Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 195 (2008) 189–209

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

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

Tiago Massoni[1]

Department of Computing Systems
University of Pernambuco
Recife, Brazil

Rohit Gheyi[2] Paulo Borba[3]

Center of Informatics
Federal University of Pernambuco
Recife, Brazil

Abstract

Conformance between structural models and their implementations are usually simplified in practice, restraining reasoning to simple mappings between modeling and implementation constructs. This is not appropriate to accommodate the usual freedom of implementation for abstract concepts. A more flexible conformance notion must be addressed by conformance checking tools and model-driven development. In this paper, we propose a formal framework for defining conformance relationships between structural object models and object-oriented programs. In our framework, a syntactic mapping between model and program elements must be provided, yielding a coupling relation, used in framework instantiations for specific conformance relationships. Additionally, as in practice some intermediate program states are not relevant to conformance, we include the notion of heaps of interest, encompassing the filtered stable states for a less strict conformance checking. The framework is applied for establishing a conformance relationship in a technique of model-driven refactoring of programs.

Keywords: Object Model, Semantics, Conformance

1 Introduction

Software models provide visualization and specification for tackling software complexity. Mainly, models offer decomposition, which provides distinct perspectives

[1] Email: tlm@dsc.upe.br

[2] Email: rg@cin.ufpe.br

[3] Email: phmb@cin.ufpe.br

doi:10.1016/j.entcs.2007.08.033

for a particular aspect; for instance, whereas structural models depict main software elements and their inter-relationships, behavioral models specify how these elements interact to accomplish the system function. Structural models depicting domain concepts, relations and invariants in an object-oriented fashion are called *object models*. Languages for expressing object models include Alloy [13] or class diagrams from the Unified Modeling Language (UML) [2] annotated with invariants in the Object Constraint Language (OCL) [21].

A desirable property of object models is abstraction; ideally, they can be implemented by several structurally-different programs, with different behavior, as long as the invariants hold during their executions. These programs are then in *conformance* with the object model. However, in practice conformance between object models and their implementations are usually only considered when the modeled structures are directly declared in the program – for instance, a set named `Account` is implemented as an `Account` class – , restraining the freedom of implementation provided by abstraction (examples of conformance relationships are described in Section 2).

In this paper, we describe a formal framework for defining conformance relationships between object models and object-oriented programs, allowing reasoning with more flexible rules of conformance between model and program constructs. It supports independence of model and program semantics, by relying upon intermediate representations of *interpretations* and *heaps* (respectively, model and program states). These representations are related by a *coupling formula* involving model and program elements, defining how a model element is implemented in the program.

This syntactic mapping, as provided by the user, offers a definition for semantic conformance between models and programs, independently from object modeling or programming language. Our definitions were encoded in PVS [18] (showed in Section 3), whose interactive type checking helped us in finding specification errors. Reasoning on conformance may be useful for several applications, ranging from syntactic (code generation from models and reverse engineering) to semantic correspondence (conformance checking).

Specifically, we apply our framework for proving soundness for model-driven refactoring of object models and programs over one predefined conformance relationship. More specifically, we use the framework for proving that a sequence of behavior-preserving program transformations maintains a particular conformance relationship between refactored models and programs.

An additional contribution supports filtering of program states checked for conformance. 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. We incorporate a method for defining these *heaps of interest* into the framework, based upon previous work on verification of object-oriented invariants [1].

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

- A general definition for conformance relationships, which can be instantiated in terms of relationships between object model and program constructs (Section 4);
- Examples of conformance relationships presented as instantiations of the general definition (Section 5);
- Application of the framework for proving conformance preservation in model-driven refactorings (Section 7);
- A formal definition for heaps of interest, in terms of a filter over the program semantics, in Section 6.

2 Motivating Examples

Software models provide visualization and specification for tackling software complexity. Mainly, models offer decomposition, which provides distinct perspectives for a particular aspect; for instance, whereas structural models depict main software elements and their inter-relationships, behavioral models specify how these elements interact to accomplish the system function. In terms of object models, languages include Alloy or class diagrams from the UML annotated with OCL. In a number of software development contexts, it is useful that abstractions in models and source code evolve consistently, for documentation or even development purposes. For the applicability of this practice, a *conformance relationship* between models and programs must be defined.

Conformance can be given in syntactic and semantic terms. In an object model, sets of objects and relations between those sets constitute the main elements, along with logic invariants over these elements. Sets and relations are abstract, which will be given concrete representations in the program, in terms of object-oriented programming language constructs (classes, attributes, inheritance, etc.); we call this correspondence *syntactic conformance*. Besides syntactic conformance, model invariants must be preserved throughout the program's execution. If this is confirmed, the program is in *semantic conformance* with the model; this concept is similar to refinement. These relationships can be applied in several contexts. For instance, generation of source code from models and reverse engineering; evolution activities in model-driven methodologies [9], in which software is completely generated and evolved by manipulating models.

