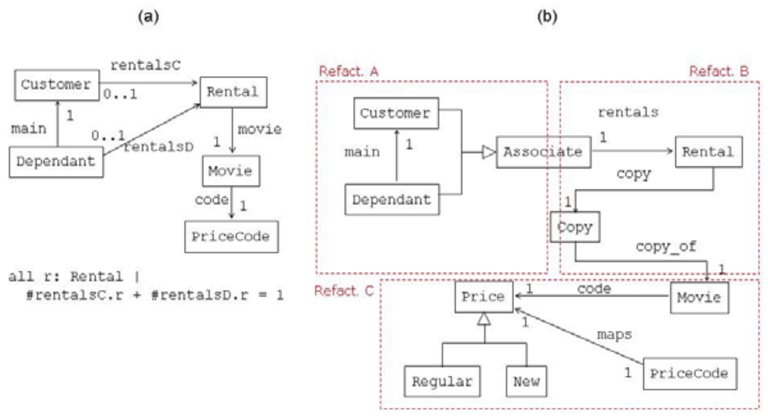


Fig. 2. Video Rental Object Model

The applied model refactorings are described as follows with results highlighted in Figure 2(b):

A extract a signature, `Associate`, defining a general structure for people who can rent movies. This refactoring is made of generalization introduction and a new relation (`rentals`), removing the original relations `rentalsC` and `rentalsD`;

B add a new signature – `Copy` – between `Rental` and `Movie`, since more than one copy is available for one movie. For this, we must split the relation `movie` into two new relations, before its removal;

C restructure the relationship between a movie and its current status – new or regular – based on the State Design Pattern. For this, we must introduce `New` and `Regular`, with the `Price` supersignature, and move `code` to Price.

Assuming that these primitive transformations have been recorded as a queue by a supporting tool, the matching strategies are semi-automatically applied, following the queue order. For the first refactoring, the opening strategy automatically introduces the `Associate` superclass. Other strategies include the new field `rentals`, replacing old references for the previous fields `rentalsC` and `rentalsD`; this replacement cannot be made automatically, due to the class refinement issue mentioned in Section 5.

For adding movie copies, the *splitField* strategy creates two fields `copy` and `copy_of`, besides a new class (`Copy`). The class refinement contained in this strategy can automatically duplicate writings using the old field, as exemplified in the next program fragment.

```
self.movie:= myMovie; self.copy.copy_of:= self.movie;
```

In the end, `movie` is eliminated. The refactored program presents an interesting aspect: even though the concept of copy is introduced, the program logic still considers movies to be "the copies". Therefore, although the resulting program has its behavior preserved, being also in conformance with the resulting model, but not reflecting the user intent. This aspect may be a limitation of model-driven refactorings from object models. We visualize two potential solutions: more concrete models with programming logic, as in MDA [1], or to consider behavioral models, in addition to object models (a possible future work). Furthermore, user interaction could be used here for application-specific transformations that could change this result.

Adding the State Design Pattern includes strategies for adding three new classes and moving the `code` field. The automatic application of the latter is only possible by explicitly defining a one-to-one relation between a `Code` to `Price` (`maps` field). This information is actually implicit in the Fowler's example in the book [2], but an analysis by means of model transformations made it evident.

7 Related Work

Opdyke proposes refactorings to which a number of preconditions are attached, which, if satisfied by the target program, ensure the preservation of behavior [3]. His work is similar to ours as it proposes a number of primitive refactorings, including creation, deletion and change of program entities. In contrast, semantics preservation is informally defined as a number of properties – related to inheritance, scope, type compatibility and semantic equivalence – that are required to hold after applying the refactoring.

A closely related approach was developed by Tip et al. [16]. They realized that some enabling conditions and modifications to source code, for automated refactorings, depend on relationships between types of variables, evident in refactorings involving subtypes. These type constraints enable the tool to selectively perform transformations on source code, avoiding type errors that would otherwise prohibit the overall application of the refactoring. This approach is similar to ours in the sense that both use additional information (in our case, model invariants) in order to achieve advanced refactorings. In fact, both approaches might be even integrated.

Co-evolution between models and programs is dealt with by several related approaches. For instance, Harrison et al. [17] show a method for maintaining conformance between models (UML class diagrams) and Java programs, by advanced code generation from models at a higher level of abstraction, compared to simple graphical code visualization. This approach is related to ours, as the relationship between model and source code avoids round-tripping. Their conformance relationship is more flexible, as we require a more strict structural similitude between the artifacts. No details are offered on how conformance mappings will consistently evolve.

