A Formal Framework for Establishing Conformance between Object Models and Object-Oriented Programs

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

[1]Informatics Center — Federal University of Pernambuco
PO Box 7851 – 50.732-970 Recife, PE

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

Abstract. *Conformance between structural models and their implementations is usually simplified in practice. This is not appropriate to accommodate the usual freedom of implementation for abstract concepts. In this paper, we propose a formal framework for defining conformance between object models and object-oriented programs. The framework is instantiated by providing syntactic mapping rules between model and program elements; from these rules, semantic conformance checking is enabled. The framework includes the notion of heaps of interest, which may remove unstable program execution states for a less strict conformance checking. We evaluate the framework in establishing a conformance notion for a model-driven approach for program refactoring.*

Resumo. *Conformidade entre modelos estruturais e suas implementações é quase sempre direta e simplificada na prática. Este cenário não é apropriado para acomodar a liberdade usual de implementação para conceitos abstratos. Neste artigo, apresentamos um framework formal para definição de conformidade entre modelos de objetos e programas orientados a objetos. O framework é instanciado pelo fornecimento de regras de mapeamento sintático entre elementos do modelo e do programa; a partir destas regras, conformidade semântica pode ser verificada. Este framework inclui uma noção de heaps de interesse, que pode eliminar estados instáveis na execução de um programa possibilitando uma conformidade semântica menos restrita. O framework é avaliado em termos de uma noção de conformidade útil para uma abordagem de refatoramento de programas dirigida por modelos.*

1. Introduction

Structural models depicting domain concepts, relations and invariants in an object-oriented fashion are called *object models*. Languages for expressing object models include Alloy [Jackson 2006] or UML [Booch et al. 1999] class diagrams annotated with invariants in the Object Constraint Language (OCL) [Warmer et al. 2003]. Object models are abstract; they can be implemented by several structurally-different programs, with different behavior, as long as the invariants hold during their executions. In order to clearly characterize when a program maintains the structural invariants from the object model, a mapping from model to program elements must be defined. *Conformance* is then characterized by two perspectives: syntactic conformance, if a model and program satisfy some syntactic mappings between their declared elements, and semantic conformance, which defines when a program follows all modeled invariants throughout its execution.

Object model elements (sets and relations) are abstract, which will be given diverse concrete representations in the program, in terms of object-oriented (OO) programming language constructs (classes and attributes). In practice, however, syntactic conformance between object models and their implementations are usually simplified – for instance, in reverse engineering and semantic conformance checking tools. In this context, model elements are directly declared in the program – each set is implemented as a class, for example –, restraining the freedom of implementation provided by abstraction (examples of syntactic mapping rules are described in Section 2).

In this paper, we describe a formal framework for defining syntactic conformance between object models and OO programs, allowing reasoning on semantic conformance. It supports independence of model and program semantics, by relying upon intermediate representations of *instances* and *heaps* (respectively, model and program states). Our definitions were encoded in the Prototype Verification System (PVS) [Owre et al. 1992] (showed in Section 3), whose interactive type checking helped us in finding specification errors. The framework is instantiated by providing syntactic mapping rules between model and program elements; if, for a particular pair model-program, syntactic conformance is confirmed, then a *mapping formula* is automatically provided, enabling semantic conformance checking. From that formula, the framework is used to yield, from the reachable program states, whether a program is in semantic conformance with the object model.

The framework further includes the notion of *heaps of interest*. This is useful for defining at which execution points states should be considered for conformance checking. For instance, invariants may be required to be valid only after object initializations, and on entry and exit of public methods. Failure in fulfilling the invariants at other locations should not invalidate conformance. The selection of heaps of interest may be based on the directives proposed by previous work on verification of OO invariants [Barnett et al. 2004].

For evaluating the framework, we instantiated a conformance notion applied for proving soundness of model-driven refactoring. More specifically, a sequence of behavior-preserving program transformations must preserve conformance between refactored models and programs.

Accordingly, we summarize the contributions of this paper as follows:

- A framework for establishing conformance notions, instantiated in terms of syntactic mapping rules between object model and program elements (Section 4). These mappings are the basis for a formalization of semantic conformance (Section 5);
- A formal definition for heaps of interest, in terms of a filter over the program semantics, in Section 6;
- Evaluation of the framework for proving conformance preservation in model-driven refactorings (Section 7).

2. Motivating Examples

Usually, a straightforward syntactic conformance consists in every modeled set or relation being mapped to a single class or attribute, respectively. However, several possible conformance notions may be useful in practice, as showed in this section.

Figure 1 shows a partial object model for concepts of a banking domain, graphically represented by a UML class diagram. Each box in our representation defines a set of objects, and the arrows are relations, indicating how objects of a set are related to objects in other sets. Also, two invariants are defined, in a notation similar to first-order logic: (1) there are no overlapping accounts for two distinct (`disj`) customers, and (2) every checking account is related to exactly one credit card. The join operator (.), denotes the standard relational composition, while # denotes set cardinality. Invariant (2) above could be easily written as relation multiplicities; they are expanded into invariants for making the explanation of semantic conformance more uniform.