Usually, a *traditional conformance relationship* consists in modeled sets or relations being mapped to a single class or attribute, respectively. However, several possible conformance relationships may be useful in practice, as showed in the following example. Figure 1 shows a partial object model for concepts of a banking domain, using UML class diagrams. Sets are boxes, and relations are showed as arrows. 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 (.), in this case, denotes the standard relational composition, while `#` denotes set cardinality.

In following Java [8] fragments, we show several implementations of the same

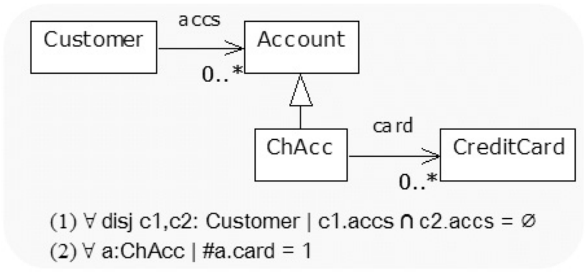
Customer
accs
0..*
Account
ChAcc
card
0..*
CreditCard
(1) ∀ disj c1,c2: Customer | c1.accs ∩ c2.accs = ∅
(2) ∀ a:ChAcc | #a.card = 1

Fig. 1. Object model for a banking example.

model using different conformance relationships. First, we define a program on the traditional conformance relationship.

Conformance relationship 1. Using the traditional conformance, sets are directly implemented as single classes. The attributes implement the modeled relations – `accs` as an array of accounts, card a single variable. `ChAcc` is implemented as a subclass of `Account`. The program declares several methods, according to software requirements.

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

Concerning semantic conformance, the following main program presents simple behavior that exercises the above declarations, creating two customers with different types of accounts. The program is in semantic conformance if, and only if, it fulfills all model invariants throughout its execution; in this case, the Java fragment maintains semantic conformance.

```
  static void main (String [] args) {
    Account a1 = new Account();
2   ChAcc a2 = new ChAcc();
    Customer c1 = new Customer();
    Customer c2 = new Customer();

    c1.addAccount(a1);
    c2.addAccount(a2);
8   a2.changeCard(new CreditCard()); ...
  }
```

The traditional conformance relationship 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. Similarly, semantic conformance with direct correspondence is usually easier to check. An informal analysis of the main program shows that the invariants hold during its execution; in every heap, no two customers have the same account and every checking account carries exactly one card. The implicit constraint of checking accounts being also accounts entails from the semantics of subclasses in Java.

Although useful for implementations, the traditional notion is too restrictive. Not so direct implementations are usually seen in practice, as shown below.

Conformance relationship 2. In another conformance relationship – in this case for relations –, the attributes are implemented as object *collections*, exemplified by the `List` of accounts and credit cards, as showed in the next Java fragment. The methods `add` and `set` include new elements in the list and replaces an item at a given position, respectively. Notice that despite choosing a collection for implementing the relation (in the prospect of extending the number of credit cards in the future), the `card` attribute must always contain a single card value, according to the invariant.

```
class Customer {
  List accs = new List(); ...
  void addAccount(Account a) { accs.add(a); } ...
}
class ChAcc extends Account {
  Card card = new List();
  void changeCard(CreditCard c){ this.card.set(0,c); } ...
}
```

Another example, specific for set conformance, is related to the subset relationship from the model, which may not be implemented with inheritance in the program, as showed in other conformance relationships.

Conformance relationship 3. Due to abstraction, another conformance relationship allows implementations to be even more distinct from the object model. Here the subset relationship from the object model is implemented as an attribute in `Account`, defining the type of each account.

```
class Account {
  int type;
  public Account(String type) { this.type = type; } ...

  //used only when type=1 (checking account)
  Card card = new List();
  void changeCard(CreditCard c) {
    if (this.type == 1) this.card.set(0,c);
    else //error!
  } ...
}
```

Conformance relationships 2 and 3 limit the application of CASE tools, since they require more intricate conformance relationship 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 [10,11]. Alternatively, a tool could allow users to define custom mappings for syntactic conformance. Well-known tools, such as Rational Software Architect [19] and Poseidon [6] 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 verify that model invariants hold in program heaps, when the program presents such disparate constructs for implement the modeled concepts. The user-defined mapping between model and program structures must be applied in this verification, in order to correctly relate values from both worlds and check the conformance. As shown in this section, several conformance relationships 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 [18]. 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 that we want to model 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 component 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. For instance, `balance(acc)(100)` is a predicate stating that the balance of the `acc` account is 100. The `WITH` keyword denotes the override operator, which replaces the mapping for `acc` by a new tuple, if `acc` is originally in the function domain. In the `withdraw` function, the expression containing the `WITH` operator denotes an account with the same owner of `acc`, but with a balance subtracted of `amount`. Similarly, we can declare a function representing the credit operation.

