Synchronizing Model and Program Refactoring
Technical Report

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

[1] Federal University of Campina Grande
{massoni,rohit}@dsc.ufcg.edu.br
[2] Federal University of Pernambuco
phmb@cin.ufpe.br

Abstract. Object models provide abstract information about software structure, but their maintenance is difficult after refactoring takes place. In Model-Driven Development (MDD), effective transferral of model refactoring changes to programs is problematic, especially if these programs are subject to developer manipulation. Consequently, code-driven approaches end up being adopted. We formalize a theory of synchronizers, which are sequences of behavior-preserving program transformations. This theory makes use of invariant-based refactoring, the key idea behind synchronizers. We also establish and prove a soundness theorem for synchronizers. By uncovering the formal requirements for correct refactoring synchronization, the proved properties point out issues – regarding consistency, refactoring automation and quality – that recur in several MDD settings that employ object models.

1 Introduction

Refactoring [1,2] improves program structure while preserving behavior. Additional benefits can be obtained from *object model* [3,4] *refactoring*, useful for restructuring software abstractions and invariants, by means of *semantics-preserving* transformations. Synchronization of these transformations to source code is essential [5] in Model-Driven Development (MDD) contexts [6].

However, this is an open problem in the MDD community, especially when both models and programs are manipulated [7]. Automatic generation of artifacts has long been known for their limitations – automation is hard to achieve [8]. In fact, the relationship between object model and object-oriented (OO) program constructs may be complex to deal with, and tools often fail to deal with the desired abstraction gap. As a consequence, many projects abandon models early in the life cycle, adhering to code-driven approaches. Methods and tools for – at least partially – removing human interaction in the process are invaluable to the refactoring practice. Several approaches try to deal with the relationship between model and program transformations [9,10,11,12,13], although, to the best of our knowledge, none has analyzed specific aspects of refactoring and synchronization issues between object models and source code.

This article presents a formal model for synchronized refactoring of object model and programs by means of *proven* primitive semantics-preserving transformations. In particular, our theory is centered on a model-driven approach on which each model transformation is associated with a *synchronizer*, a sequence of program transformations, which (1) updates code declarations and (2) adapts statements according to the modified declarations – as explained in Section 3. This unidirectional (model to code)

approach to synchronization presents a fundamental effect: it allows powerful program refactoring directly from abstract information provided by object model invariants. Previous publications delineate the model-driven approach to refactoring [14], and preliminary conclusions on the use of invariants as basis for automatic refactoring [15]. Those contributions are extended in this paper with a description of synchronizers, soundness proofs and discussion (Sections 3, 4 and 5, respectively).

We establish our theory on previous work in primitive model [16] and program transformations [17,18]. Object models in Alloy [3] express objects, relations and invariants equivalently to the core concepts of UML class diagrams. For programs, we consider a Java-like language [19]. These languages are explained in Section 2. A general soundness theorem is defined and proved for synchronizers (Section 4); in this proof, we ensure that synchronizers are program refinements (preserving behavior) and the refactored program is consistent with the refactored model. Consistency is defined in terms of syntax and semantics; only the syntactic consistency must be adjusted for other languages – the semantic mapping is language independent. The need for the proved properties unveils issues that will recur in several MDD contexts that employ object models, related to automation and refactoring quality (Section 5).

2 Languages

In our theory, we consider object models in Alloy [3], and programs in a Java-like language, developed for reasoning about object-oriented programming [19].

2.1 Model refactoring

Alloy [3] presents formal type system and semantics for writing object models. An Alloy model contains a sequence of *paragraphs*; a *signature* defines a new type. A signature paragraph introduces a basic type and a collection of *relations*, along with their types and constraints on their values. For instance, an object model for a file system defines signatures for `FSObject` and `Name` – **`set`** defines unconstrained relations. In Alloy version 3, one signature can extend another, establishing that the extended signature is a subset of its supersignature (`File`). In addition, facts are used to package model invariants. In the following fragment, the formula states that, from all file system objects, only directories may contain other objects. The join operator '.' represents relational dereference, and `FSObject - Dir` expresses all `FSObjects` instances that are not directories (set difference has the conventional meaning).

```
sig Name {}
sig FSObject {  name: set Name, contents: set FSObject  }
fact { (FSObject-Dir).contents = {} }
sig File extends FSObject {}
```

Regarding model transformations in Alloy, a catalog of primitive transformations was proposed [16]. *Algebraic laws* formalize two primitive transformations; equivalence allows application of the law in both directions. As an example of a law, a new subsignature can be introduced or removed from an existing hierarchy (Law 1). An empty subsignature X can be introduced if declared with a fresh name. After this transformation, the supersignature U becomes abstract (defining no direct instances), as denoted by the invariant denoting the objects in X are the objects in U, except those in S

or T ($X = U - S - T$). Similarly, X can be removed if it is not being used and no expression exp of its type exists ($exp \leq U$, but $exp \not\leq S$ and $exp \not\leq T$), where $\leq$ denotes subtyping. Some meta-variables are useful as notation: rs represents relations, while $forms$ represents a set of formulas. Each box represents a template with which actual Alloy declarations can match (ps denotes the paragraphs that are not showed in the template). Additionally, below the templates, provisos that ensure transformation correctness are established. ($\leftarrow$) defines provisos for application from Left-to-Right (L-R), while ($\rightarrow$) defines provisos for applying the law from Right-to-Left (R-L).

Alloy Law 1 ⟨*introduce subsignature*⟩

ps
sig U { rsU }
sig S **extends** U{ rsS }
sig T **extends** U{ rsT }
fact F {$forms$}
	=	ps
		sig U {rsU}
		sig S **extends** U{ rsS }
		sig T **extends** U{ rsT }
		sig X **extends** U{}
		fact F {
		$forms$
		$X = U - S - T$
		}

provided

($\rightarrow$) (1) ps does not declare any paragraph named X; (2) there is no signature in ps that extends U;

($\leftarrow$) X does not appear in ps, rsU, rsS, rsT and $forms$; (2) there is no expression exp, where $exp \leq U$ and $exp \not\leq S$ and $exp \not\leq T$, in ps or $forms$.