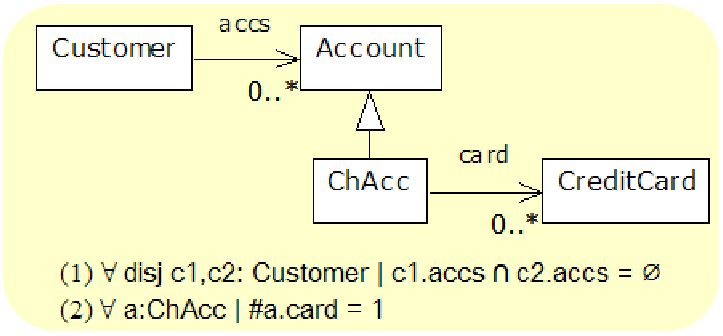
Customer
accs
0..*
Account
ChAcc
card
0..*
CreditCard
(1) ∀ disj c1,c2: Customer | c1.accs ∩ c2.accs = ∅
(2) ∀ a:ChAcc | #a.card = 1

Figure 1. Object model for a banking example.

The following Java fragments represent implementations of the model in Figure 1. First, we define a program using the simplest syntactic mapping.

Mapping 1. In the next fragment, sets are directly implemented as single classes. The attributes implement the modeled relations – `accs` as an array of accounts, card a single variable. The subtyping relationship in the model is implemented with subclassing.

```
class Customer {
  Account [] accs = new Account[1000]; int next = 0;...
  void addAccount(Account a) { accs[next++] = a; }...
}
public class ChAcc extends Account {
  Card card = new CreditCard();
  void changeCard(CreditCard c){ this.card= c; } ...
}
class Account { ... }
class CreditCard { ... }
```

Concerning semantic conformance, the main program showed next presents simple behavior that exercises the above declarations, creating two customers with different types of accounts. The program (classes plus main) is in semantic conformance if, and only if, it fulfills all model invariants throughout its execution.

```
Account a1 = new Account();
ChAcc a2 = new ChAcc();
Customer c1 = new Customer(); Customer c2 = new Customer();
c1.addAccount(a1);
c2.addAccount(a2);
a2.changeCard(new CreditCard()); ...
```

Mapping 1 favors the manipulation of those constructs by CASE tools, especially for code generation and reverse engineering. For instance, a skeleton declaration for the `Customer` can be easily generated from the object model, including the `accs` attribute; on the other hand, the object model sets and relations can also be generated from the source code with minimal user intervention. Although useful for implementations, Mapping 1 is too restrictive. Not so direct implementations are usually seen in practice, as illustrated in the following types.

Mapping 2. An alternative syntactic mapping is added for relations – *collection attributes* – as exemplified by the `List` object holding accounts (`accs`). The `add` method includes new elements in the list. In fact, two alternative implementations for relations may even be allowed, as seen with the `card` single attribute in the following fragment.

```
class Customer {
  List accs = new ArrayList(); ...
  void addAccount(Account a) { accs.add(a); } ...
}
class ChAcc extends Account {
  CreditCard card = new CreditCard();
  void changeCard(CreditCard c){ this.card= c; } ...
}
```

Another example illustrates a common alternative for implementing subtypes that does not involve inheritance.

Mapping 3. The subtype relationship from the object model may be alternatively mapped to an attribute in the original class, defining the type of each object. This is done for `ChAcc` in the following Java fragment.

```
class Account {
  enum Types { CHECKING };
  Types type;
  public Account(Types type) { this.type = type; } ...
  CreditCard card = new List();
  void changeCard(CreditCard c) {
    if (this.type == Types.CHECKING) this.card.set(0,c);
    else //error!
  } ... }
```

Mappings 2 and 3 limit the application of CASE tools, since they require more intricate syntactic mapping rules that most tools do not support. For instance, a complex algorithm is required to detect inheritance in the implementation of `Account`; similar conclusions are drawn in other approaches dealing with relations [Guéhéneuc and Albin-Amiot 2004, Harrison et al. 2000]. Alternatively, a tool could allow users to define custom mappings for syntactic conformance. Well-known tools, such as Rational Software Architect [Rational Software 2006] and Poseidon [Gentleware 2005] still offer the traditional notion for code generation and reverse engineering. This scenario usually results in rather concrete reverse-engineered models, cluttered with implementation details, hence losing abstraction.

Regarding semantic conformance, it is much harder to characterize that model invariants hold in program heaps, when the program presents such disparate elements implementing model elements. The mapping rules between model and program elements must be applied in this setting, in order to correctly relate values from both worlds and check conformance. Several conformance notions can be useful in practice; offering a precise definition of those relations may be an aid for tool supporting several software engineering tasks. This is the aim of our solution.