Besides declaring types and functions, a PVS specification can also declare axioms, lemmas and theorems. For instance, next we declare a theorem stating that the balance of an account is not changed when performing the withdraw operation after the credit operation with the same amount.

```
    withdrawCreditTheorem: THEOREM
      FORALL(acc: Account, amount: int) :
        balance(withdraw(credit(acc,amount),amount)) = balance(acc)
  END BankingSystem
```

The `FORALL` keyword denotes the universal quantifier. The previous quantification is over an account and an amount to be deposited and then withdraw.

4 Conformance Framework

In this section we introduce the formal framework definitions for establishing conformance relationships in PVS. A notion of conformance is given in terms of syntactic correspondence between constructs from models and programs (which we call coupling). This correspondence is a *hot spot* of the framework, as instantiated for particular conformance relationships. With a given coupling, a notion of semantic conformance can be verified.

Section 4.1 provide the building blocks for conformance – model and program states (*interpretations* and *heaps*, respectively). In section 4.2, we show how the syntactic coupling between constructs from models and programs is established, with the idea of translation functions. Finally, we formalize conformance – which is independent of modeling and programming languages –, showing its reliance on the coupling chosen by the user.

4.1 Basic Definitions

A formal definition of models and programs is given, using uninterpreted types, in the following PVS fragment[4]. 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. We treat relations and attributes as a name, not qualified with the name of the declaring set or class (considering models and programs that declare relations and attributes, respectively, using unique names).

```
  Model: TYPE
  Program: TYPE

  sets: [Model -> set[Set]]
  relations: [Model -> set[Relation]]
  classes: [Program -> set[Class]]
  attribs: [Program -> set[Attribute]]
```

We consider the semantics of an object model as the set of all valid *interpretations*. An interpretation contains mappings of set and relation names to sets of *values*, as declared next. Values may be single objects for sets and pairs of objects

[4] Besides mixing some well-known mathematical symbols with PVS keywords and functions, we consider a few extensions to the original PVS language for improving readability.

for relations; we consider only binary relations. A valid interpretation satisfies all modeled invariants.

```
objValue: TYPE
objPairValue: TYPE

Interpretation: TYPE =
  [# mapSet: [SetName->set[objValue]],
     mapRel: [RelName->set[objPairValue]]
   #]
semantics(m:Model): set[Interpretation]
```

The chosen representation for model interpretations is language independent, indicating how the semantics of the object model is defined; mapping from names to values can describe interpretations of object models written in languages such as Alloy [13] or UML class diagrams [2]. In fact, any modeling language whose semantics can be defined in terms of interpretations is applicable.

Regarding object-oriented programs, states are formalized as *heaps* of object values, 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 considered equivalent to model values. If an object in a heap contains an attribute storing a *null* value, no pair of values exists with that object as the first member.

```
Heap: TYPE =
  [# mapClass: [ClassName -> set[objValue]],
     mapAttrib: [AtribName -> set[objPairValue]]
   #]
```

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 in the next PVS fragment. For conformance with a structural 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))
```

For illustrating our heap definition, Figure 2 depicts examples of partial heaps after executing the indicated lines in the main program from Section 2. The values are represented by the reference variables pointing to the objects (except for `CreditCard`, whose objects are anonymous – we used `cc1` and `cc2`).

Similarly to interpretations, we adopted an intermediate representation for heaps. Object-oriented programming languages may present diverse structures for heaps; for instance, heaps in Java programs are essentially graphs, where nodes are objects and edges represent attributes (reference-based). Nevertheless, Java heaps can be easily translated into this intermediate representation; as a result, the definitions are language independent, considering the semantics as a sequence of heap traces.

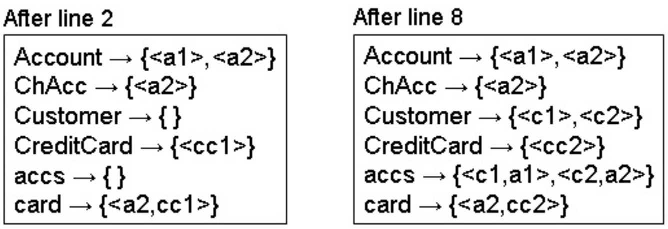

Fig. 2. Heaps from the banking example.

4.2 Syntactic Coupling

In order to establish conformance, a correspondence between model and program declarations is established. For instance, the traditional conformance relationship for the bank-related concepts is classes representing sets and attributes for relations. If we define these rules in terms of formulae, exemplars could be: $\texttt{Account}^{\texttt{m}}=\texttt{Account}^{\texttt{p}}$ and $\texttt{accs}^{\texttt{m}}=\texttt{accs}^{\texttt{p}}$, where `m` and `p` indicate names from the model or program, respectively. Likewise, more complex formulae can be provided, adding required flexibility for different categories of conformance. In our framework, the set of these definition formulae – for given model and program – is defined as the *coupling* relation, as shown in the next fragment; coupling is another framework hot spot, given for specific conformances. Section 5 describes particular definitions of coupling for the conformance relationships from the banking example.