The equivalence respects the notion proposed for Alloy [20]. The catalog of Alloy laws has been proven sound and complete in a theorem prover [21]. Furthermore, these laws can be used as basis for several applications that require semantics-preserving transformations, such as *model refactorings*. Since primitive laws are simpler – dealing with a few language constructs – they can be more easily proven sound. By construction, a composition of laws is also correct, providing safe refactorings for object models.

2.2 Program Refactoring

The Java-like language is inspired by Banerjee and Naumann's work [19]; it does not consider interfaces, multithreading or library classes. A program is a set of classes, a *class table* CT, which always includes a class named `Main`, with a main method as the starting point of execution. A generic class declaration is defined as follows: **class** C **extends** D { $\bar{T}$ $\bar{f}$; $\bar{M}$ }. , where $\bar{T}$ $\bar{f}$ stands for typed fields in the class, while $\bar{M}$ represents a list of methods.

Only **bool** and **unit** (the empty type) are predefined as primitive types in the language; from these types, other primitives may be built. Furthermore, expressions do not have side effects; object construction occurs only as a command. Similarly, method calls occur in special assignments `x:= e.m(`$\bar{e}$`)` defining both side effect and a return value. Methods can be defined recursively, so loops are omitted.

We adapted to this language a complete set of laws of programming [18]. The following law, for instance, establishes that instantiations of a class B can be replaced by instantiations of its superclass A, as long as B is an empty class. This replacement can occur either in the body of A's methods or any method in the set of class declarations CT (a class table). $CT[exp'/exp]$ denotes a substitution. Replacement is applicable when expressions of type A are not cast with B and B instances are assigned only to

A-typed variables. The opposite application is constrained by a proviso: tests involving A-typed variables with B may not work if A's instances become B instances. fds and mts represent, respectively, fields and methods. A primed metavariable, like mts', is a transformed version of the unprimed one, as defined in the **where** clause. A sample of other laws used in this paper are presented in Appendix A.

Law 1 $\langle$*new superclass*$\rangle$

class A **extends** C {
  fds
  mts
}
class B **extends** A { }
CT
	=	**class** A **extends** C {
		  fds
		  mts'
		}
		class B **extends** A { }
		CT'

where
$CT' = CT[\textbf{new}\ A/\textbf{new}\ B]$
$mts' = mts[\textbf{new}\ A/\textbf{new}\ B]$
provided
($\rightarrow$) (1) B is not used in type casts or tests in CT or mts for expressions of type A;
(2) $x := \textbf{new}\ B$ only appears if $type(x) \leq A$;
($\leftarrow$) Variables of type $T \leq A$ are not involved in tests with type B.

In addition to equivalence laws, there are laws for *class refinement* that involve internal representation changes such as addition and removal of private fields. In these laws, simulation is established within a class by a constructor making the coupling invariant true and every method executing on a valid state and resulting in another valid state, as they are based on Morgan's refinement notion [22]. These laws have been proven sound and complete as well, although within a language lacking object references [18]. In order to avoid this limitation in work, we guarantee modular reasoning by using *confinement* as a requirement for programs.

Ownership confinement [23] is a discipline for controlling aliasing in object-oriented languages, restricting access to designated *representation objects* (reps), except through their *owners*, to avoid representation exposure [19]. An owner is a class that maintains representation objects stored in the fields of its objects. Banerjee and Naumann [19] present a number of static analysis rules for ensuring a property called by them *safety*, which is shown in their work to imply confinement. The input is a class table and its division into three sets of classes: *Own* and *Rep*, defining the possibly non-disjoint sets of owner and representation classes, respectively, and *Client* with all other classes. The analysis is modular, as only *Own* and *Rep* code is constrained (with one exception, for **new** commands). They present the rules showed in Table 1, as adopted in this work.

3 Synchronization

Given a specific consistency relationship between object models and programs, we formalize a unidirectional approach in which model refactoring changes are transferred to a consistent program through the application of a *synchronizer*. In order to investigate how this consistency can be maintained throughout refactoring tasks, the consistency relationship is also described.

Table 1. Confinement Rules [19]

1. Public methods declared in *Own* or subclasses cannot return *Rep* types; otherwise, references to internal objects might leak to clients;
2. Methods inherited or declared by *Own* cannot have parameters of *Rep* types; otherwise, non-owner subclasses might have access to *Rep* instances;
3. *Rep* classes cannot inherit any methods from non-*Rep* superclasses; for instance, a method could return **self** to a client, which is highly undesirable;
4. For any field access $e.f$, if e is of type *Own*, it cannot access fields of type *Rep*, unless e is **self**; this rule must be checked only for public fields, as it is guaranteed by type safety for private fields;
5. For assignments `x:= new B` in *Client*, *B* cannot be *Rep* or any of its subclasses; otherwise, these clients would have direct access to *Rep* instances;
6. For method calls `x:= e.m(`$\bar{e}$`)`: (1) if e is a *Client* object, and the call is within *Own* or *Rep* (or subclasses), m cannot have *Rep* parameters (otherwise *Rep* instances could leak); also, (2) if the call is within *Own*, `m` is declared in *Own*, and e is **self**, parameters and return may be *Rep* type. The second case in fact weakens the confinement constraints with a condition that can be detected by static analysis.

3.1 Synchronizers

For object models, we adopt the approach of primitive transformations being composed into refactorings. To each Alloy law from the catalog we associate a *synchronizer* – set of conditional program transformations disciplined by laws of programming – to be applied to a program, making it consistent with the transformed object model. The mechanics of the synchronization is depicted in Figure 1, where OM represents an object model, and P a program. The first step is the application of a model refactoring (a) – in this case, made up of two Alloy laws, X and Y, applied from L-R. Next, each applied law is associated with a synchronizer (depicted as "corresponds" in Figure 1; for instance, `Law X` corresponds to `Sync X`), applied to P. The sequential application of the synchronizers in (b) results in a synchronized program.