3. PVS Overview

The Prototype Verification System (PVS) provides mechanized support for formal specification and verification [Owre et al. 1992]. The PVS system contains a specification language, based on simply typed higher-order logic, and a prover. Each specification consists of a collection of theories. Each theory may introduce types, variables, constants, and may introduce axioms, definitions and theorems associated with the theory. Specifications are strongly typed, meaning that every expression has an associated type.

Suppose part of a banking system in PVS, on which each bank contains a set of accounts, and each account has an owner and a balance. Next, we declare a theory named `BankingSystem` that declares two uninterpreted types (`Bank` and `Person`), representing sets of banks and persons, and a record type denoting an account. An uninterpreted type imposes no assumptions on implementations of the specification, contrasting with interpreted types such as `int`, which imposes all axioms of the integer numbers. Record types, such as `Account`, impose an assumption that it is empty if any of its components types is empty, since the resulting type is given by the cartesian products of their constituents. The `owner` and `balance` are fields of `Account`, denoting the account's owner and its balance, respectively.

```
BankingSystem: THEORY
BEGIN
  Bank: TYPE
  Person: TYPE
  Account: TYPE = [# owner: Person, balance: int #]
```

In PVS, we can also declare function types. Next, we declare two functions types (mathematical relation and function, respectively). The first one just declares the `accounts`'s type, establishing that each bank relates to a set of accounts. The `withdraw` function declares the withdraw operation and defines the associated mapping.

```
  accounts: [Bank -> set[Account]]
  withdraw(acc: Account, amount: int): Account =
    acc WITH [balance := (balance(acc)-amount)]
```

The `balance(acc)` expression denotes the balance of the `acc` account. We can use these fields as predicates.

4. Syntactic Mapping for Conformance Notions

In this section we introduce the formal framework definitions in PVS with emphasis on syntactic conformance. The syntactic correspondence between model and program elements is a *hot spot* of the framework, as instantiated for particular conformance notions. Section 4.1 provide the building blocks for defining syntactic mappings. Then in Section 4.2, we show how the syntactic correspondence between model and program is formalized.

4.1. Basic Syntactic Definitions

A formal definition of models and programs is given in the following PVS fragment. These types can be seen as interfaces of our framework, regarding, in any language, object models declaring sets and relations, and programs declaring classes and attributes, given by the indicated functions.

```
Model,Program: TYPE
sets: [Model -> set[Set]]
relations: [Model -> set[Relation]]
classes: [Program -> set[Class]]
atribs: [Program -> set[Atribute]]
```

For defining a conformance notion, sets and relations must be represented in the program somehow, in terms of classes and attributes, respectively. Based on the work of Rinard and Kuncak [Rinard and Kuncak 2001], we can define several classifications for syntactic mapping rules. For instance, the most straightforward mapping rules for sets and relations, respectively, are described as follows:

- **Class-Based Mapping.** A set is mapped to all objects of a given class (including its subclasses);
- **Attribute-Based Mapping.** A relation is mapped to all values for the corresponding attribute name – pairs of objects from the two corresponding classes.

Mapping 1 for the banking application in Section 2 uses class and attribute-based mappings; `Account`, `ChAcc`, `Customer` and `CreditCard` sets are implemented as classes, besides `accs` and `card` as attributes. The next PVS fragment formalizes the predicates for both mappings, for each set and relation. They are valid whether a class implements each set and an attribute implements each relation (for simplicity, we consider the same names). The function `name` yields the name of the argument element, analogous to `type`. The type an attribute consists in a pair of class types – its source and target types.

```
ClassBasedMapping(s:Set, p:Program): boolean =
  ∃ c:classes(p) | name(s)=name(c)
AtribBasedMapping(r:Relation, p:Program): boolean =
  ∃ a:atribs(p) | name(r)=name(a) ∧ type(r)=type(a)
```

As an alternative, relations in Mapping 2 for the banking model may follow a *collection-based* mapping rule:

- **Collection-Based Mapping.** A relation is mapped to the values referenced by a collection object (offered by standard programming language libraries, as in Java).

In our example, a pair customer-account from the relation `accs` in the model is given by relating this customer to the elements of its `List` objects in the heap. The following fragment defines this mapping for all relations from a model `m`, in which `targetType` denotes the function yielding its target type, which must be subtype of `Collection`.

```
CollectionBasedMapping(r:Relation,p:Program): boolean =
  ∃ a:atribs(p) | name(r)=name(a) ∧ targetType(a)=Collection
```

Also, as in Mapping 3, subtype relationships may be mapped to an alternative representation:

- **Attribute-Based Subtype Mapping.** A subset is mapped to objects of the superclass whose special attribute stores a specific value that indicates the subtype.