```
coupling(m:Model, p:Program): Formula
```

The definition of a specific conformance relationship in this framework involves the establishment of a formula for the coupling. This formula can be specified in language based on first-order logic with transitive closure, which we also defined in PVS. For instance, $\texttt{Account}^{\texttt{m}}=\texttt{Account}^{\texttt{p}}$ 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. The other kinds of formulae, such as existential quantification and disjunction of formulae, can be appropriately derived from the core constructs. Moreover, we consider some binary (union, intersection, difference, join and product) and unary (transpose and transitive closure) expressions. We show the language's formulae and expressions as follows (the PVS definition, using abstract datatypes [18], is omitted here for clarity).

```
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 ::= ~ | ^
```

In our core language for coupling formulae, we consider twelve kinds of expressions, which are specified next. We have expressions for set, class, relations, attributes and variable names. Moreover, there are five kinds of binary expressions representing the union, intersection, difference, join and product expressions. Finally, we have the transpose and closure unary expressions. In order to formalize them, we create a PVS abstract datatype [18].

```
Expression: DATATYPE BEGIN
  IMPORTING Names
    VARIABLE(n: VarName): VARIABLE?: Expression
    SETNAME(n: SetName): SIGNAME?: Expression
    RELNAME(n: RelName): RELNAME?: Expression
    CLASSNAME(n: ClassName): SIGNAME?: Expression
    ATRIBNAME(n: AtribName): RELNAME?: Expression
    UNION(l,r: Expression): UNION?: Expression
    INTERSECTION(l,r: Expression): INTERSECTION?: Expression
    DIFFERENCE(l,r: Expression): DIFFERENCE?: Expression
    JOIN(l,r: Expression): JOIN?: Expression
    PRODUCT(l,r: Expression): PRODUCT?: Expression
    TRANSPOSE(exp: Expression): TRANSPOSE?: Expression
    CLOSURE(exp: Expression): CLOSURE?: Expression
END Expression
```

A PVS datatype is specified by providing a set of *constructors*, *recognizers* and *accessors*. The previous datatype has some *constructors*, such as `SETNAME` and `UNION`, which allow the expressions to be constructed. For instance, the expression `SETNAME(n)` is an element of this datatype if `n` is a set name. The `UNION?` and `CLOSURE?` *recognizers* are predicates over the `Expression` datatype that are true when their argument is constructed using the corresponding constructor. For instance, `CLOSURE?(e)` is true when `e` is a closure expression. Suppose that we have the `UNION(e1,e2)` union expression, where `e1` and `e2` are expressions. We can use the `l` and `r` *accessors* to access the left and right expressions. For example, the `l(UNION(e1,e2))` expression yields the `e1` expression. When a datatype is type checked, a new theory is created that provides the axioms and induction principles needed to ensure that the datatype is the initial algebra defined by the constructors [18]. In our core language, we have seven kinds of formulae. Besides formulae representing true and false, there are negations, conjunctions, universal quantifications, subset and equality formulae. The set membership formula can be expressed in terms of the subset formula. Similar to expressions, we create a PVS datatype for formulae.

```
Formula: DATATYPE BEGIN
  IMPORTING Expression, Names
    TRUE: TRUE?: Formula
    FALSE: FALSE?: Formula
    NOT(f: Formula): NOT?: Formula
    AND(l,r: Formula): AND?: Formula
    FORALL(x:VarName, t:SigName, f:Formula): FORALL?: Formula
    SUBSET(l,r: Expression): SUBSET?: Formula
    EQUAL(l,r: Expression): EQUAL?: Formula
END Formula
```

For example, suppose in the ∀ `b:Bank | some b.accounts` formula stating that all banks have at least one account. The `some` keyword, when applied to an expression, defines a predicate stating that there is at least one element in the expression. Considering our formalization for representing universal quantification formulae, the variable name `x` of ∀ `b:Bank | some b.acc` is `b`, its type `t` is `Bank` and the `f` formula is `some b.accs`. The `satisfyFormula` predicate is a recursive PVS function. For example, suppose an universal quantification formula ∀ `x:exp | f`, the `satisfyFormula` predicate checks whether the `f` formula is valid in the original interpretation extended with a value given to the variable name `x`, as formalized next.

```
∀ v: map(i)(T) |
  satisfyFormula(f, i WITH [map := map(i) WITH [x |-> {v}]])
```

The evaluations of other formulae are very similar and have the standard semantics [14]. For example, an interpretation satisfies a conjunction formula when it satisfies each subformula. Moreover, an interpretation satisfies an equality formula (`exp1 = exp2`) when both subexpressions have the same values in the interpretation, as declared next.