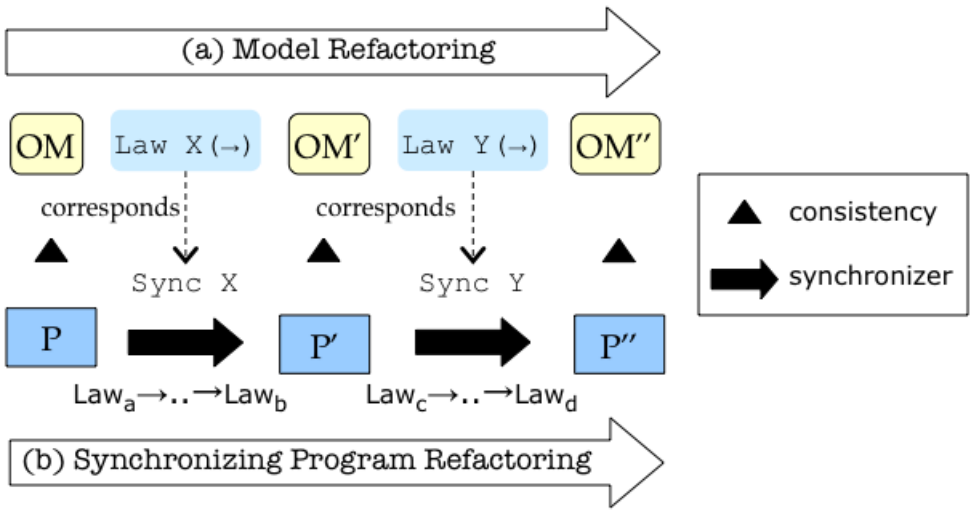


Fig. 1. Model-driven refactoring with synchronizers

A synchronizer carries out program refactoring by applying a sequence of law applications, on the assumption of the consistency relationship defined in the next section. The only preconditions for the application of synchronizers is that the program must have confinement for a subset *Own* of classes and in syntactic and semantic consistency with the previous version of the model. Therefore, the synchronizers are especially conceived to exploit the model invariants that are known to be met by the program; program transformations are specialized with high-level assumptions about the program. In fact, we analyzed each Alloy law and conceived synchronizers for both application directions.

A synchronizer must then exhibit the following characteristics: it rewrites programs for updating corresponding abstractions that were refactored in the object model, and preserves program behavior. Therefore, every synchronizer fulfills a few requirements: its application results in programs that refine the previous version and establish consistency with the refactored model, syntactically and semantically, as explained next. In addition, confinement is maintained.

3.2 Consistency Relationship

A desirable property of object models is abstraction; ideally, they can be implemented by several structurally-distinct programs, as long as the invariants hold during their executions. Structures in the model must be somehow implemented in the program, offering a basis for evaluating whether the modeled constraints are met. We call this correspondence *syntactic consistency*. Given a syntactic consistency relationship, fulfillment of model invariants by the executions of a given program is regarded as *semantic consistency*.

We chose a particular syntactic consistency: there must be one direct class for each signature declared in the model (for simplicity, this specification relies on the equality between names, although a mapping between names could be easily established as well). Also, all supersignatures of a signature must have corresponding superclasses of class S, indicating that more superclasses may be declared in the program, but the modeled hierarchy is maintained. Likewise, every relation is mapped to one field with an exactly matching type, with one additional constraint: relations with single multiplicity (yielding a scalar value) are mapped to single field, whereas relations with set multiplicity must be mapped to collection-type fields. Additional classes, fields or methods can be freely declared in the program.

Regarding semantics, an Alloy model defines valid states for a given system – *interpretations* [16] – that contain mappings of signatures and relation names to sets of *object values*. Object values may be single objects for sets and pairs of objects for relations. We consider the semantics of an object model in Alloy as the set of all valid interpretations satisfying all modeled invariants. Each of these interpretations consists of all valid assignments of values to signatures and the relation names. All modeled invariants – implicit or explicit [16] – are satisfied. Invariants are implicit when they constrain the model but are not declared in facts, such as implicit constraints from **extends**.

For programs, states are formalized as *heaps* of object values, mapping class names to sets of objects and field names to pairs of object values (references). If an object in a heap contains a field storing a null value, no pair of values exists with that object as the first member. The semantics of a program is given by *the set of sequences of*

heaps resulting from all possible execution traces – depending on the possible program inputs.

It would be straightforward to consider all heaps from every execution trace; however, this approach does not truly reflect the real intentions of consistency checking, since some heaps may be acceptably invalid at some well-defined points of the program. We adopt a specification methodology by Barnett et al. [24], in which every object is added a special *validity field*. If this field has a true value, the invariants over its state should hold, and consistency checking is only performed when all objects are valid. This field can only be modified through the use of two special statements, **unpack** and **pack**. The command **unpack** `obj` sets the field to false, while **pack** `obj` does the opposite.

Our semantic consistency regards solely valid heaps (which we call *heaps of interest*). A program is in semantic consistency with a model if, and only if, it is in syntactic consistency, and, for every valid heap from its execution, there is a corresponding interpretation from the semantics of the model.

3.3 Examples of Synchronizers

For each applied Alloy law, two synchronizers are defined. In this paper we show two synchronizers associated with Law 1: introduce and remove subclass. We define synchronizers in a notation for *refinement tactics*, based on the Angel language [25]. Angel constructs are appropriate for describing law applications, with the needed arguments. Tactics may be a simple law applications, with the law name with arguments. A law application may have two possible outcomes: if all provisos are satisfied, then program is transformed. Otherwise, the application of the law fails. For instance, **law** $newSuperclass(U, X, \rightarrow)$ applies Law A (*new superclass*) to the program, with three arguments: the superclass (U), the subclass (X) and the application direction ($\rightarrow$: from L-R). A special atomic tactic, **skip**, always succeeds, leaving the program unchanged.