In our example, Mapping 3 indicates the subtype relationship as an attribute `type` into the `Account` class; when the value of `type` is `CHECKING`, it is considered a checking account. The following PVS fragment defines this mapping; `superType(s)` describes the parent set of `s` (considering only single inheritance), whereas `enum(sc)` denotes an enumeration type declared within class `sc`. Also, the subset's name must be a component of the enumeration type.

```
AttributeBasedSubtypeMapping(subset:Set,p:Program): boolean =
  ∃ sc:classes(p),t:attribs(p),en:enum(sc) |
         name(superType(subset))=name(sc) ∧ sourceType(t)=sc ∧
         targetType(t)=en ∧ name(subset) ∈ values(en)
```

4.2. Syntactic Conformance

With the definition of commonly used mappings, now we can choose mapping rules for sets, relations and subtypes in order to characterize syntactic conformance. These mappings may be mixed; for instance, a particular syntactic conformance may encompass mappings for subtypes implemented either as subclasses or attribute-based subtyping in the superclass. An instantiation of the framework is established when the mapping rules are provided, in the form of a predicate for a particular pair model-program.

The predicate must quantify over all sets and relations from the model, which characterizes a correct mapping – every model element must be representable in the program. This predicate is then used as a basis for a semantic conformance definition, as showed in Section 5. As an example of predicate for syntactic conformance, the next PVS fragment defines a syntactic conformance notion in which sets have class-based implementations and relations can be either collection-based or attributes-based.

```
SyntacticConformance1(m:Model,p:Program): boolean =
  ∀ s:sets(m) | ClassBasedMapping(s,p) ∧
  (∀ r:relations(m) | AtribBasedMapping(r,p) ∨
             CollectionBasedMapping(r,p))
```

Another example of syntactic conformance admits a program that implements subsets either as subclasses or attribute-based subtyping. The `topLevel` predicate indicates whether a set has no supertypes.

```
SyntacticConformance2(m:Model,p:Program): boolean =
  ∀ s:sets(m) | topLevel(s) ⇒ ClassBasedMapping(s,p) ∧
             (¬topLevel(s) ⇒ ClassBasedMapping(s,p) ∨
                   TypeBasedInheritanceMapping(s,p)) ∧
  ∀ r:relations(m) | AtribBasedMapping(r,p)
```

5. Semantic Conformance

For a given syntactic conformance, the framework offers a formal definition for semantic conformance; this conformance is established in terms of allowed model states and the reachable program states. Section 5.1 shows the semantic definitions useful for conformance. Then, in Section 5.2, we describe an approach for applying the syntactic mappings in semantic conformance, by means of a mapping formula. Finally, we establish semantic conformance, which is modeling and programming language independent.

5.1. Model and Program Semantics

We consider the semantics of an object model as the set of all valid *instances*. An instance contains mappings of set and relation names to a sets of *values*, as declared next. Values for set names are single objects, whereas relation names are assigned lists of objects, with the latter's size depending on the relation's arity. As more useful in practice, we only consider binary relations, so the length of the relation value is always two (2). A valid instance satisfies all modeled invariants.

```
Value: TYPE = list[Object]
Instance: TYPE =
  [# mapSet: [SetName->set[SetValue]],
     mapRel: [RelName->set[RelValue]] #]
semantics(m:Model): set[Instance]
```

The chosen representation for model instances is language independent, indicating how the semantics of the object model is defined; mapping from names to values can describe instances of object models written in languages such as Alloy or UML class diagrams. In fact, any modeling language whose semantics can be defined in terms of instances is applicable.

Regarding OO programs, states are formalized as *heaps* of object values, being defined in the following PVS fragment as a record mapping class names to sets of objects and attribute names to pairs of objects (indicating their relationship) – program values are equal to model values. If an object in a heap contains an attribute storing a *null* value, no attribute value exists for that object.

```
Heap: TYPE =
  [# mapClass: [ClassName -> set[ClassValue]],
     mapAtrib: [AtribName -> set[AtribValue]]  #]
```

The semantics of programs is given by the set of sequences of heaps resulting from all possible execution traces (depending on the possible program inputs), as showed next. For semantic conformance with an object model in our framework, the transition between heaps is not relevant. Our focus is on defining how each *relevant heap* follows the model invariant – we regard a relevant heap as a stable program state that is important for conformance checking. They are acquired in our definitions by means of a set of heaps taken from all reachable heaps, yielded by a `filter` function, defining the heaps for a set of program names (this is another hot spot of the framework, as detailed in Section 6).

```
semantics(p:Program): set[seq[Heap]]
heaps(p:Program): set[Heap]= filter(semantics(p),names(p))
```

5.2. Mapping Formula

In order to establish semantic conformance, a correspondence between model and program elements must be precisely defined; the model invariants must be interpreted for program values within heaps. In fact, when a syntactic conformance is confirmed, we can apply a *mapping formula* relating model and program elements. Since model and program values have a uniform mapping, they can be related by equivalences.