```
evalExpression(exp1,i) = evalExpression(exp2,i)
```

The full specification of `satisfyFormula` is declared next.

```
satisfyFormula(f:Formula,i:Interpretation,h:Heap): RECURSIVE boolean=
  CASES f OF
    TRUE_: TRUE,
    FALSE_: FALSE,
    NOT_(f1): NOT satisfyFormula(f1,i,h),
    AND_(f1, f2):
      satisfyFormula(f1,i,h) AND satisfyFormula(f2,i,h),
    FORALL_(x, t, f1):
      FORALL(v:SetValue): mapSet(i)(t)(v) ⇒
        satisfyFormula(f1,i WITH [mapVar := mapVar(i) WITH
                                   [x |-> v1:Value | v=v1]],h),
    EQUAL(e1, e2):
      evalExpression(e1,i,h) = evalExpression(e2,i,h),
    SUBSET(e1, e2):
      FORALL(v:Value): evalExpression(e1,i,h)(v) ⇒
                         evalExpression(e2,i,h)(v)
  ENDCASES
  MEASURE complexity(f)
```

The `evalExpression` relation is a recursive PVS function, which evaluates an expression for the given interpretation and heap values. Next, we specify the evaluation of a union expression (`exp1∪exp2`).

```
evalExpression(e1,i,h) ∪ evalExpression(e2,i,h)
```

The evaluation of other expressions is specified similarly. For instance, the evaluation of a product expression is the product of each subexpression's evaluation. Next we specify the complete specification of `evalExpression`.

```
evalExpression(e:Expression,
               i:Interpretation, h:Heap): RECURSIVE set[Value] =
  CASES e OF
    VARNAME(n): mapVar(i)(n),
    SIGNAME(n): mapSet(i)(n),
    RELNAME(n): mapRel(i)(n),
    CLASSNAME(n): mapClass(h)(n),
    ATRIBNAME(n): mapAtrib(h)(n),
    UNION(e1, e2):
      union(evalExpression(e1,i,h), evalExpression(e2,i,h)),
    INTERSECTION(e1, e2):
      intersection(evalExpression(e1,i,h), evalExpression(e2,i,h)),
    DIFFERENCE(e1, e2):
      difference(evalExpression(e1,i,h), evalExpression(e2,i,h)),
    JOIN(e1,e2):
      {v:Value | EXISTS (v1,v2:Value) :
                   evalExpression(e1,i,h)(v1) ∧
                   evalExpression(e2,i,h)(v2)  ∧
                   canJoin(v1,v2) ∧ v = join(v1,v2)
      },
    PRODUCT(e1,e2):
      {v:Value | EXISTS (v1,v2:Value) :
                   evalExpression(e1,i,h)(v1) ∧
                   evalExpression(e2,i,h)(v2) ∧
                   v = product(v1,v2)
      },
    TRANSPOSE(e1):
      {v:Value | EXISTS (v1:Value) :
                   evalExpression(e1,i,h)(v1) ∧
                   v = transpose(v1)
      },
    CLOSURE_(e1):
      {v:Value | evalClosure(v,evalExpression(e1,i,h))}
  ENDCASES
  MEASURE complexity(e)
```

4.3 Semantic conformance

With the building blocks and the coupling 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 filtered heap from its execution there is a correspondent interpretation from the semantics of the model; this correspondence is given by the coupling formula. The models and programs are considered well-formed – this requirement may be specified in PVS.

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

The `map` predicate states, for each formula in `coupling`, whether a pair heap-interpretation satisfies the formulae from `coupling` (represented by the `satisfyFormula` predicate). In this case, the heap satisfies the invariants from the model – there is a corresponding interpretation.

```
map(m:Model,p:Program,i:Interpretation,h:Heap): boolean =
  satisfyFormula(coupling(m,p),i,h)
```

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 3. This function relies on the `map` predicate, which relates model and program names according to the formulas given in the `coupling` definition.

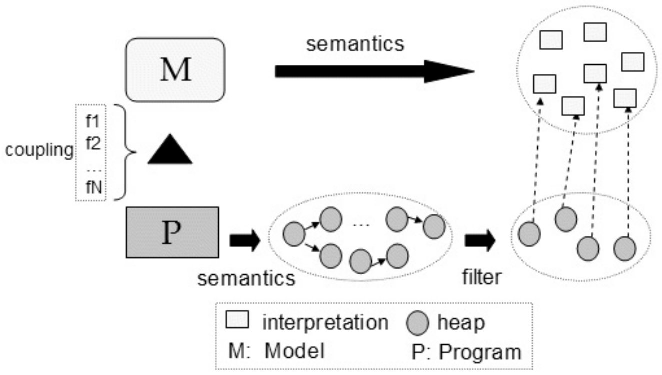

Fig. 3. Semantic conformance.

5 Instantiation

The previous section describes the conformance framework; in this section, we instantiate the indicated hot spot for the conformance relationships showed in Section 2. We also describe a technique for defining coupling formulae in terms of types of syntactic couplings seen in practice.

When establishing conformance, a coupling relation is provided, linking each set in the object model to a corresponding set of objects in the heap (same for relations). Rinard and Kuncak [20] provide a classification for usually applied couplings. The traditional syntactic conformance for sets and relations, respectively, are described as follows:

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

Conformance relationship 1 for the banking application in Section 2 uses class and attribute-based couplings; `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 kinds of couplings, 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).