In order to sequentially composing two tactics, the $t_1; t_2$ construct can be used. Similarly, tactics combined in alternation have the form $(t_1|t_2)$. First, t_1 is applied to the program; if this application is successful, then the composite tactic succeeds. Otherwise t_2 is applied. Finally, if t_2 fails, then the whole tactic aborts (which is a more critical situation than failure). When the tactic contains many choices, the first choice that succeeds is selected. In addition, the language allows us to define pattern matching within a program, with the constraint **applies to**. For instance, **applies to** $cmd[(X)e]$ **do** t applies the t tactic to every command in the program that includes an expression cast with X.

Introduce Subclass Law 1(L-R) introduces a subsignature for one of the declared signatures. This makes `U` abstract according to the modeled invariant (`X=U-S-T`). Here we show the associated synchronizer, that accepts a consistent program. However additional classes can be declared in the program hierarchy.

Tactic $introduceSubclass(X, U : Class)$
  (**law** $rename(X, X')$ | **skip**);
  law $classElimination(\textbf{setExtends}(X, U), \leftarrow)$;
  law $newSuperclass(U, X, \rightarrow)$;
end

The trivial law $rename(X, X')$ renames the X class. If it fails (for the case in which the class is not declared), nothing happens (**skip**). If class `X` is already present, it is freely renamed to `X'`, because `X` is considered in this case an *implementation detail* that was not modeled. This action does not have impact on the consistency, as renamed declarations are not in the model. Next, the synchronizer introduces the new class `X` as a direct subclass of `U`, with Law *class elimination* (Appendix A). **setExtends** makes X a direct subclass of U. Other subsignatures of U will be declared as classes, although their inheritance relationship with `U` may be indirect – implementation-only subclasses are allowed. Finally, every `U` object creation in the entire program is replaced by `X` instantiations, by Law 1 from L-R (**`[new X/new U]`**).

The synchronizer provides evidence on how model-driven refactoring can improve tool support for refactoring, since the semantic properties from object models can aid refactoring automation. In *introduceSubclass*, the program is refactored to a specific configuration of U objects, making them X instances. This information cannot be obtained solely from the source code, then introducing a plain subclass would not include the changes applied by the synchronizer.

Remove Subclass The opposite transformation given by Law 1 removes subsignature `X` assuming the invariant (`X=U-S-T`); `S` and `T` become the `U`'s only subsignatures. The synchronizer removes the corresponding `X` class, although, differently from the model, the program class may declare fields and methods, and may have implementation-only subclasses. These implementation details must be rearranged, as showed in Figure 2.

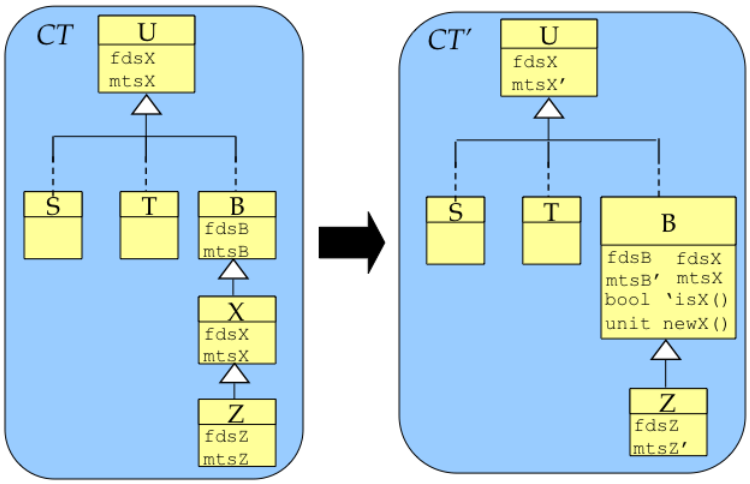


Fig. 2. Remove subclass synchronizer

The following definition uses several auxiliary tactics, which are informally shown in this paper; their complete definitions are described in [17].

Tactic $removeSubclass(X : Class)$
  tactic $moveUpFields(X)$;

```
    tactic moveUpMethods(X);
    tactic changeDeclarationsTypetoSuper(X);
    applies to cmd[(X)e] do law eliminateCastExpressions(cmd[(X)e], →);

    tactic eliminateTypeTests(X, "bool isX(){ result := self is X }");
    tactic eliminateNew(X);
    law changeSuperfromEmptyToImmediateSuperclass(immedSubs(X),
      super(X), →);
    law classElimination(X, →);
end
```

The auxiliary tactic *moveUpFields* pulls up the fields declared in `X` to the immediate superclass. If the target superclass has any other subclass declaring the moved field, the tactic moves two or more fields with the same name to the superclass in one step; if it is not the case, the single field is moved to the superclass, with Law *move field to superclass* in Appendix A). Next, `X`'s methods are pulled up as well, with the auxiliary tactic *moveUpMethods*. In this case, the synchronizer must deal with two cases: redefined and non-redefined methods:

- The redefined methods are removed from `X` and the corresponding method body in the superclass is modified with an **if** command that adds the body of the moved method, using Law *move redefined method to superclass*. Also within the tactic, **super** method calls are eliminated by inlining from **object** to `X`, top-down in the hierarchy (Law *eliminate super* in Appendix A); for this, all private fields in this hierarchy are first made public;
- The non-redefined methods must be copied to other subclasses of B, with an empty body, so no type errors occur with the new method in B.