Conformance is more specifically addressed by another work [18], aims at bridging the gap between object-oriented modeling and programming languages, in particular regarding binary associations, aggregations and compositions in UML class diagrams. They describe algorithms that detect automatically these relationships in code. As the semantics of such constructs is not well-defined, the authors provide their own interpretation from textual descriptions. This approach introduces a more flexible conformance relationship for relationships between objects, while still maintaining the model abstract, which may be applicable to our solution as well. However, further investigation is required for assessing potential benefits of this notion to model-driven refactoring based on algebraic laws.

8 Conclusions

In this paper, we propose a formal approach for program refactoring by refactoring only object models, maintaining program conformance by correct and semi-automatic refactorings corresponding to the applied model transformations. The approach is backed by a formal infrastructure of transformation laws, which are proved to be semantics preserving for both object models and programs, and a conformance relationship. We also exemplified the approach with a case study.

The Alloy laws can be directly leveraged for UML class diagrams [10], as well as the used programming language is similar to mainstream OO languages, as Java and C++. The investigation of model refactoring and its implications to source code in this work provides evidence over the challenges that effective model-driven methodologies will face in order to support evolution. In this context, the level of abstraction is a key aspect; in this research we ended up with a tighter conformance relationship than initially expected. First, for useful model refactoring, the main declarations often must be maintained. Second, less restrictions to the source code implementation imply in more transformations required to make the source code conforming to the refactored model, which would lower the quality of the outcome. Nevertheless, the required conformance relationship still preserves some abstraction: methods and additional classes can be freely implemented, and hierarchies can contain more classes then modeled.

Assumptions include reliance on the maturity of conformance checking tool support in practice – still incipient in practice – and a closed-world context in which we have access to the full source code of a program. Also, strategies involving class refinement may present limitations, in particular when an arbitrary formula must be translated to the coupling invariant for the refinement transformations. This situation reduces to the complex problem of generating code from a logical formula. However, we believe that automation can still be applied to some extent; interaction with users could be an alternative.

References

1. Kleppe, A., et al.: MDA Explained: the Practice and Promise of The Model Driven Architecture. Addison Wesley (2003)
2. Fowler, M.: Refactoring—Improving the Design of Existing Code. (1999)
3. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
4. Jackson, D.: Software Abstractions: Logic, Language and Analysis. (2006)
5. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM **52**(6) (2005) 894–960
6. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Workshop on Best Practices for Model-Driven Software Development (OOPSLA 2004)
7. Balcer, M.J., Mellor, S.J.: Executable UML: A Foundation for Model Driven Architecture. (2002)
8. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal **45**(3) 451–461
9. Gheyi, R.: A Refinement Theory for Alloy. PhD thesis, Informatics Center – Federal University of Pernambuco (August 2007)
10. Massoni, T., Gheyi, R., Borba, P.: Formal Refactoring for UML Class Diagrams. In: 19th SBES, Uberlandia, Brazil (2005) 152–167
11. Gheyi, R., Massoni, T., Borba, P.: An abstract equivalence notion for object models. ENTCS **130** (2005) 3–21
12. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of Programming. Communications of the ACM **30**(8) (1987) 672–686
13. Borba, P., et al.: Algebraic Reasoning for Object-Oriented Programming. Science of Computer Programming **52** (October 2004) 53–100
14. Massoni, T., Gheyi, R., Borba, P.: A Formal Framework for Establishing Conformance between Object Models and Object-Oriented Programs. In: SBMF. (2006) 201–216
15. Oliveira, M., Cavalcanti, A., Woodcock, J.: ArcAngel: a Tactic Language for Refinement. Formal Aspects of Computing **15**(1) (2003) 28–47
16. Tip, F., et al.: Refactoring for Generalization Using Type Constraints. In: 18th OOPSLA, ACM Press (2003) 13–26
17. Harrison, W., et al.: Mapping UML Designs to Java. In: Proceedings of OOPSLA 2000, ACM Press (2000) 178–187
18. Guéhéneuc, Y.G., Albin-Amiot, H.: Recovering Binary Class Relationships: Putting Icing on the UML Cake. In: Proceedings of the 19th OOPSLA, ACM Press (October 2004) 301–314