```
ClassBasedCoupling(s:Set, p:Program): boolean =
  ∃ c:classes(p) | name(s)=name(c)
AtribBasedCoupling(r:Relation, p:Program): boolean =
  ∃ a:atribs(p) | name(r)=name(a) ∃ types(r) = types(a)
```

As the classification describes a way to map syntactic constructs, it implies in an concrete definition for the `coupling` formula. In fact, this set can be automatically generated from a given conformance type. The following PVS theorem formalizes this property for class- and attribute-based couplings. The theorem states that for every pair model-program in syntactic conformance following the traditional couplings, a coupling set of formulae is automatically defined, in terms of equalities for set-class and relation-attribute pairs of names.

```
couplingFromClassAttributeBased: THEOREM
  ∀ m:Model,p:Program |
    ∀ s:sets(m),r:relations(m) |
      ClassBasedCoupling(s,p) ∧ AtribBasedCoupling(r,p) ⇒
    coupling(m,p) =
      {f:Formula_ |
      ∃ s:sets(m),c:classes(p) |
      name(s)=name(c) ∧
      f = EQUAL(SIGNAME(name(s)),CLASSNAME(name(c)))
      ∨
      ∃ r:relations(m),a:attribs(p)) |
      name(r)=name(a) ∧
      f = EQUAL(RELNAME(name(r)),ATRIBNAME(name(a)))}
```

For each set and class with the same name (for instance `Account`), an equality formula (`EQUAL`) between those names is implied; the same is observed for a relation and an attribute name.

As an alternative for relation coupling, Conformance relationship 2 for the banking model follows a *collection-based* coupling:

- **Collection-Based Coupling.** 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 coupling this customer to the elements of its `List` objects in the heap.

The following fragment defines this coupling for all relations from a model `m`, in which `targetType` denotes the function yielding its target type, which must be subtype of `Collection`. In the coupling relation, the formula for the `accs` is given by $\texttt{accs}^{\texttt{m}}=\texttt{accs}^{\texttt{p}}\texttt{.elems}$, where `elems` denotes the attribute from the collection to its elements.

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

Our conformance framework can be general enough to allow defining different kinds of couplings for elements in the same model. For instance, some relations may be defined as class-based, while others use a collection-based coupling. In this case, only part of the coupling relation is generated by following these kinds of couplings; other couplings can be used for particular names in the same model.

Although coupling types can be useful for developers in indicating how concepts are implemented in source code, more complex definitions can be used to indicate this correspondence. In our example, Implementation 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. Therefore, there is no direct coupling 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.

The language for coupling formulae presented in Section 4.2 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). For instance, the formulae for `Account` and `ChAcc` in the last implementation could be represented as follows, where value `checking` for type represents checking accounts:

$$\texttt{Account}^{\texttt{m}} = \texttt{Account}^{\texttt{p}}$$
$$\texttt{ChAcc}^{\texttt{m}} = \{\texttt{a:Account}^{\texttt{p}} \mid \texttt{a.type=checking}\}$$

Although some set comprehension constructs do not appear in our definition for the language presented in Section 4.2, we could easily derive it as *shorthands* for the core constructs, for making this logic practical. For instance, the existential quantifier can be built from the universal quantifier.

6 Heaps of Interest

In our framework, the semantics of an object-oriented 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 heaps from those sequences. At first, we considered a filter yielding all possible heaps from execution traces; however, this approach does not truly reflect the real intentions of conformance checking sometimes, since some of the heaps may be acceptably invalid at some well-defined points of the program.

In order to illustrate the problem, consider the class `Customer` in the first implementation for the object model in 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 (line 4), before finally cleaning the `accs` array of the current customer (line 5).

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

From the object model invariant in the example in Figure 1, no two `Customer` objects may 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 to restrict on which portions of the program code the clients may rely on model invariants, for example before entry and after execution of the `transferAccountToCustomer` 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. [1], which 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=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` [1]. The command `unpack obj` changes `obj.st` to `invalid`, opening a portion of code that is not considered for conformance – this portion is finalized with the `pack obj` 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]
   #]
```

Now we can define a predicate that indicates invalid heaps for a set of class names, based on the new 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 PVS fragment by the `filter` 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:cNames | n ∈ invalid(h)

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 in Section 4. This set is then used in the `semanticConformance` predicate for a more flexible conformance checking. Figure 3 shows this filtering in action during conformance checking.