After removing its fields and methods, `X` is replaced by its superclass on declarations over the program, with *changeDeclarationsTypetoSuper*; in this tactic, Law *change field type* and analogous laws are applied. Next, In the main tactic, casts to `X` are removed with another law (*eliminate casts of expressions*). Consecutively, *eliminateTypeTests* removes type tests involving `X`, with the following steps:

1. A boolean method `isX` is declared within `B` and its subclasses. This method is a surrogate for the type tests that are going to be eliminated.
   The method body returns the value of testing **self** with `X` and subclasses (in this example, `Z`);

```
class B { ..
  bool isX(){ result:= self is X ∨ self is Z} }
```

2. Every occurrence of `x is X` must be replaced by a special statement, a *parameterized command* [18]. A parameterized command of the form `test:= (result:= x is X)` is then be replaced by a method call to `isX`. For avoiding null pointer errors, we introduce an **if** statement for ensuring that the expression being tested is not null;

```
if (x=null) then test:= false else test:= x.isX()
```

3. Additional changes are performed for backing up the `isX` test. Field `type` is introduced and initializations to this field are added to `X`'s constructor and every constructor in `X`'s subclasses;

```
class X extends B {   constr { ..; self.type:= "X"}  }
class Z extends X {   constr { ..; self.type:= "Z"}  }
```

4. Within overriding `isX` implementations, expression **self is** `X` is replaced with the equivalent expression **self**`.type = "X"`;

Regarding constructors, `X` declares a constructor that must be replaced, as every **new** `X` will be rewritten as **new** `B`. Hereafter, we consider a command of type `x:=` **newX** to be a syntactic sugar for the following sequential composition: `x:=` **new'** `X; x.newX()`, in which **new'** is the regular instantiation of an object, whose reference is assigned to `x`. It is followed by a call to `newX`, a method of class `X` containing the actual constructor body, used for initializing fields. After defining this replacement for every `X` instantiation, the synchronizer moves `newX` to the superclass `B` (which contains the initialization for the `type` field). After this, the **new'** `X` commands can be replaced by **new'** `B` commands in the whole program, with Law A. An excerpt of the result can be seen next.

```
class B { .. string type; ..
  bool isX() { result:= self.type="X" ∨ self is Z }
  unit newX() {     { .. self.type:= "X"} }  }
class X extends B { } ..
B x:= new' B; x.newX(); ..
```

Finally, the **extends** clause of `X`'s subclasses, then `X` can be eliminated. In general, automated refactorings only remove subclasses when they are not used anywhere in the program. In contrast, this synchronizer can prepare programs when removal of the given subclass is desirable. It replaces all uses of this subclass by the correspondents given by an invariant (stating that class `U` is abstract).

The defined synchronizers follow the correspondence in Table 2. Other laws of modeling do not have corresponding synchronizers, as they deal with syntactic sugar in the model, which does not affect the syntactic consistency.

Table 2. Synchronizers corresponding to Alloy laws

Alloy Law	synchronizer →	synchronizer ←
1.Introduce Relation	*introduceField*	*removeField*
2.Introduce Subsignature	*introduceSubclass*	*removeSubclass*
3.Introduce Signature	*introduceClass*	*removeClass*
4.Introduce Generalization	*introduceSuperclass*	*removeSuperclass*
5.Split Relation	*splitField*	*removeIndirectReference*
6.Remove Lone Relation	*fromOptionalToSetField*	*fromSetToOptionalField*
7.Remove One Relation	*fromSingleToSetField*	*fromSetToSingleField*

4 Soundness

A soundness theorem is established for synchronizers. The rationale behind this theorem is the set of conditions for a sound synchronized refactoring. Given these conditions, the compromises for automating the involved transformations can be analyzed in depth, showing issues that will recur in several MDD contexts. Thus, proofs for synchronizers constitutes the core of our approach. Sound synchronizers depend on the defined consistency relationship and two additional properties: (1) they must express refinements and (2) preserve program confinement.

Theorem 1 is defined for an arbitrary object model (OM), and an arbitrary program (P), in which the application of a law to OM results in OM', and a synchronizer applied to P results in P'. We define additional predicates from law definitions: *Refines(P',P)*, in which the second argument refines the first, and *Confined(P)*, stating that P satisfies the static analysis confinement rules from Table 1, for a subset *Own* of the class table. $premises(OM, OM', P)$ states the conditions before the application of a synchronizer, as defined in Appendix B – an Alloy law applied to OM results in OM', and consistency and confinement constraints apply to OM and P.

Theorem 1. $\forall\ OM, OM', P, P' \bullet premises(OM, OM', P') \Rightarrow syntConsistency(OM', P') \wedge Confined(P') \wedge Refines(P', P) \wedge semanticConsistency(OM', P')$

The proof of each synchronizer is split in four *supporting lemmas*. The theorem's meta-variables OM, OM', P and P' are concretized for each synchronizer.

Lemma 1. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow syntConsistency(OM', P')$
Lemma 2. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow Confined(P')$
Lemma 3. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow Refines(P', P)$
Lemma 4. $\forall OM, OM', P, P' \bullet premises(OM, OM', P) \Rightarrow semanticConsistency(OM', P')$

We proved the synchronizers listed in Table 2, as detailed in [17]; for illustration, here we present the proof for *removeSubclass*. Model and program definitions for the proof are shown in Appendix B. Assuming $premises(OM, OM', P)$, we now prove the four previous lemmas.

Proof for Lemma 1. Since no relations are added or removed, the mapping between relations and fields is unchanged. Regarding signatures, X is removed, which is the only class that is removed from the program, establishing the conformance. The hierarchy remains unchanged. Thus $syntConsistency(OM', P')$ follows from the premises.

Proof for Lemma 2. By case analysis on P' for the six static analysis rules of confinement from Table 1. For each rule, we justify its maintenance in terms of the premises and P'. In this case, `X` $\notin$ *Rep*.

1. Class `B` is the only one to receive new methods. If `B` $\in$ *Own*, then from $premise(OM, OM', P)$ methods previously in `X` could never have *Rep* return types;
2. No inherited methods are added, thus from $premise(OM, OM', P)$ no inherited methods have *Rep* parameters;
3. Same as above, thus from $premise(OM, OM', P)$, *Rep* classes do not inherit methods from non-*Rep* classes;
4. No public fields of *Own* classes are used outside their declaring module, thus from $premise(OM, OM', P)$ no `e.f` is seen, unless `e` is **self**;
5. From premise, if `x:=new X` was outside *Own* classes, `X` $\notin$ *Rep*. Assuming the command is outside *Own*, it is impossible to have `X` $\notin$ *Rep* and `B` $\in$ *Rep*, since all subclasses of *Rep* classes are also included. Therefore, property is maintained;
6. From premise, `e.m(..)` within *Own* or *Rep* does not have *Rep* parameters; no changes in parameters or *Rep* are made, so property is maintained.