We explain this idea by using `SyntacticConformance1` from Section 4.2. Assuming that the banking model from Figure 1 is implemented with structures that follow the mentioned syntactic conformance, the following formula entails from the definition:

```
Customer^m=Customer^p ∧ Account^m=Account^p ∧
ChAcc^m=ChAcc^p ∧ CreditCard^m=CreditCard^p ∧
(accs^m=accs^p ∨ accs^m=accs^p.elems) ∧
(card^m=card^p ∨ card^m=card^p.elems)
```

Names from the model are decorated with m, analogously with programs. In this case, all sets are implemented by corresponding classes, while relations can be implemented either with attributes or collections. In this case, each relation name present two options for mapping, which is represented by a disjunction. Part of the formula for accs is given by $\mathtt{accs}^m$=$\mathtt{accs}^p$.elems, where elems denotes the attribute from the collection to its elements. The join (.) operator allows us to compose values from Customer to directly relate its Account objects held by the collection.

As a second example, the following formula entails from SyntacticConformance2 being valid. The formula is different concerning the ChAcc subset.

```
Customer^m=Customer^p ∧ Account^m=Account^p ∧
CreditCard^m=CreditCard^p ∧
(ChAcc^m=ChAcc^p ∨ ChAcc^m={a:Account^p | a.type=CHECKING}) ∧
accs^m=accs^p ∧ card^m=card^p
```

In this example, the option of attribute-based subtyping is defined as a disjunction. In this case, there may be no direct mapping for the ChAcc concept in the program, indicating that correspondence is *content based* – the ChAcc concept in the heap is represented by Account objects whose attribute has a particular value.

We apply a simple logic for defining these formulae, based on first-order logic with transitive closure. For instance, $\mathtt{Account}^m$=$\mathtt{Account}^p$ is an example of equality formula between two expressions relating set and class names. Additional formulae include subset, negation, conjunction and universal quantification, allowing developers to specify more complex relationships between model and program constructs. In fact, we can express these formulae as first-order logic. It allows for more flexible definitions than simple correspondence between names, which offers the capability in defining complex content-based relationships (no only for inheritance, for sets and relations as well). We show the language's formulae and expressions as follows:

```
formula ::= expr ∈ expr | expr ⊆ expr | expr = expr | ¬ formula |
    formula ∧ formula | (∀ var: sigName | formula)
expr ::= setName | relName | className | attribName | var |
         expr binop expr | unop expr
binop ::= ∪ | ∩ | - | . | ->
unop ::= ~ | ^
```

5.3. Semantic conformance

With the syntactic mapping rules and the mapping formula, we can now establish the conditions on which programs are in semantic conformance with an object model. A program is in conformance with a model if, and only if, for every selected heap from its execution there is a correspondent instance from the semantics of the model; this correspondence is given by the translation provided for model names from the mapping formula.

```
semanticConformance(m:Model,p:Program,f:Formula): boolean =
  ∀ h:heaps(p) | ∃ i:semantics(m) | satisfyFormula(i,h,f)
```

The `satisfyFormula` predicate tests whether the formula is true for a given pair heap-instance. If semantic conformance is confirmed, we say that the model invariants hold during executions of the program. The established relationship between heaps and instances in semantic conformance is depicted in Figure 2.

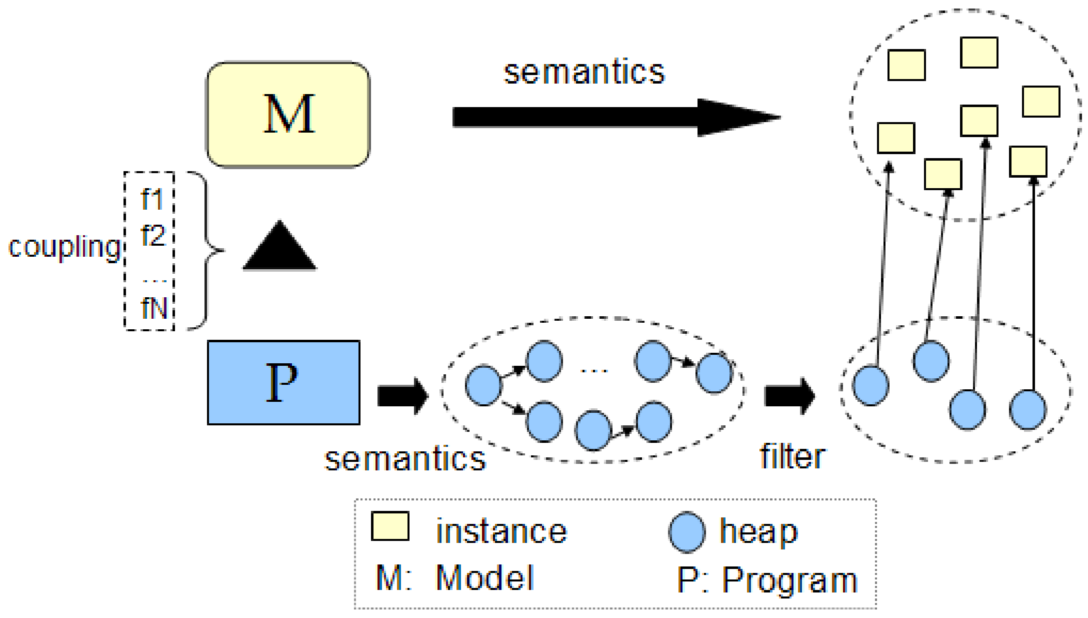

Figure 2. Semantic conformance.