7 Application of Conformance in Model-driven Refactoring

After showing all aspects of our framework, we describe some contexts in which its instantiations can be applied. A conformance relationship is required when *refactoring* [5] – a transformation that improves software structure while preserving the observable behavior – is applied to an object model, and the conforming program is refactored accordingly. Section 7.1 shows an overview of this approach, while Section 7.2 details the conformance relationship used in the approach in the context of our conformance framework.

7.1 The Approach

In a number of software development projects, it is useful that abstractions in models and source code evolve consistently, for documentation or even development purposes, as seen in model-driven methodologies [9]. In this context, 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 addition, no evidence is provided on whether conformance is maintained after regenerating program elements. In a previous work [17], we propose an alternative, by refactoring models and programs simultaneously (model-driven refactoring), as show in Figure 4.

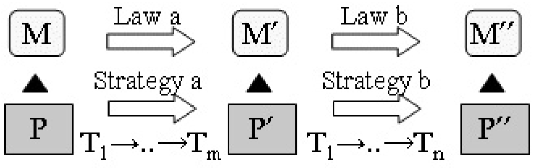
M
Law a
M′
Law b
M″
Strategy a
Strategy b
P
$T_1 \rightarrow .. \rightarrow T_m$
P′
$T_1 \rightarrow .. \rightarrow T_n$
P″

Fig. 4. Model-driven Refactoring.

We consider object model refactoring as a composition of primitive semantics-preserving transformations (*laws* of modeling [7]). 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.

In particular, we consider object models in Alloy and programs in a Java-like formal language, ROOL (Refinement Object-Oriented Language) [3]. Both languages present relatively complete sets of primitive semantics-preserving transformations and refinement calculus, used as a basis for model-driven refactoring. Refactoring strategies can only be correctly applied to source code with a condition: original object model and program must be in conformance. Therefore, a specific conformance relationship which is appropriate for the consistent refactoring must be defined; we then use our formal framework for defining this conformance.

7.2 Syntactic and Semantic Conformance

Developers intended to reflect model refactoring to source code might also expect some syntactic conformance, otherwise model-driven refactoring would not be applicable. In model-driven refactoring, we used a specific conformance relationship, as shown in the next PVS fragment. The syntactic mapping for sets defines one direct class for each set (for simplicity, with the same name). Also, all supersignatures for the set (given by `super(s)`) are included in the set of superclasses of the corresponding class (`super(c)`), indicating that more superclasses can be declared in the program, but the hierarchy is maintained.

```
setMapping(s:Set,p:Program): boolean =
  ∃ c:classes(p) | name(s)=name(c) ∧ super(s)⊆ super(c)
```

Likewise, every relation is mapped to one attribute, with one additional constraint: relations with single multiplicity (yielding a scalar value) are mapped to single attributes, while relations with set multiplicity must be mapped to collection-type attributes.

```
relationMapping(r:Relation, p:Program): boolean =
  ∃ a:attribs(p) | name(r)=name(a) ∧ types(r) = types(a) ∧
   isScalar(r) ⇒ ¬(targetType(a)=Collection) ∧
   ¬isScalar(r) ⇒ targetType(a)=Collection
```

The following PVS fragment depicts these variations, where `unconstrained` is the function that yields all relations with unconstrained multiplicity.

```
ModelDrivenRefactCoupling(m:Model, p:Program): boolean =
  ∀ s:sets(m) | setMapping(s,p) ∧
  ∀ r:relations(m) | relationMapping(r,p)
```

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, a program strategy associated with the applied law of modeling, in addition to be a valid program refactoring, always results in a program *in conformance* with the refactored model. This is a relevant property, since users do not worry about conformance preservation when applying laws of modeling, refactoring the program automatically. We used the described conformance instantiation to prove this property. The proof obligations for each strategy are showed in Figure 5.

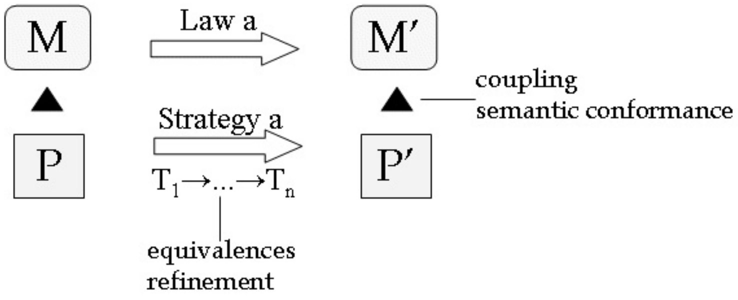

Fig. 5. Proof Obligations for Strategies.

8 Related Work

Some conformance relationship types were based on Rinard and Kuncak's work [20]. 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 (similar to the language we presented in Section 4.2), whereas our work is mostly concerned with the connection itself. The authors identify the usefulness of several kinds of mappings (class, attribute, collection and content-based), 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 a notion of heaps of interest, besides not formalizing the types of conformance or providing a framework for more flexible conformance relationships.

Another approach [15] relates object models and programs by defining models – called *heap invariants* – that describe sets of legal program states. The object models considered in that approach are more concrete than the ones we used, extended with additional constructs (qualified sets and indexed relations). In this context, interpretation of model elements in terms of program elements follow direct mappings only. Our work may be used to extend these interpretations with a flexible coupling formula.

Our notion of conformance is very similar to the classic notion of data refinement [12,16]. 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 coupling, 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. Whereas data refinement is a better option for programs that are correct by systematic construction from a specification, our formalization is better fit for alternative methods, that rely on conformance checking of implementations.

Our framework can be useful for conformance checking tools. Conformance checking tools consist in automatically verifying whether an implementation is in 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 object-oriented programs against object models is used in Embee [4], which captures the runtime state of a Java program at certain user-specified points. If the runtime states at that points conform with the object model, the program follows a structural correspondence with Alloy at least for that execution. Embee is limited to class- and attribute-based mappings; a notion of coupling is applicable for extending Embee's approach.

Regarding syntactic conformance, Harrison et al. [11] 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 [10], 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 coupling 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 relationships between object models and object-oriented programs. It supports independence of modeling and programming language semantics, besides the definition of a family of conformance relationships. Complete or partial models can be implemented; the relevant program elements are related to model elements by a *coupling* relation based on first-order logic. We exemplified framework instantiations with examples of conformance. Additionally, we assume a more realistic notion of program semantics by filtering *heaps of interest*, which define the stable states of a program 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, code generation and round-trip engineering, and model-driven development, more specifically refactoring. In the latter, a formal conformance relationship is critical to the soundness of transformations that affect model elements and its correspondent implementations.

A scenario where reasoning on conformance is required is characterized by analysis and proofs involving semantic conformance. The framework definitions, in conjunction with specific framework instantiations, offer a foundation for establishing conformance when needed and checking whether it holds in several contexts. For instance, a dynamic analysis approach for checking object-oriented programs against object models is used in the Embee tool [4]. It works by capturing the runtime state of a Java program at certain user-specified points. The tool is limited to class and attribute-based mappings; more flexible conformance relationships formalized in this paper can surely be applied to extend the applicability of such tool.

Our framework can also be applied to improve *traceability* between analysis models and implementation, especially in round-trip engineering tools. A formal definition of this relationship is useful for those tools, which might be able to generate source code based on coupling formulae – the same can be used for reverse engineering. Users would be able to choose the conformance relationship for each model element in isolation. As an example, the content-based mapping for inheritance from Section 2 could be generated by using its coupling formula (over the value of the `type` attribute) as a basis. Today, most tools offer limited options for correspondence – usually class and attribute-based implementation only, restraining abstraction in modeling.

Future work surely includes using and improving the framework for more useful conformance relationships. Issues such as adequacy of the coupling relation will be explored as well. We also intend to employ this framework in CASE tools and automatic conformance checking, evaluating the real power of coupling 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.

References

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

[2] Grady Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison Wesley, 1999.

[3] Paulo Borba and Augusto Sampaio, Basic Laws of ROOL: an object-oriented language, Revista Brasileira de Informtica Terica e Aplicada, **7** (2000), 49-68,

[4] Michelle L. Crane and Juergen Dingel. Runtime Conformance Checking of Objects Using Alloy. In *3rd Workshop on Runtime Verification*, pages 62–73, 2003.

[5] Martin Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] Gentleware. Poseidon for UML, 2005. http://www.gentleware.com/.

[7] R. Gheyi, T. Massoni, and P. Borba. Basic laws of object modeling. In *3rd SAVCBS*, pages 18–25, Newport Beach, United States, October 2004.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, third edition, June 2005.

[9] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[10] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering Binary Class Relationships: Putting Icing on the UML Cake. In *Proceedings of the 19th OOPSLA*, pages 301–314. ACM Press, October 2004.

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

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

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

[14] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[15] Daniel Jackson. Object Models as Heap Invariants. *Essays on Programming Methodology*, pages 247–268, 2003.

[16] H. Jifeng, C. Hoare, and J. Sanders. Data refinement refined. In *ESOP '86: Proceedings of the European Symposium on Programming*, pages 187–196, London, UK, 1986. Springer-Verlag.

[17] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A model-driven approach to formal refactoring. In *Companion to the OOPSLA 2005*, pages 124–125, USA, October 2005.

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

[19] Rational Software. Rational Software Architect, 2006, `http://www-306.ibm.com/software/awdtools/architect/swarchitect/`

[20] Martin Rinard and Viktor Kuncak. Object Models, Heaps, and Interpretations. Technical Report 81, January 2001. MIT Laboratory of Computer Science.

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