Proof for Lemma 3. Since the synchronizer steps are exclusively defined as law applications and class refinement [18], P refines P'. Some of the applied laws can be seen in Appendix A[3].

[3] The laws have been proven semantics preserving for a programming language without references [18]. However, we have proven that these laws are still correct in the presence of confinement [17]

Proof for Lemma 4. First, any interpretation can be reduced without the mappings to X. This is possible since values of X cannot be interpreted alone, from the given invariant $X = U - S - T$. Consequently, semantics of OM' is the set of interpretations from the semantics of OM, with mapping from X removed. Likewise, for P and P', the valid heaps of P' are the valid heaps of P reduced in mappings from the X class; this conclusion is implied from the *invariant that is assumed in the program.* Also, the changed commands (type casts and tests) do not add or remove new possible heaps; all X instances that are not B instances are still mapped by U in the heap.

5 Discussion

In this section, the contributions and limitations of our synchronization model are discussed, especially topics related to automation and quality of refactorings.

5.1 Invariants as Basis for Refactoring Automation

The practice of refactoring has been improved by supporting tools, avoiding manual work and increasing trust on semantics preservation. Usually a catalog of refactorings is offered, from which developers can choose the desired transformation for the problem in context. These automated refactorings present *preconditions* that are checked against the code subject to refactoring, in order to ensure correctness. While being effective to ensure safe refactorings – at least in theory – it leads to prevention of refactoring on programs that would be eligible if some *semantic assumptions* about the program behavior were considered.

Semantic assumptions about the program can be provided by object models, then synchronizers exploit invariants to increase the applicability of some automated refactorings. Transformations based on these invariants can be applied to programs that would not be eligible for refactoring using the current tools. When removing a subclass, for example, the synchronizer *assumes* the invariant $X = U - S - T$ as true in every reachable program state outside a **unpack/pack** block. In this case, the subclass X can be removed, given that U is an abstract class.

Certainly there are several open questions. For instance, it is not clear how invariants will be automatically identified by a refactoring tool for the application of specific refactorings. Our intuition is that catalogs of program refactorings could be extended with improvements based on invariants, conditionally applied based on a set of invariants. Also, consistency is the enabling condition of synchronizers, which is hard to verify in some scenarios (especially semantic consistency).

5.2 Quality of refactorings

With formal synchronizers, quality factors such as cohesion and legibility still requires some improvement in the resulting program. For instance, the successive application of *removeSubclass* may result in numerous implementations of methods for eliminating type tests, which is clearly amenable to simplification. Therefore, additional refactoring might be necessary, such as inlining these calls and removing methods. These transformations *are also formalizable as laws of programming.* Although theoretically feasible, these law applications could not be automatically applied in the formal model, since

our initial assumption is that each synchronizer is recorded and independently applied in order, disregarding the composed refactoring that was applied to the model.

In this scenario, we envisage *developer feedback* as a possible answer to this challenge, in addition to *complementary synchronizers*. In this case, the application of the model refactoring could bring additional information that is then applied in the program refactoring, according to feedback from the developer of a supporting tool. If the developer agrees, a complementary synchronizer, containing the additional law applications, is automatically applied. The outcome of the complementary synchronizer is an improved program, yet still conforming, syntactically and semantically, to the refactored model. This additional synchronizer is conditional to the employment of the specific model refactoring for introducing several fields at the same time, not independent of isolated synchronizers.

Furthermore, due to the independence of synchronizer applications, information regarding the composed model refactoring as a whole is not used for improving the resulting program, often missing the refactoring's original goal. An alternative for dealing with this problem is to refactor programs exclusively based on *the refactoring's initial and final models*, ignoring the intermediate law applications. For such approach, we see two possible alternatives: (1) a fixed catalog of major refactorings, whose corresponding synchronizers would be tailored for these refactorings; and (2) automatically generate a synchronizer from the applied model refactoring. The first option seems to be easier, but likely to end up with the same issue. The second option is visibly more complex.

5.3 Consistency and Synchronizers

The required consistency relationship was adjusted during the formalization of synchronizers. Several choices have been considered and this scenario allowed us to gather evidences on how the chosen consistency affects the final results of model-driven program refactorings.

The rule of thumb states that, the more abstract are the models, the looser (different possible implementations for the same object model) is the syntactic consistency relationship. The syntactic mappings between model and program declarations drive the freedom of implementation for modeled signatures and relations. At the end, we adopted a tighter consistency relationship than initially expected: signatures must be implemented as classes and relations as fields in the corresponding class. Nevertheless, the required consistency relationship still preserves some abstraction: methods and additional classes can be freely implemented, and hierarchies can contain more classes than modeled. In addition, the modeled signatures and relations must be implemented in a uniform way, so the synchronization is still compelling for the user. As refactoring is a structural modification, the declarations in the model must be reflected in the source code for desired transformation; otherwise, the task would be rather pointless.

In addition, the looser is the syntactic consistency, the more complex become the program transformations needed to refactor the program. When giving more freedom of implementation to a specific model declaration, synchronizers must consider every implementation option for this declaration, in order to achieve automation. In this context, synchronizers must be more elaborate, which often *clutters the program, decreasing quality*. This is certainly a trade-off for any synchronization approach.

6 Related Work

Co-evolution between models and programs is dealt with by several related approaches. For instance, Harrison et al. [9] 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 decision is consonant to our choices, as the relationship between model and source code avoids round-tripping. Their consistency relationship is more flexible; we formalized a more strict structural similitude between the artifacts. No details are offered on how their consistency mappings will consistently evolve.

The concept of coupled transformation in Lammel's overview [10] has a close correspondence to our approach. Coupled transformations occur when "two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such global consistency is reestablished" [10], which is the scenario for model-driven refactoring. This type of synchronization seem to fit into the "symmetric reconciliation" category, in which two distinct transformations – for model and program – are defined for a given consistency relationship, adapting changes according to the specific level of abstraction for which they are defined.