6. Heaps of Interest

In our framework, the semantics of an OO program encompasses a set of heap sequences, each one resulting from possible execution traces of the program. For the purpose of verifying semantic conformance with object models, we consider the set of all heaps that appear in those sequences; in this case, no *filter* is applied. However, this approach does not always reflect the real intentions of conformance checking, since some of the heaps may be acceptably invalid at some well-defined points during program execution.

In order to illustrate the problem, consider the class `Customer` in Mapping 1 from Section 2, extended with a method for transferring its accounts to another customer. In the following method, the `for` command is used to navigate through the array of accounts, adding these accounts to the other customer (lines 3 and 4), before finally cleaning the `accs` array of the current customer (line 5).

```
  class Customer { ...
    void transferAccountTo(Customer c) {
3     for (int i=0;i<this.next;i++)
4        c.addAccount(this.accs[i]);
5     this.accs= null; }
  }
```

From the object model invariants in Figure 1, no two `Customer` objects have overlapping accounts. This is guaranteed before and after calls to the indicated method; however, this is not true for resulting heaps when executing the loops from the `for` command. For each new account reference copied to another customer, this account is owned by two customers, breaking the invariant.

Nevertheless, in practice, this program is suitably in semantic conformance, since it is natural that object methods perform encapsulated state changes which are not perceived by the users. For that, we need a less constrained filter that selects on which portions of the program code the clients may rely on model invariants, for example before entry and after execution of the `transferAccountTo` method. *Heaps of interest* are then the program heaps from those portions.

A suitable solution for making those program portions explicit is provided by Barnett et al. [Barnett et al. 2004], who present a specification methodology for enriching the program with constructs that indicate code on which invariants may be invalid. In their approach, every object is added a special public field, named `st` (for "state"), of type `{Invalid,Valid}`; if `obj.st` stores `Valid`, the object `obj` is considered valid, which means that the invariants over its state should hold. Otherwise, this is not guaranteed. As a result, conformance checking is performed only when all objects are valid.

In source code, the value of `st` can only be modified through the use of two new statements, `unpack` and `pack` [Barnett et al. 2004]. The command `unpack o` changes `obj.st` to `invalid`, opening a portion of code that is not considered for conformance – this portion is finalized with the `pack o` command. These commands are exemplified in a new version of the `transferAccountTo` method, in the following Java fragment.

```
class Customer { ...
  void transferAccountTo(Customer c) {
    unpack this;
    for (int i=0;i<this.next;i++)  c.addAccount(this.accs[i]);
    this.accs= null;
    pack this; } }
```

These two commands can be seen as object transaction delimiters. Object transactions include copy or removal of references, value changes and other operations for consolidating major state changes. The invariants are known to hold before or after those transactions. We now extend our formal definitions in the light of the presented solution, for defining a more powerful filter for heaps of interest. We first extend the definition of heaps including the `invalid` field, which indicates the truth for class names whose any object presents an invalid status, according to the `st` field.

```
Heap: TYPE =
  [# mapClass: [ClassName->set[ClassValue]],
     mapAtrib: [AtribName->set[AtribValue]],
     invalid: [ClassName->boolean] #]
```

The use of `unpack` and `pack` directives defines a more strict filter of heaps, which represents the second hot spot of the framework; the directives uncover the invalid heaps.

Now we can define a predicate that indicates invalid heaps for a set of class names, based on the `st` field. A heap is invalid for a set of class names if, and only if, it is invalid for any of the names in this set. The predicate is then used for selecting heaps of interest from the semantics of the program, as denoted by the following fragment by the `filter` PVS function. This function takes a set of sequences of heaps and a set of class names, resulting in the *set* of valid heaps from those sequences, ignoring equivalent heaps. `seq2set(s)(h)` converts the sequence `s` into a set, evaluating whether h is an element of this set.

```
invalidHeap(h:Heap,cNames:set[ClassName]): boolean =
   ∃ n:ClassName | n ∈ cNames ∧ invalid(h)(n)
filter(sequences:set[seq[Heap]],n:set[ClassName]): set[Heap] =
 {h:Heap | ∃ sq:sequences | h ∈ seq2set(sq) ∧ ¬invalidHeap(h,n)}
```

The `filter` function is then composed with `semantics`, resulting in the set of heaps of interest, as showed in Section 5. This set is then used in the `semanticConformance` predicate for a more flexible conformance checking. Figure 2 shows this filtering in action during conformance checking.

7. Applications

After showing all aspects of our framework, we describe one context in which it is instantiated. A conformance relationship is required when *refactoring* [Fowler 1999] – a transformation that improves software structure while preserving the observable behavior – is applied to an object model, and the conforming program is refactored accordingly. For this purpose, code regeneration is usually ineffective, due to the representation gap between model and program elements; existing implementation cannot be rewritten, since program statements usually refer to abstractions that have been changed by the model refactoring. This scenario requires manual updates in order to fix refactor the program.