Bidirectional model transformations (*bx*) [11,12] have the purpose of formalizing synchronization between changed artifacts during the software life cycle (in this approach, model is a comprehensive concept, which includes programs). The proposal includes an abstract definition of synchronizers, which may even be bidirectional (updating both artifacts). Several concepts are similarly formulated – such as unidirectional synchronizers – but no particular approaches of *bx* are defined for object models and programs. Therefore, our results could be confirmed in such scenario by concretizing *bx*.

The Harmony tool [13], for instance, is based on the concept of *bx*. The authors introduce the concept of relational lenses, which are pairs of transformation functions, namely *get* and *putback*, between source and target artifacts. The *get* function transforms a source artifact into a target artifact. Updates can be performed on the target artifacts, then an updated source artifact can be obtained with the *putback* function, with information from the original source artifact and the updated target artifact. Analogously, in our theory *get* is similar to the required consistency relationship, although we avoid generation of artifacts. The source artifact can be a program, and the target can be an object model.

Another related study is carried out by Antiewicz and Czarnecki [26], which formally defines several synchronization alternatives between software artifacts. Their synchronization definitions are applied with the help of formal operators. Several elements are common with our approach, for instance developer feedback for automation and related and independent transformations. Since we focus on a specific type of synchronization (object models to programs), our theory is able to reveal detailed issues about consistency and transformation.

7 Conclusions

In this paper, we formalized a synchronization theory from object model refactoring to object-oriented programs. The theory is backed by a formal infrastructure of primitive transformations proved to be semantics preserving, both for object models

and programs, and a specific consistency relationship. Synchronizers are formalized as a sequence of primitive program transformations, explicitly avoiding generation of programs from object models. The investigation unveils several issues concerning consistency, refactoring automation and behavior preservation and quality, providing evidence over the challenges that effective MDD methodologies will face in order to support evolution. Potential improvements for refactoring tools are identified, since the semantic properties from object models can aid refactoring automation. In our synchronizers the invariants expressed in the object model offer semantic information to extend its automatic refactoring capabilities.

The level of abstraction is a key aspect. First, useful model refactoring requires that the main structures 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. Assumptions include reliance on the maturity of consistency checking tool support in practice and a closed-world context in which we have access to the full source code of a program.

The theory described in this paper is language specific, although the formalization is amenable to adaptation to other object-oriented languages. In addition, our approach supports only refactoring; dealing with generic evolution in MDD is a challenge for future research. A potential solution might rely on primitive transformations for standard evolution, and model invariants could be used to transform programs accordingly.

Acknowledgment

We'd like to thank Augusto Sampaio, Alexandre Mota, Ana Cristina de Melo, Marcel Oliveira, Juliano Iyoda, and all anonymous reviewers for the relevant comments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq, grant 573964/2008-4.

References

1. Fowler, M.: Refactoring—Improving the Design of Existing Code. Addison Wesley (1999)
2. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, UIUC (1992)
3. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press (2006)
4. Liskov, B., Guttag, J.: Program Development in Java. Addison Wesley (2001)
5. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) 126–139
6. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley (2004)
7. France, R.B., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: FOSE '07. (2007) 37–54
8. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: Theory and Practice of Model Transformations. (2008) 31–45
9. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML Designs to Java. In: Proceedings of OOPSLA 2000. (2000) 178–187
10. Lammel, R.: Coupled software transformations. (In: SET 2004) 31–35
11. Diskin, Z.: Algebraic models for bidirectional model synchronization. In: MoDELS 2008. (2008) 21–36

[4] www.ines.org.br

12. Stevens, P.: A Landscape of Bidirectional Model Transformations. GTTSE 2007 (2008) 408–424
13. Bohannon, A., Pierce, B., Vaughan, J.: Relational lenses: a language for updatable views. In: PODS 2006. (2006) 338–347
14. Massoni, T., Gheyi, R., Borba, P.: Formal model-driven program refactoring. In: FASE-ETAPS 2008. (2008) 362–376
15. Massoni, T., Gheyi, R., Borba, P.: An approach to invariant-based program refactoring. In: Setra Workshop 2006. (2006) 91–101
16. Gheyi, R., Massoni, T., Borba, P.: A static semantics for alloy and its impact in refactorings. ENTCS **184** (2007) 209–233
17. Massoni, T.: A Model-Driven Approach to Formal Refactoring. PhD thesis, UFPE (2008)
18. Borba, P., Sampaio, A., Cavalcanti, A., Cornélio, M.: Algebraic Reasoning for Object-Oriented Programming. Science of Computer Programming **52** (2004) 53–100
19. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. Journal of the ACM **52**(6) (2005) 894–960
20. Gheyi, R., Massoni, T., Borba, P.: An abstract equivalence notion for object models. ENTCS **130** (2005) 3–21
21. Gheyi, R., Massoni, T., Borba, P.: A Complete Set of Object Modeling Laws for Alloy. In: SBMF. (2009) 204–219
22. Morgan, C.: Programming from Specifications. Second edn. Prentice Hall (1998)
23. Clarke, D.: Object Ownership and Containment. PhD thesis, UNSW (2001)
24. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of Object-Oriented Programs with Invariants. Journal of Object Technology **3**(6) (2004) 27–56
25. Martin, A.: Machine-Assisted Theorem-Proving for Software Engineering. PhD thesis, Penbroke College (1994)
26. Antkiewicz, M., Czarnecki, K.: Design space of heterogeneous synchronization. GTTSE, Braga, Portugal (2008) 3–46

A Additional Laws of Programming

Law 2 ⟨*class elimination*⟩

$$CT\ cd_1\ =\ CT$$

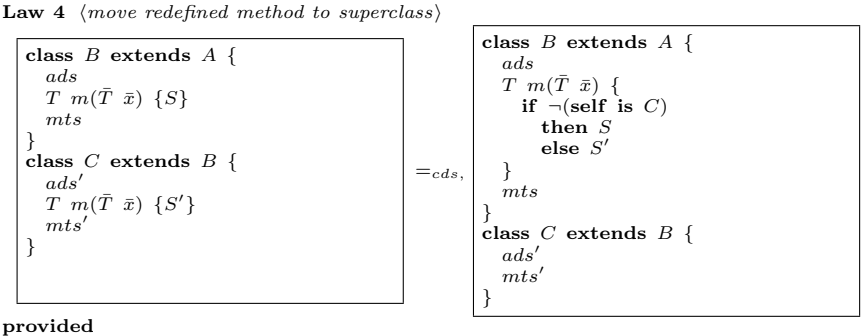
provided
(↪) $cd_1 \neq$ `Main`;
(→) $name(cd_1)$ is not used in CT;
(←) (1) cd_1 is a distinct name;
(2) Field, method and superclass types in cd_1 are declared in CT.

Law 3 ⟨*move field to superclass*⟩

class B **extends** $A\{$ fds mts $\}$ **class** C **extends** $B\{$ **pub** $T\ a; fds'$ mts' $\}$	=	**class** B **extends** $A\{$ **pub** $T\ a; fds$ mts $\}$ **class** C **extends** $B\{$ fds' mts' $\}$	**provided**

(→) The field name a is not declared by the subclasses of B in CT;
(←) $D.a$, for any $D \leq B$ and $D \not\leq C$, does not appear in CT, c, mts, or mts'.

Law 4 ⟨*move redefined method to superclass*⟩



provided
(↔) (1) **super** and private fields do not appear in S'; (2) **super**.m do not appear in mts';
(→) S' does not contain uncast ocurrences of **self**;
(←) m is not declared in mts'.

Law 5 ⟨*eliminate* **super**⟩
Consider that CDS is a set of two class declarations as follows.

class B **extends** A{
fds
$T\ m\ (\bar{T}\ \bar{x})\ \{\ pc\ \}$
mts
}

class C **extends** B{
fds'
mts'
}
Then we have that

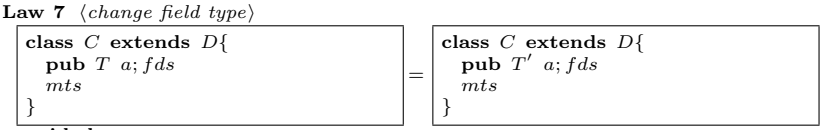
$$CT\ CDS,\ C\triangleright\ \textbf{super}.m = pc$$

provided
(→) **super** and the private fields in fds do not appear in pc.

Law 6 ⟨*eliminate cast of expressions*⟩
If $CT, A\triangleright le : B$, $e : B'$, $C \leq B'$ and $B' \leq B$, then

$$CT, A\triangleright le := (C)e = \{e\ \textbf{is}\ C\}le := e$$

Law 7 ⟨*change field type*⟩



provided
(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of mts, CT and c is cast with T or any subtype of T declared in CT.
(←) (1) every expression assigned to a, in mts, CT and c, is of type T or any subtype of T;
(2) every use of a as return value is for a corresponding formal parameter of type T or any subtype of T.[a]

[a] Almost identical laws are defined for parameters, return values and local variables.

B Proof Definitions

B.1 Premises of the main theorem

$premises(OM, OM', P') \;=\; syntConsistency(OM, P) \;\wedge\; Confined(P) \;\wedge$
$\quad Refines(OM', OM) \;\wedge\; semanticConsistency(OM, P)$

The formal definition of $Refines(OM', OM)$ is fully shown in related work [16]. In summary, it establishes that the refined object model's instances (assignments of values to signatures and relations) make a *subset* of the more abstract model's instances. The whole catalog of Alloy laws is proven to respect this refinement notion.

B.2 Definitions for proving *removeSubclass*

Program templates are assembled directly from the synchronizer; in those, classes may extend another class. Let OM, OM' be any two object models and P, P' two programs as follows. The **where** clause declaratively define the program constructs that are rewritten by the synchronizer. Special functions are used in quantifications, such as $hierarchy(X, \mathbf{object})$, which yields a set of classes from X to **object**, hierarchically. Similarly, $meths$ yields all methods from a list of classes, and $body(m)$ gives the commands within method m. A boolean expression within braces ($\{exp.isX()\}$) indicates an assertion, that must be fulfilled by the execution before the succeeding command.

OM

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

OM'

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

P

```
CT
class U [extends]{..mtsU}
class S [extends]{..mtsS}
class T [extends]{..mtsT}
class B [extends]{
  adsB;
  mtsB}
class X extends B{
  adsX;
  mtsX;
  constr {c}
}
class Z extends X{
  ..mtsZ'}
  constr {b}
}
```

P'

```
CT'
class U [extends]{..mtsU'}
class S [extends]{..mtsS'}
class T [extends]{..mtsT'}
class B [extends]{
  adsB; adsX;
  pub String type;
  mtsB'; mtsX;
  bool isX(){
    result := (self.type = "X" ∨
self.type = "Z")
  }
  void newX(){
    c; self.type := "X"
  }
}
class Z extends B{
  ..mtsZ';
  constr {b; self.type := "Z" }
}
```

where:
$S, T, B \leq U$;
$CT' = rewrite(CT)$; the $rewrite$ function performs the following substitutions:

[**unpack** $x; x :=$ **new** $B; x.newX();$ **pack** x / $x :=$ **new** X];
[$\{exp.isX()\}cmd[exp]/cmd[(X)exp]$];
[$exp.isX()/exp$ **is** X];
[B id/X id], and id is any declared identifier;
$A' = A[pub\ Type\ f/pri\ Type\ f]|A \in hierarchy(X, object)$;
$mt' = mt[body(m)/super.m()]$,$mt \in meths(hierarchy(X, object))$,
and m is a method called within mt;
$mtB' = mtB$[**if** $\neg$(**self is** X) **then** b **else** b'],
$mtB \in name(mtsX \cap mtsB), b = body(B.m), b' = body(X.m)$