In model-driven refactorings, besides semantics preservation, we are concerned with *conformance preservation*, which is usually not verified in traditional refactorings. Given that model and program are originally in conformance, in addition to be a valid program refactoring, always results in a program *in conformance* with the refactored model. In a previous work [Massoni et al. 2005], we propose an alternative, by refactoring models and programs simultaneously (model-driven refactoring), as show in Figure 3.

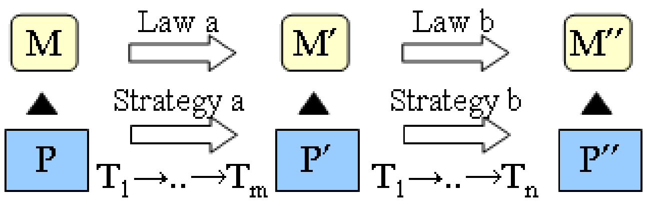
M
Law a
M′
Law b
M″
P
Strategy a
$T_1 \rightarrow .. \rightarrow T_m$
P′
Strategy b
$T_1 \rightarrow .. \rightarrow T_n$
P″

Figure 3. Model-driven Refactoring.

We consider object model refactoring as a composition of primitive semantics-preserving transformations (*laws* of modeling [Gheyi et al. 2004]). Each law applied to the model triggers the application of a *strategy* – a controlled sequence of program behavior-preserving transformations – to the source code. Strategies (1) update code abstractions as refactored in the model and (2) adapt implementation details according to the modified abstractions, with little or no user intervention.

The main idea behind strategies is the assumption that the original program is in conformance with the model, implying that all model invariants are guaranteed to be true in the program. In addition, developers intended to reflect model restructuring to source code might also expect some syntactic conformance, otherwise model-driven refactoring would not be applicable.

The conformance notion for model-driven refactoring defines the following syntactic mapping rules: class-based for all sets in the model (they all must be implemented by classes, besides additional implementation classes); relations with unconstrained multiplicities (0..*) are implemented following a collection-based mapping; otherwise, they are implemented using attribute-based mapping; also, subtyping is implemented with inheritance. The following PVS fragment depicts these variations, where `unconstrained` is the function that yields all relations with unconstrained multiplicity.

```
SyntConformModelDrivenRefact(m:Model, p:Program): boolean =
  ∀ s:sets(m) | ClassBasedMapping(s,p) ∧
  ∀ r:relations(m) |
    r ∈ unconstrained(m) ⇒ CollectionBasedMapping(r,p) ∧
    r ∉ unconstrained(m) ⇒ AttributeBasedMapping(r,p)
```

We used the described conformance notion to prove this property. The proof obligations for each strategy are showed in Figure 4.

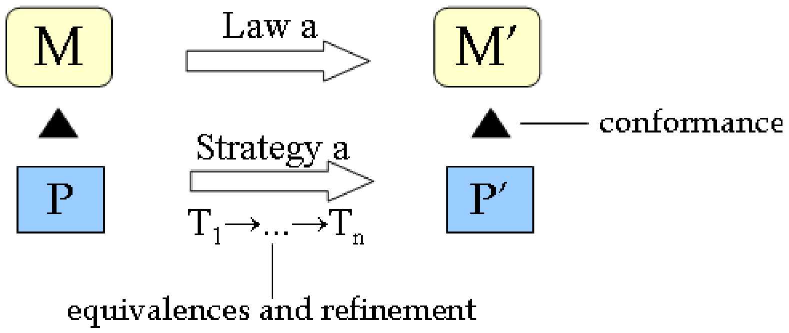

Figure 4. Proof Obligations for Strategies.

8. Related Work

Some ideas on syntactic conformance were based on Rinard and Kuncak's work [Rinard and Kuncak 2001]. They establish a connection between model and program by interpreting the predicate calculus from the object model in terms of heap values. This idea is used as a foundation for automatic analysis of the program in terms of invariants. They focus on the semantics of the modeling language, whereas our work is mostly concerned with the connection itself. The authors identify the usefulness of several kinds of mappings (class, attribute, collection and attribute-based subtyping), although no formal method is provided for the mapping. In turn, conformance is similarly addressed. It is confirmed for models and programs when all of the heaps that it may build conform to the object model under the given interpretation rules. However, they do not provide flexibility when defining filters for heaps of interest, besides not formalizing the types of conformance or providing a framework for more flexible syntactic mapping rules.

Our notion of semantic conformance resembles the classic notion of data refinement [Hoare 1972]. In data refinement, a retrieve relation defines the relationship between abstract values and their implementations, from the latter to the former; this idea is very similar to our mapping formula, establishing the relationship between object models and programs. The notion of data refinement is rather strong – besides stating a mapping between abstract and concrete states, it places constraints on operation over the states (state transitions). A correctness condition is that the effect of any program step over the concrete state can be simulated by an analogous operation in the abstract state. This notion is applied for (modeling) languages that allow state invariants and a collection of operations (with pre and postconditions); in contrast, object modeling may not consider operations, providing simpler and more abstract models which still can be precisely related to implementations.

Our framework can be useful for conformance checking tools, which perform automatic verification of semantic conformance with a given model. Techniques can be classified into at least two categories: static checking, which only applies to the implementation's source code, and dynamic analysis, which makes use of information available during the implementation's execution, not limited to artifacts available at compile time. A dynamic analysis approach for checking OO programs against object models is used in Embee [Crane and Dingel 2003], which captures the runtime state of a Java program at certain user-specified points. If the runtime states at those points conform with the object model, the program follows the simplest syntactic mapping for Alloy at least for that execution. Embee is limited to class- and attribute-based mappings; a notion of syntactic mapping is applicable for extending Embee's approach.

Regarding syntactic conformance, Harrison et al. [Harrison et al. 2000] show a method for maintaining consistency between object models (UML class diagrams) and Java programs, by advanced code generation from models at a higher level of abstraction, which allows more independence when making program changes not affecting models. A more specific case of syntactic conformance is addressed by another work [Guéhéneuc and Albin-Amiot 2004], aims at bridging the gap between object modeling and programming languages, in particular regarding binary associations, aggregations and compositions in UML class diagrams. They describe algorithms that automatically detect these relationships in code, introducing a more flexible conformance relationship for model relations. These conformance relationships can be represented by our mapping formula, allowing reasoning on semantic conformance, which is not present in those approaches.

9. Conclusions

In this paper, we described a formal framework for defining conformance notions between object models and OO programs. It supports independence of modeling and programming language semantics. Models can be implemented; the relevant program elements are related to model elements by a mapping formula based on first-order logic. The formula is generated by syntactic mapping rules (the first framework's hot spot). Additionally, our framework enables the user to select heaps of interest for establishing semantic conformance (the second hot spot). A more elaborated filter can be used to choose the stable states that must be considered when establishing conformance.

The framework is appropriate to accommodate freedom of implementation for abstract concepts, a useful task in design and implementation practice. Applications of this framework include conformance checking (semantic conformance), code generation and round-trip engineering (syntactic conformance), and model-driven refactoring (semantic and syntactic conformances). In the latter, conformance is critical to the soundness of transformations that affect model elements and its correspondent implementations.

Future work surely includes using and improving the framework for more useful syntactic mapping rules. We also intend to employ this framework in CASE tools and automatic conformance checking, evaluating the real power of mapping formulae in real case studies. Still, the main aim of the framework is to provide a formal basis for model-driven refactoring, establishing a precise correspondence between models and programs in order to automatically refactor programs based on restructuring changes in object models.

Acknowledgements

We'd like to thank the anonymous referees for useful suggestions, besides members of the Software Productivity Group.

References

[Barnett et al. 2004] Barnett, M. et al. (2004). Verification of Object-Oriented Programs with Invariants. *Journal of Object Technology*, 3(6):27–56.

[Booch et al. 1999] Booch, G. et al. (1999). *The Unified Modeling Language User Guide*. Object Technology. Addison Wesley.

[Crane and Dingel 2003] Crane, M. L. and Dingel, J. (2003). Runtime Conformance Checking of Objects Using Alloy. In *3rd Workshop on Runtime Verification*, pages 62–73.

[Fowler 1999] Fowler, M. (1999). *Refactoring—Improving the Design of Existing Code*. Addison Wesley.

[Gentleware 2005] Gentleware (2005). Poseidon for UML. http://www.gentleware.com/.

[Gheyi et al. 2004] Gheyi, R., Massoni, T., and Borba, P. (2004). Basic Laws of Object Modeling. In *3rd SAVCBS*, pages 18–25, USA.

[Guéhéneuc and Albin-Amiot 2004] Guéhéneuc, Y.-G. and Albin-Amiot, H. (2004). Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Proceedings of the 19th OOPSLA*, pages 301–314. ACM Press.

[Harrison et al. 2000] Harrison, W. et al. (2000). Mapping UML Designs to Java. In *Proceedings of OOPSLA 2000*, pages 178–187. ACM Press.

[Hoare 1972] Hoare, C. (1972). Proof of correctness of data representations. *Acta Informatica*, pages 271–281.

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

[Massoni et al. 2005] Massoni, T., Gheyi, R., and Borba, P. (2005). A Model-driven Approach to Formal Refactoring. In *Companion to the OOPSLA 2005*, pages 124–125, USA.

[Owre et al. 1992] Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A Prototype Verification System. In Kapur, D., editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, USA. Springer-Verlag.

[Rational Software 2006] Rational Software (2006). Rational Software Architect. http://-www-306.ibm.com/software/awdtools/architect/swarchitect/.

[Rinard and Kuncak 2001] Rinard, M. and Kuncak, V. (2001). Object Models, Heaps, and Interpretations. Technical Report 81. MIT Laboratory of Computer Science.

[Warmer et al. 2003] Warmer, J. et al. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition.