Faithful mapping of model classes to mathematical structures*

Ádám Darvas and Peter Müller

ETH Zurich
Clausiusstrasse 59
CH-8092 Zurich
Switzerland
{adam.darvas,peter.mueller}@inf.ethz.ch

Abstract

Abstraction techniques are indispensable for the specification and verification of the functional behavior of programs. In object-oriented specification languages like JML, a powerful abstraction technique is the use of model classes, that is, classes that are only used for specification purposes and that provide object-oriented interfaces for essential mathematical concepts such as sets or relations.

While the use of model classes in specifications is natural and powerful, they pose problems for verification. Program verifiers map model classes to their underlying logics. Flaws in a model class or the mapping can easily lead to unsoundness and incompleteness.

This article proposes an approach for the faithful mapping of model classes to mathematical structures provided by the theorem prover of the program verifier at hand. Faithfulness means that a given model class semantically corresponds to the mathematical structure it is mapped to.

Our approach enables reasoning about programs specified in terms of model classes. It also helps in writing consistent and complete model-class specifications as well as in identifying and checking redundant specifications.

1 Introduction

Abstraction is indispensable for the functional specification and verification of object-oriented programs. Without abstraction, types without an implementation (interfaces or abstract classes) cannot be specified. Abstraction is also necessary to support subtyping and information hiding.

One way of expressing data abstraction in specification languages is by relating implementations to mathematical structures such as sets and tuples. This approach was pioneered by the Larch project [1], which advocated two-tiered specifications consisting of a contract and a theory providing the mathematical structures.

The Java Modeling Language (JML) unifies these tiers to simplify the development of specifications [2]. Instead of using a separate language to describe mathematical structures, JML describes them in an object-oriented manner through **model classes**. These classes contain only pure (side-effect free) methods. Therefore, they can be used in specification expressions.

Figure 1 shows the use of model class `JMLObjectSet` for the specification of the class `SingletonSet`.[1] The model class, through its pure methods, encodes a mathematical set of objects. In order to use the model class, a model field `_set` is declared. It represents the abstraction of an instance of type `SingletonSet` as specified by the **`represents`** clause. One can specify `SingletonSet`'s method `setValue` in an abstract way (that is, without referring to the private field `value`) using model field `_set` and `JMLObjectSet`'s pure method `has`, which checks for set membership.

*This paper is a postprint of a paper submitted to and accepted for publication in IET Software and is subject to Institution of Engineering and Technology Copyright. The copy of record is available at IET Digital Library.

[1] A brief summary of the most important JML constructs is given in Section 2.1. Readers who are not familiar with JML might want to read that section first.

```
package java.util;
//@ import org.jmlspecs.models.JMLObjectSet;

public class SingletonSet {
  private Object value;
  //@ public model JMLObjectSet _set;
  //@ private represents _set <- new JMLObjectSet(value);

  /*@  public normal_behavior
    @    ensures _set.has(o);
    @    assignable _set;
    @*/
  public void setValue(Object o) { value = o; }

  // other constructors and methods omitted
}
```

Figure 1: Specification of type `SingletonSet` using model class `JMLObjectSet`.

While model classes are useful for specification purposes, they pose problems for verification. Program verifiers have to encode model classes in the underlying theorem prover. This can be done by encoding pure methods and their contracts by uninterpreted function symbols and axioms, respectively [3, 4, 5, 6]. However, this approach is not optimal for model classes because the tactics of a theorem prover are optimized for theories that are part of the prover's theory-library rather than for encodings of JML model classes. Moreover, it is difficult to ensure soundness of such encodings, especially in the presence of recursive specifications [3].

To overcome these problems, previous work [7, 8, 9] proposes to map model classes and their pure methods directly to theories of the theorem prover. This is possible because model classes are in fact very similar to mathematical structures. Their objects are immutable, all their operations are side-effect free, and equality is based on their state rather than object identity. Therefore, instances of model classes behave like mathematical values rather than heap-allocated objects. This view greatly simplifies reasoning about model classes, in particular, because it is typically not necessary to make the heap explicit in the reasoning.

For instance, the specification expression `_set.has(o)` in the specification of method `setValue` in Figure 1 can be mapped to $o \in \_set$. By mapping the expression `new JMLObjectSet(value)` in the **`represents`** clause for `_set` to the singleton set $\{\texttt{value}\}$, we obtain the following proof obligation for `setValue`: $o \in \{o\}$, which is trivial to prove. This proof obligation is considerably better suited for verification than the proof obligation $\hat{has}(\hat{JMLObjectSet}(o), o)$ that one would obtain by encoding each method and constructor `m` of the model class by a function symbol $\hat{m}$ and the corresponding axioms.

However, existing work only discusses the mapping of method signatures, but ignores their contracts. With this approach, the meaning of `has` is given by the definition of symbol '$\in$', and not by the contract of `has`. This is problematic if there is a mismatch between the contract and the semantics of the operation given by the theorem prover: static program verifiers might produce results that come unexpected for programmers who rely on the model class contract. The results may also vary between different theorem provers, which define certain operations slightly differently (for instance, division by zero yields 0 in Isabelle [10], while it is not admissible in PVS [11]). Moreover, runtime assertion checking might diverge from static verification if the model class implementation used by the runtime assertion checker is based on the model class contract.

To illustrate the possible discrepancies between static verification and runtime assertion checking, assume that we have the following specification for one of the constructors of `JMLObjectSet`:

```
/*@ public normal_behavior
  @  ensures (e == null ==> \result.isEmpty())
  @       && (e != null ==> \result.has(e));
  @*/
public JMLObjectSet(/*@ nullable @*/ Object e);
```

That is, the constructor returns an empty set if the argument is `null`, and a set that contains `e` otherwise.

With this behavior of the constructor, the runtime assertion checker reports an error for the call `mySingletonSet.setValue(null)` because in the post-state of `setValue`, the set `_set` is empty and, thus, does not actually contain `null`. That is, `_set.has(null)` yields false. However, as we have argued above, the proof obligation obtained by applying the mapping can be trivially proven. This discrepancy is due to the fact that mapping `JMLObjectSet(/*@ nullable @*/ Object e)` to {e} alters the semantics of the specification in the case when `e` is `null`.

In this article, we show how model classes can be mapped to theorem provers without semantic mismatches. The main contribution of our work is a technique for proving that the mapping of a model class to a mathematical structure defined by the theorem prover is faithful. **Faithfulness** means that the model class and the structure indeed correspond to each other in their properties. To show faithfulness, we prove formally that the mapping is consistent and complete. **Consistency** means that everything that can be proved using the contracts of the model class can also be proved using the corresponding structure of the theorem prover. **Completeness** means that everything that can be proved using the structure defined by the theorem prover can also be proved using the contracts. Once faithfulness of a model class has been proven, the mapping can be used for program verification without worrying about semantic discrepancies. In particular, the above discrepancy between the static verification of `setValue` and runtime assertion checking is detected during the consistency proof for `JMLObjectSet`.

Our approach leads to important results beyond semantical correspondence and simplified reasoning. Model class contracts are complex and can easily get inconsistent, which can lead to unsound reasoning. Showing that a model class can be mapped consistently to a mathematical structure proves that the model class contract itself is consistent (provided that the structure is well-defined). In fact, one of our case studies discovered an inconsistent specification in one of the most basic model classes of JML.

This shows that proving faithfulness of mappings helps in writing better specifications for model classes by making them consistent and complete. Our approach can also be used to identify redundant parts of specifications as well as to check whether specifications marked as redundant are indeed derivable from non-redundant specifications. These capabilities further improve the quality of model-class specifications.

Throughout the article we will use JML [12] as specification language and Isabelle [10] as target theorem prover. However, the presented approach is applicable to any combination of specification language and theorem prover, for instance, Eiffel [13] and Coq [14].

This article extends our earlier paper [15] by more details, explanations, examples, and, in particular, the coverage of inductive data types, which are a common way of specifying mathematical structures in theorem provers. Inductive data types come with a variety of properties such as the induction principle that can typically not be derived from JML specifications. Therefore, a different proof technique is required to show completeness of a JML model class w.r.t. an inductive data type.

The rest of this article is structured as follows. Section 2 describes the essential features of JML and Isabelle that are referred to in this article. Furthermore, a JML model class and an Isabelle theory is introduced, through which we illustrate the main concepts of our approach. Section 3 presents our solution for the faithful mapping of model classes to mathematical structures defined by a theorem prover. Section 4 presents a case study we performed on model class `JMLObjectSet`. Section 5 extends our approach to handle model classes that are mapped to inductive data types. Section 6 demonstrates the extension by a second case study on model class `JMLObjectSequence`. Section 7 gives an overview of related work, and in Section 8 we conclude.

2 Preliminaries

In this section we briefly introduce those constructs of JML and Isabelle that are used in later sections. We also introduce a JML model class and an Isabelle theory that will be used as running example later.

2.1 The Java Modeling Language

JML is a behavioral interface specification language that follows the paradigm of Design by Contract [16] and is specifically tailored to the Java programming language. JML specifications are embedded in Java comments that start with an at-sign (@). Thus, Java programs annotated with JML specifications can still be processed by Java compilers.

One of the key design decisions behind JML is that the syntax of specification expressions is based on side-effect free Java expressions, extended with a few additional constructs. Thus, programmers need not learn a formal language from scratch but can get acquainted with the language with significantly less effort.

Method specifications. The functional behavior of methods[2] is specified by preconditions, postconditions, exceptional postconditions, and assignable clauses. The precondition (**`requires`** clause) of a method specifies what must hold when the method is called. The postcondition (**`ensures`** clause) specifies what must hold if and when it terminates. The exceptional postcondition consists of two clauses: (1) the **`signals_only`** clause specifies what type of exceptions may be thrown by the method; (2) the **`signals`** clause specifies what must hold at the point when a certain type of exception is thrown. The **`assignable`** clause of a method specifies which locations the method may modify. All other locations must remain unchanged. Constructors need not mention locations of the object being initialized.

JML provides different kinds of method specification cases of which the most general one is the "behavior specification case". It may contain all five clauses mentioned above. A "normal behavior specification case" (**`normal_behavior`** clause) specifies method behavior in which methods may not throw exceptions. It is syntactic sugar for a behavior specification case to which the "**`signals`** (Exception) **`false`**;" clause is added. An "exceptional behavior specification case" (**`exceptional_behavior`** clause) specifies method behavior in which methods must throw exceptions. It is syntactic sugar for a behavior specification case to which clause "**`ensures false`**;" is added. Note that a method may have both a normal and an exceptional behavior specification case. If so, the preconditions of the cases must be disjoint, otherwise the behavior is not implementable.

JML extends Java with the modifier **`pure`** that marks methods to be side-effect free. Thus, the pure modifier can be thought of as a short-hand for the clause "**`assignable \nothing`**". When a constructor is marked to be pure then it may only have side-effects on the object being initialized. Due to their side-effect freeness, pure methods may be invoked in specification expressions.

Another JML modifier applicable on method declarations is **`model`**. It marks methods to be specification-only, that is, calls to model methods must not appear in implementations.

Type specifications. Object invariants (**`invariant`** clause) specify the consistent states of objects. In JML, they specify properties that must hold in all **visible states**, that is, in all pre- and post-states of method executions [17].

The **`pure`** modifier applied to a type means that all its instance methods are pure. The **`model`** modifier applied to a field declaration marks the field to be used for specification purposes only. The relation between the value of a model field and the values of other (possibly model) fields is determined by the **`represents`** clause, which allows one to write abstraction functions [18].

The semantics of JML does not allow, by default, declared fields, method parameters, and return values of reference type to be **`null`**. To allow them to take the **`null`** value, one has to use the **`nullable`** modifier in field declarations and method signatures.

Invariants and method specifications can be marked redundant by the **`implies_that`** keyword, pre- and postconditions by the **`requires_redundantly`** and **`ensures_redundantly`** keywords, respectively. Specification elements marked as redundant should be derivable from other, non-redundant specifications.

The **`import`** declaration in specifications allows one to refer to types that are used for specification purposes only, for instance, model classes.

At the time of writing, **`immutable`** was not part of the JML syntax. Still, for the purpose of this article we will use it as a modifier on types to specify that instances of such types are immutable. This is particularly important for model classes, which can be thought of as mathematical structures.

Expressions. JML's expression syntax extends Java's side-effect free expression syntax. The most important extensions are additional logical connectives (for instance, implication `==>` and equivalence `<==>`), the **`\old`** construct used in postconditions to refer to values in pre-states, and the **`\result`** construct used in postconditions of non-void methods to refer to the return value. JML also adds

[2] In the sequel, we include constructors when referring to "methods". We will distinguish between constructors and methods explicitly only when specifically needed.

quantifiers `\forall` and `\exists`. The syntax of universal quantification is (`\forall x;` $\mathtt{E}_R$ `;` E), where both $\mathtt{E}_R$ and E are predicates and $\mathtt{E}_R$ defines the range of the quantification for which E must hold. If omitted, $\mathtt{E}_R$ defaults to true.

2.2 An example model class: `JMLObjectSet`

As an example to demonstrate some of the features of JML, we take model class `JMLObjectSet`, which is part of the model library of the JML distribution [19]. The class encodes sets of objects. That is, it provides the usual operations of mathematical sets and equality of elements is based on Java's reference equality ("`==`"). Figure 2 presents the class with those members that we discuss in this article. Other methods and specification elements are omitted for brevity.

The class is specified to be pure, thus, all its instance methods are pure. Furthermore, the class is specified to be immutable. Thus, methods that return `JMLObjectSet`s (for instance, `union`) do not mutate their receiver objects but return new instances. Since no arguments or return values are marked to be nullable, all reference type arguments and return values are considered to be non-null.

The class is specified by method specifications and an invariant. Two sample normal behavior specification cases are given for methods `insert` and `union`. Note that assignable clauses are not needed since all methods are pure. The invariant expresses that method `equational_theory` has to return true in every visible state for every non-null `JMLObjectSet` instance `s2`, and objects `e1` and `e2`. Method `equational_theory` is a static pure model method and has a large normal behavior specification case containing equations written in the style of algebraic laws. As a result, the invariant of the model class prescribes that all equations specified by the method specification of `equational_theory` have to hold in all visible states. For brevity, Figure 2 shows only a sample equation defining method `union`.

This way of writing invariants is typical for model classes and thus, such invariants are commonly referred to as the **equational theory** of the class. We will use this terminology in the sequel, too. Furthermore, we will use the term "invariant" when referring to an equation prescribed by the specification of method `equational_theory`, for instance, to the equation that defines method `union`.

The proposed mapping of the class and its methods to one of Isabelle's set structures is given by the `mapped_to` clauses that we introduce in this article.

We follow the proposal of Leavens et al. [8] and Charles [7], and consider model classes to be final and unrelated to Java's type hierarchy rooted in class `Object`. This prevents problems related to inheritance, method overriding, and dynamic dispatch. In the realm of model classes, these restrictions seem acceptable since model classes are supposed to describe elementary mathematical concepts and to be used only for specification purposes.

2.3 Isabelle/HOL and theory `HOL/Set`

Isabelle [20] is a generic interactive LCF-style theorem prover. It is generic in that it provides a metalogic, which allows one to implement different logical formalisms, for instance, FOL (first-order logic) and ZF (Zermelo-Fraenkel set theory). This article uses Isabelle/HOL [10, 21], which is the specialization of Isabelle for the logical system of higher-order logic. In the sequel, we will interchangeably use the term "Isabelle" to refer to both Isabelle, the theorem prover, and Isabelle/HOL, the logical system.

Isabelle is interactive, that is, users have full control over the process of proving theorems. However, several automatic decision procedures and tableaux provers (collectively called **tactics**) are available, which significantly simplifies and accelerates the proof process. Two of the often used tactics are **simp**, which is based on term rewriting; and **auto**, which is a tableaux prover.

Isabelle is an LCF-style theorem prover, that is, it is based on a small logical core [22]. Everything else is supposed to be defined on top of this core by conservative extensions [23], which ensure its logical consistency. Although Isabelle allows users to state axioms, it is strongly discouraged due to the risk of making the specification inconsistent.

When working with Isabelle, users write theories. Theories typically consist of type and function-symbol declarations, and of definitions and theorems over these types and symbols.

An example theory. We guide the reader through a snippet of theory `HOL/Set` [24] in order to introduce the constructs that are necessary for the understanding of the article. The relevant parts of

```
package org.jmlspecs.models;
//@ immutable
//@ mapped_to("Isabelle", "HOL/Set", "α set");
public final /*@ pure @*/ class JMLObjectSet {
  /*@ public invariant
    @   (\forall JMLObjectSet s2; s2 != null;
    @     (\forall Object e1, e2; ;
    @       equational_theory(this, s2, e1, e2))); */
  /*@ public normal_behavior
    @   ensures \result <==>
    @     (s.union(s2)).has(e1) == (s.has(e1) || s2.has(e1));
    @   ...
    @ static public pure model boolean
    @   equational_theory(JMLObjectSet s, JMLObjectSet s2,
    @                     Object e1, Object e2); */

  //@ mapped_to("Isabelle", "{}");
  public JMLObjectSet();

  //@ mapped_to("Isabelle", "insert e {}");
  public JMLObjectSet(Object e);

  //@ mapped_to("Isabelle", "elem : this");
  public boolean has(Object elem);

  //@ mapped_to("Isabelle", "this = s2");
  public boolean equals(Object s2);

  //@ mapped_to("Isabelle", "this = {}");
  public boolean isEmpty();

  public int int_size();

  //@ mapped_to("Isabelle", "this <= s2");
  public boolean isSubset(JMLObjectSet s2);

  //@ mapped_to("Isabelle", "this < s2");
  public boolean isProperSubset(JMLObjectSet s2);

  //@ mapped_to("Isabelle", "SOME x. x : this");
  public Object choose();

  /*@ public normal_behavior
    @   ensures (\forall Object e; ; \result.has(e) <==>
    @     (this.has(e) || e == elem)); */
  //@ mapped_to("Isabelle", "insert elem this");
  public JMLObjectSet insert(Object elem);

  //@ mapped_to("Isabelle", "this - (insert elem {})");
  public JMLObjectSet remove(Object elem);

  /*@ public normal_behavior
    @   ensures (\forall Object e; ; \result.has(e) <==>
    @     (this.has(e) || s2.has(e))); */
  //@ mapped_to("Isabelle", "this Un s2");
  public JMLObjectSet union(JMLObjectSet s2);

  //@ mapped_to("Isabelle", "this - s2");
  public JMLObjectSet difference(JMLObjectSet s2);
}
```

Figure 2: Model class `JMLObjectSet` containing the signatures of methods we consider in this article.

```
theory Set
imports LOrder
begin

typedecl α set

consts
  "{}"         :: "α set"
  insert       :: "α  ⇒ α set ⇒ α set"
  Collect      :: "(α  ⇒ bool) ⇒ α set"
  Un           :: "α set ⇒ α set ⇒ α set"
  "op :"       :: "α  ⇒ α set ⇒ bool"

translations
  "{x. P}"     == "Collect (λ x. P)"

axioms
  mem_Collect_eq: "(a : {x. P(x)}) = P(a)"
  Collect_mem_eq: "{x. x:A} = A"

defs (overloaded)
  subset_def:  "A <= B     == ∀ x. x:A ⇒ x:B"
  psubset_def: "A < B      == A <= B & ~A=B"
  set_diff_def: "A - B     == {x. x:A & ~x:B}"

defs
  Un_def:      "A Un B     == {x. x:A | x:B}"
  empty_def:   "{}         == {x. False}"
  insert_def:  "insert a B == {x. x=a} Un B"

lemma subsetI: "(∀ x. x:A ⇒ x:B) ⇒ A <= B"
  by (simp add: subset_def)

end
```

Figure 3: Snippet of Isabelle theory `HOL/Set` containing signatures, axioms and definitions we consider in this article.

the theory are presented in Figure 3.

The theory begins by importing another theory `HOL/LOrder`, which means that all symbols, definitions, axioms, and theorems that are available in `HOL/LOrder` are also available in `HOL/Set`. Next, a new type α *set* is introduced, where α is a type variable that gives rise to polymorphic types [10].

Keyword **`consts`** declares five new function symbols for the empty set, element insertion, set comprehension, union, and set membership, respectively. Union and set membership are defined to be infix operators—not shown in the figure to avoid unnecessary syntactic clutter.[3]

The constant declaration is followed by the definition of syntactic sugar for set comprehension and the axiomatization of set comprehension. The axioms are followed by definitions; the first group defines the semantics of symbols that are not introduced by theory `HOL/Set` but are already available in the imported theory `HOL/LOrder`. These are subset, proper subset, and set difference. The second group defines the newly introduced symbols—except the already axiomatized set comprehension.

Finally, a theorem is stated (introduced by keyword **`lemma`**) and proven using the **simp** tactic, which uses a pre-defined set of axioms, definitions, and theorems. For the proof of the theorem, this set is extended by definition `subset_def`.

[3]We also omitted other, rather cryptic parts to ease understanding. As a result, the presented specification does not parse with Isabelle.

3 Faithful mapping

In this section, we present our solution for proving that the mapping of a model class M to a mathematical structure S (as defined by some theory) is faithful. That is, there is a semantic correspondence between M and S, namely, they are **isomorphic**. Here, we assume that structure S is defined non-inductively. Inductive structures are discussed in Section 5.

The process consists of three stages. In the first stage, we specify the mapping of M to S by the proposed `mapped_to` clause. Then we prove consistency and completeness of the specified mapping in the second and third stage, respectively. In this section, we present the details of these stages.

3.1 Specifying the mapping

In the first stage, one has to decide how to map model class M. That is, one has to specify (1) to what structure S is the model class mapped; and (2) to which function symbols of S are the methods of the model class mapped.

The mapping of a model class is specified by attaching to it the specification construct `mapped_to` that we propose in this article. The first parameter specifies the target environment, the second the target context, and the third the specific type in the context to which the model class is mapped. Figure 2 shows a possible mapping of model class `JMLObjectSet`: it is mapped to the Isabelle theorem prover, namely to its `HOL/Set` theory [24], more specifically to type α *set*. Alternatively or additionally, the class could be mapped to other provers such as PVS and Coq. In these provers, sets are represented as predicates [25, 26], that is, functions from a given type to booleans. A possible mapping is:

```
//@ mapped_to("PVS", "sets[Object]", "set");

//@ mapped_to("Coq", "Coq.Sets.Classical_sets", "Ensemble");
```

The mapping of methods is specified by `mapped_to` clauses attached to the methods. The first parameter of the clause is again the target environment and the second specifies the way the model-class method is mapped to some term in the target context. For instance, in Figure 2, `JMLObjectSet`'s `has` method is mapped to Isabelle's set membership ':'. In PVS and Coq, the following mapping could be chosen for the method:

```
//@ mapped_to("PVS", "member(elem, this)");

//@ mapped_to("Coq", "In this elem");
```

The second parameter of `mapped_to` clauses typically mentions function symbols of the target context as well as parameters (including the receiver) of the method being specified. Note, however, that we permit arbitrary terms of the target context; this flexibility allows us to specify mappings even if the target theorem prover does not provide a structure that directly corresponds to the model class being mapped. An example where this flexibility is necessary is `JMLObjectSet`'s `remove` method, which removes a single element of a set. The theory `HOL/Set` does not have a corresponding operation; thus, if our approach allowed only direct correspondence between methods and function symbols, then this simple operation could not be handled with the chosen mapping. See Section 4.3 for further details on the mapping of the method.

We call the mapping of some method `m` with parameters **`this`**, $\mathtt{p}_1$, ..., $\mathtt{p}_n$ to some term T **direct** if and only if T is of the form $f(a_1, \ldots, a_m)$, where f is some function symbol, $m \geq 0$, and each a_i is either one of the parameters of `m` or a constant symbol. For instance, the mappings of the two constructors of model class `JMLObjectSet` are direct: `JMLObjectSet()` is mapped to the constant symbol $\{\}$, and `JMLObjectSet(Object e)` is mapped to function application *insert e* $\{\}$. However, the mapping of method `remove` is not direct: it is mapped to the term *this* $-$ *(insert elem* $\{\}$*)*, where $-$ is an infix operator. Since the second argument contains the non-constant symbol *insert*, the mapping is not direct.

To support multiple theorem provers, multiple `mapped_to` clauses may be attached to the model class and its methods. This is needed since different theorem provers provide different theories with different

function symbols and syntax for the same functionality. Thus, the isomorphism proof has to be carried out in every target theorem prover specified in `mapped_to` clauses.

Important to note is that mappings need not be specified by programmers who are typically not familiar with theorem provers and their theories and syntax. They can be specified by the author of a model class or by the team that performs the verification. The same applies also to the consistency and completeness proofs, which typically require experience in theorem proving.

The proposed `mapped_to` clause is only used for the purpose of static verification. For runtime assertion checking the clause can be ignored.

3.2 Consistency

Once the mapping is specified, its faithfulness has to be proven. For each theorem prover, this proof needs to be carried out only once. Afterwards, any verification system can make use of the specified mapping to handle model classes in specifications. In this subsection, we describe how to prove consistency of the mapping, that is, we prove that the properties of model class M (as specified by its contracts) can be derived from the properties of structure S (as defined by axioms and definitions).

In order to prove consistency, one has to encode the method specifications and invariants of M in the language of S based on the `mapped_to` clauses and prove the resulting formulas using the properties of S. In fact, not all method specifications have to be translated and proved but only the ones that specify the normal behavior of a given method. Behavior and exceptional behavior specifications cases describe situations when the method might throw exceptions. We ignore such cases for the isomorphism proof because we assume that all JML specifications are well-defined [6], that is, exceptions are guaranteed never to be thrown.

In the sequel, we use the term **relevant specification element** to refer either to an invariant or to a normal behavior method specification case of a model class. Every relevant specification element s_M in M needs to be translated and proved as follows:

1. (a) If s_M is a method specification of a method m with precondition *pre* and postcondition *post* then it is translated into a formula of the form "$pre \Rightarrow post$", which is universally quantified over all parameters (including the implicit receiver **`this`**) of m. Note that it is not necessary to conjoin M's invariant to the pre- and postcondition because invariants of model classes do not restrict the state space of their instances, but rather give equational laws about their operations. These laws are dealt with separately when the invariant is translated as one of the relevant specification elements.
   (b) Occurrences of every method call in s_M to some method m are replaced by the term prescribed in the `mapped_to` clause of method m. For simplicity, we assume that JML's logical operators are also method calls with implicit mappings to the underlying theorem prover (e.g., JML's `==>` operator is mapped to logical implication) and that this implicit mapping is faithful.
   (c) If s_M is a method specification of a method m, then in the postcondition all occurrences of **`\result`** (**`this`** if m is a constructor) are replaced by the term prescribed in the `mapped_to` clause of method m.

2. The formula that is derived in step 1 is turned into a lemma and proved in the theorem prover specified by the `mapped_to` clause using the axioms, definitions, theorems, etc. of S.

We demonstrate this process on `JMLObjectSet`'s `insert` method. Its specification is presented in Figure 2.

In step 1(a), the postcondition gets universally quantified over the parameters of `insert`, *this* and *elem*. In step 1(b), the two method calls to `has` get replaced by Isabelle's set membership operator ':' as prescribed by the `mapped_to` clause of `has` in Figure 2. This yields terms "e : **`\result`**" and "$e : this$". Additionally, the logical operators **`\forall`**, `<==>`, `||`, and `==` get replaced by the corresponding Isabelle operators $\forall$, $=$, $\vee$, and $=$, respectively. Step 1(c) replaces **`\result`** by "$insert\ elem\ this$" as prescribed by the `mapped_to` clause of method `insert`. This yields the following formula:

$$\forall\ this, elem.\ \forall e.\\ (e : (insert\ elem\ this)) = ((e : this)\ \vee\ (e = elem))$$

In step 2, the formula is turned into a lemma in Isabelle. Its proof can be completed automatically by the **auto** tactic. This is not surprising since theorem provers like Isabelle are typically well-equipped with theorems over structures such as sets.

Completing this stage successfully for every relevant specification element in model class M gives us the guarantee that whatever can be proven using the properties of M also can be proven using S. An important consequence of this result is that the specification of M is consistent (that is, free from contradictions), provided that S is consistent. Since structures like Isabelle's `HOL/Set` are defined using conservative extensions and have been reviewed by many people, it is rather unlikely that they contain inconsistencies. In other words, in this stage we prove that Isabelle's `HOL/Set` theory is a **model** for the axioms extracted from the specification of model class `JMLObjectSet`. By exhibiting this model, we prove that the specification of `JMLObjectSet` is consistent.

This is also an interesting result concerning the use of pure methods in specification expressions. As we have shown earlier [3], the use of pure methods in specifications can easily lead to unsoundness. The solution we proposed in our earlier work to prevent unsoundness was to exhibit a witness for showing that the specification of the method is satisfiable. Recursive specifications require a proof of well-foundedness [6], which can be difficult for pointer structures. With the approach presented above, recursive specifications do not pose any problems because the consistency of the mapping of a method m to a well-defined element of S implies that m is also well-defined, no matter whether it is specified recursively or not.

As the example of method `insert` suggests, proving this stage may be fully automated: (1) the lemma was generated following three simple steps performing syntactic replacements based on `mapped_to` clauses, and (2) the lemma was proved without any user interaction using a powerful tactic of Isabelle. We note, however, that for more complicated properties the mere use of tactics does not suffice and one has to help the prover by giving hints at which theorems of S to use.

Consistency and sound verification

Although proving consistency is sufficient to show that the specification of a model class is free from contradictions, we demonstrate by an example that it is not sufficient for the use of `mapped_to` clauses for verification purposes.

Assume model class `JMLObjectSet` had a method `bogusUnion` that was mapped to Isabelle's set union operation and was specified as follows:

```
/*@ public normal_behavior
  @   ensures !this.isEmpty() ==> !\result.isEmpty();*/
//@ mapped_to("Isabelle", "this Un s2");
public JMLObjectSet bogusUnion(JMLObjectSet s2)
```

Consistency of the specification can be proven trivially since the specified property holds for Isabelle's union operation *Un*, too. However, consider the following expression:

```
new JMLObjectSet().
  bogusUnion(new JMLObjectSet(e)).has(e)
```

The specification of `bogusUnion` does not allow us to determine whether the expression is true or false, while the corresponding Isabelle formula "$e \ : \ (\{\}\ Un\ (insert\ e\ \{\}))$" is trivially provable. Thus, it would be unsound for a program verifier to map method `bogusUnion` to symbol *Un* since we would allow more properties to be proven after the mapping than we could prove before the mapping using the model-class specifications. For instance, the specification of the method allows the implementation `return this`, which clearly violates the properties of set union.

Intuitively, the problem is that the specification of `bogusUnion` only describes one particular property of what we understand under the mathematical union of sets. However, after having mapped `bogusUnion` to Isabelle's *Un* symbol, the method is bequeathed all the additional properties that symbol *Un* has. For the faithful mapping of `bogusUnion`, we need to show that the method indeed possesses these bequeathed properties. This is the task of the next stage, where we prove completeness, which often requires one to strengthen specifications of model classes.

3.3 Completeness

In the third stage, we complete the isomorphism proof by showing completeness of the mapping, that is, that the properties of structure S can be derived from the properties of model class M. The proof procedure is as follows:

1. Each method m of M is turned into a function symbol $\hat{m}$ and its signature is declared based on m's signature.

2. Each relevant specification element s_M in M is turned into a formula in a similar way as described in step 1 of the translation for the consistency proof (Section 3.2). The only difference is that in step 1(b), calls to some method m are not replaced according to m's `mapped_to` clause, but by an application of function $\hat{m}$. Analogously, `\result` (and `this`) are also replaced by function applications in step 1(c).

   The resulting formula is then stated as axiom.

3. A lemma is generated from every definition and axiom s_S of S[4] by replacing all occurrences of function symbols in s_S by some corresponding term, which may contain function symbols declared in step 1. Since `mapped_to` clauses specify the mapping from M to S, finding the **corresponding term** is non-trivial unless the clause specifies a direct mapping between function symbols of M and S. When several methods of M are mapped to a function symbol of S then one lemma is generated for each possible backward mapping, see details below.

4. The lemma is proven using the axioms generated in step 2.

As an example, we show this procedure for Isabelle's definition of proper subsets. In the first two steps, we introduce function symbols and axioms for all methods of the model class. Here, we focus on the only method needed for proper subsets, `isProperSubset`. In the first step, the signature of $\hat{isProperSubset}$ is declared as follows:

$$\hat{isProperSubset} : \ \alpha\ set \ \times \ \alpha\ set \Rightarrow bool$$

The second step is based on the specification of the method. For demonstration purposes, this time we take the specification prescribed by the invariant, that is, the equation given in the specification of method `equational_theory`:

```
s.isProperSubset(s2) == (s.isSubset(s2) && !s.equals(s2))
```

Since the equation is part of the method specification of method `equational_theory`, first it gets quantified over its parameters: s, $s2$, $e1$, and $e2$. Then method calls to `isProperSubset`, `isSubset`, and `equals` are translated into the function applications $\hat{isProperSubset}(s, s2)$, $\hat{isSubset}(s, s2)$, and $\hat{equals}(s, s2)$, respectively. The resulting formula is turned into the following axiom:

$$\begin{aligned} \forall\ s, s2, e1, e2.\ \hat{isProperSubset}(s, s2) = \\ (\hat{isSubset}(s, s2) \wedge \neg \hat{equals}(s, s2)) \end{aligned} \tag{1}$$

In step 3, we take the definition of proper subsets from Isabelle's theory [24]:

```
psubset_def: "A < B == A <= B & ~A=B"
```

and translate it to the following lemma:[5]

$$\begin{aligned} \forall\ A, B.\ \hat{isProperSubset}(A, B) = \\ (\hat{isSubset}(A, B) \wedge \neg \hat{equals}(A, B)) \end{aligned}$$

Note that in this case, the backward mapping (that is, "finding the corresponding term" in step 3) is trivial since there is a direct mapping between the three JML and Isabelle function symbols.

In step 4, the lemma needs to be proven using the axioms defined in step 2. The proof is trivial since axiom (1) (derived from the equational theory) is equivalent to the lemma.

Proving this stage guarantees that whatever can be proved using the axioms, definitions, and theorems of Isabelle's `HOL/Set`, can also be proved using JML's `JMLObjectSet`. An interesting consequence is that

[4]The set of definitions and axioms is identified by syntactically inspecting the target context, for instance, an Isabelle theory.

[5]Isabelle definitions are implicitly universally quantified over all free variables.

we have proved that the axiom system extracted from the specifications of `JMLObjectSet` is complete relative to Isabelle's `HOL/Set` theory. Since Isabelle structures like `HOL/Set` are heavily used in formalizations and proofs, one can be confident that they contain all important properties of the structure.

As noted above, the generation of lemmas in step 3 is not as trivial as for the consistency proof because the backward mapping from S to the model class is not explicitly specified. Furthermore, proving the lemmas in step 4 is also less trivial than for the consistency proof. First, even the application of automated tactics typically requires one to manually select the set of axioms to be used for proving a given lemma because selecting all axioms might cause the tactic to loop. Second, beyond trivial cases, tactics often fail to find the proper instantiations of axioms that are generated in step 2. In such cases, further hints need to be given to the prover, for instance, by the insertion of simple helper lemmas. In practice, we also found that the specification of the model class may be too weak to verify some axioms or definitions of the structure. In this case, the missing specifications need to be identified and added to the model class. Thus, it seems that the automation of this stage can, in general, only be partial, and manual intervention is needed. However, the effort is justified by the increased quality of the model class specification.

An interesting side-effect of this proof technique is that redundant specifications can be discovered in the model class, and one can check whether specification elements marked as redundant are indeed implied by other specification elements. If an axiom is never used in the completeness proof then the specification element from which the axiom was derived is redundant in the model class. To check redundancy of a specification element, we follow step 1 and 2 as before, but generate lemmas (and not axioms) out of specification elements that are marked as redundant. Lemmas that are provable using the axioms, generated from the non-redundant specification elements, confirm that the corresponding specification elements are indeed redundant. The identification and checking of redundant specifications further improve the quality of model-class specifications.

Issues with backward mappings

It is crucial to note that the completeness proof should not only show that the model class provides all the functionality that the Isabelle structure provides. It should also show that the related JML methods and Isabelle functions indeed correspond to each other. For instance, if model class `JMLObjectSet` provides several methods to insert an element into a set then all of them must have the properties of *insert*, even though one method with these properties would be sufficient to obtain the expressiveness of the structure. We achieve this strong notion of completeness by applying all possible backward mappings that can be derived from the specified `mapped_to` clauses during the process of lemma generation (step 3). In the following, we describe the three situations in which the backward mappings are non-trivial. These situations are not exclusive, that is, their occurrence may overlap.

Multiple mappings. A given Isabelle function may be mapped to by several model methods. Assume model class `JMLObjectSet` contained both method `union` as presented in Figure 2, and method `bogusUnion` as presented in Section 3.2. We can show that the model class has the functionality of set union by applying the trivial backward mapping of symbol *Un* to method `union`. However, this is not sufficient for the use of the mappings: if we do not additionally apply the backward mapping of *Un* to `bogusUnion`, then the discrepancy described in Section 3.2 would still remain. Thus, every occurrence of symbol *Un* in s_S has to be mapped back both to `union` and `bogusUnion` when generating lemmas (step 3). This yields multiple generated lemmas from a single Isabelle definition or axiom. In step 4, the attempt to prove some of the lemmas for the mapping of *Un* back to `bogusUnion` will fail. That is, some of the properties of *Un* are not derivable from the specification of `bogusUnion`. This points out the bogus mapping of the method.

Mappings with mismatching parameters. Model classes often offer "convenience methods", methods that are redundant in the sense that they are equivalent to some compound expression consisting of calls to more basic model methods. Such methods make the use of model classes more convenient, but their functionalities are often not directly available in the corresponding structures leading to a mismatch in the parameters of the model method and the related function.

An example is constructor `JMLObjectSet(Object e)`, which is equivalent with creating an empty set and then inserting element `e` into it: `new JMLObjectSet().insert(e)`. Accordingly, the constructor is

mapped to the term "*insert e* {}". Note that Isabelle symbol *insert* is mapped to by method `insert`, too. In fact, that mapping is direct, and thus the backward mapping is trivial. The faithful mapping of method `insert` to function *insert* can be easily proven, which implicitly also covers the case when the second parameter of *insert* is the empty set. However, for the same reason as for symbol *Un* above, it is not sufficient to map Isabelle symbol *insert* back only to model method `insert` in step 3. Additionally, we need to map it back to the constructor.

The issue with the backward mapping of *insert* to the constructor is that the former has two parameters while the latter has only one. Furthermore, term "*insert e* {}" does not appear in Isabelle definitions because no definition handles the special case when the second parameter is the empty set. Thus, we map every occurrence of *insert* back to the constructor and substitute parameters appropriately: (1) the general term "*insert e B*" that appears in Isabelle definitions and axioms s_S is to be mapped back to `JMLObjectSet(e)`, and thus replaced by function application $\hat{JMLObjectSet}(e)$[6]; (2) every occurrence of B is to be substituted by {}, the empty set. Since {} trivially maps back to `JMLObjectSet()`, occurrences of B in s_S are to be replaced by function application $\hat{JMLObjectSet}()$.

Indirect mappings. As mentioned in Section 3.1, `mapped_to` clauses may not only contain direct mappings, but may as well prescribe mappings to arbitrary terms in the language of the Isabelle structure. This is typically the case when the functionality of a model method is not directly supported by the related theory.

Consider `JMLObjectSet`'s `remove(Object elem)` method, which removes a single element `elem` of a set, and theory `HOL/Set`, which only supports set difference. Thus, the method is mapped to the term *this*−(*insert elem* {}). Note that this term contains three function symbols: "−" (set difference), *insert*, and {}. All three symbols can be directly mapped back to the model class (methods `difference`, `insert`, and `JMLObjectSet()`), however, proving the faithful mapping of these symbols does not necessarily imply that method `remove` expresses the functionality of removing an element from the set. To demonstrate the problem, assume that the only specification of the method was the following:

```
(\forall e; ; new JMLObjectSet().
   remove(e).equals(new JMLObjectSet()))
```

Consistency of the specification can be shown trivially, since it expresses a valid property given the above mapping of `remove`. If we were satisfied by the consistency and completeness proof of the three symbols used in the mapping, then we would conclude that the prescribed mapping for `remove` is faithful. However, the specification would admit, for instance, the implementation **`return new`** `JMLObjectSet()`, which would be obviously wrong.

This shows that we need to do more than just separately proving the faithful mapping of symbols used in an indirect mapping. The symbol on the top-most level of the mapping needs to be mapped back to the method in every Isabelle definition and axiom s_S. In our example this means that symbol "−" needs to be mapped back to method `remove`, while properly matching its arguments based on the `mapped_to` clause and its occurrences in s_S: The first parameter of symbol "−" appearing in s_S needs to be substituted by *this* and the second by term *insert elem* {}, which gets mapped back to parameter `elem` according to the prescribed mapping.

The term *insert elem* {} further needs to be mapped back since it is still in the language of Isabelle and not of the model class. As we have seen above, the term *insert elem* {} has two different backward mappings: to method `insert` and to constructor `JMLObjectSet(elem)`. However, now that we are in a recursive case of the backward mapping, we need not perform both mappings since above, when dealing with `JMLObjectSet(Object e)`, we have already proved that both of them are correct and thus equivalent.

[6]For simplicity, we assume that the functions that encode model class constructors (such as $\hat{JMLObjectSet}(e)$) yield a value that corresponds to the new, initialized object—in contrast to Java, where constructors take the new object as implicit parameter and do not have a result.

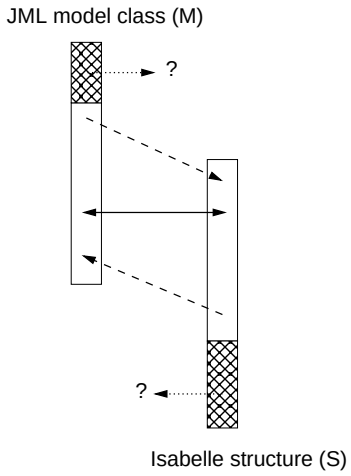

Figure 4: Mapping model methods to Isabelle functions and vice versa.

3.4 Summary

Successful completion of the three stages described above guarantees that model class M and structure S are isomorphic. This property confirms that the mapping prescribed by the `mapped_to` clauses are semantically correct.

The most important property from the consistency proof is that the axiom system extracted from the model class is consistent; thus, its usage cannot lead to unsoundness. This is obviously a crucial property for every verification system. For this stage, the generation and proving of lemmas seem to be automatable to a great extent. Failing to prove a lemma most probably indicates an error in the model class contract.

The most important result of the completeness proof is that the model class expresses the properties of the mathematical structure. This is important in order to prevent mismatches between the property one wants to express in a specification and the property one actually proves during the verification process. As noted above, the generation and proving of lemmas is not as trivial as for consistency.

Once both directions are successfully proven, method calls can be directly translated to the corresponding function applications without being worried about soundness issues or semantic discrepancies.

3.5 Mismatches between model class and structure

The interface of a model class is typically developed independently from the structures that it may potentially be mapped to. This is justified by the fact that model classes are used by different program verifiers equipped with different back-end theorem provers. Different theorem provers provide different structures and functions; thus, typically it is not possible to develop model classes such that they can be trivially mapped to all suitable structures.

There are mainly three different situations the mapping of a model class M to some structure S might lead to. These situations are illustrated by Figure 4.

The best situation is when all methods of M can be directly mapped to functions of S, depicted by a solid arrow. In this case, isomorphism of the related methods and functions can be proved and the lemma generation (step 3) for the completeness proof is rather straightforward, as described in Section 3.3.

Another possibility, depicted by dashed arrows, is that a method of model class M cannot be mapped directly to a function of structure S, but to a compound expression in the language of S. This situation may also occur in the reverse direction. Although isomorphism of related symbols can still be shown, the derivation of all backward mappings becomes non-trivial, as discussed in Section 3.3.

The worst situation is when no mapping can be given for a model method or a structure operation (dotted arrows in the figure). In such cases, there is a mismatch that cannot be bridged: M and S are not isomorphic. However, the "direction" of the mismatch makes a difference in the consequences.

If a method of M cannot be translated to S then we cannot be sure whether specifications that contain calls to the method are consistent, and whether the method semantically corresponds to some mathematical operation. In such situations, one needs to pick a different target structure where the mapping is possible.

The situation is better if an operation of S cannot be translated to M. Although isomorphism of M and S cannot be proven, isomorphism of those operations that are defined by the interface of M and the corresponding operations in S can still be shown. The mapping prescribed by the `mapped_to` clauses can be safely used because the consistency and completeness of the methods in M can still be shown. We call this kind of isomorphism **observational faithfulness**, which is a sufficient result for the sound use of `mapped_to` clauses since clients of a model class can only call model methods that are available in the model-class. Observational faithfulness is important because typically there are many operations in S that cannot be mapped to M. For instance, a theorem prover with a higher-order logic typically supports operations like filter or map, which are not expressible in JML, which is first order.

4 Case study: mapping `JMLObjectSet` to `HOL/Set`

In this section, we demonstrate our technique presented in the previous section by mapping model class `JMLObjectSet` to Isabelle's `HOL/Set` theory. Overall, we considered 17 members of the model class: 2 constructors, 9 query methods, and 6 methods that create new `JMLObjectSet` instances. These were all the methods that remained after the simplification step that we discuss below.

All proofs were carried out in Isabelle. The proof scripts contained a total of ca. 380 lines of code (LOC) without comments and empty lines. Consistency of the mapping was proven in ca. 100, completeness in ca. 120 LOC. Equivalence of the equational theory and the method specifications (see Section 4.5) was proven in ca. 160 LOC. All proofs of the case study and the one presented in Section 6 are available at `pm.ethz.ch/people/darvas/iet`.

Our approach does not yet have tool support. All steps in the case study, including the writing of the proof scripts, were performed manually. In Section 8, as future work, we briefly mention areas where tool support could greatly help.

4.1 Simplifications

Since we were interested in the mapping of `JMLObjectSet` and its methods to an Isabelle theory, we first removed all methods that provided object-oriented features irrelevant for the mapping of the model class. These methods included, for instance, `clone`, `singleton`, `hashCode`, and `toString`. In our opinion, such methods need not be part of model classes if one thinks of them as mathematical structures.

As next step, we removed all implementation details, including non-public methods and specifications, since our approach is merely based on the public specification of model classes. Additionally, we removed public members that only provide syntactic sugar, for instance, the constant field `EMPTY` that represents the empty set `JMLObjectSet()`. As mentioned in Section 3.2, only method specifications that describe normal behavior need to be treated by our approach. Thus, we removed all other method specification cases.

In order to keep our case study comprehensible, we removed ghost fields from the model class together with all specification expressions that referred to them. Ghost fields can be handled by mapping a model class M with n ghost fields to an $n+1$-tuple, where the first component represents the structure for M and the other components represent the state of the ghost fields [27]. Omitting ghost fields allows us to map a model class M directly to some structure S and to avoid cluttering up the proof obligations with projections of tuples.

Obviously, omitting ghost fields had some effects on the semantics of the model class, however, not on its general properties. The two ghost fields of the class are `containsNull` and `elementType`. The former has value true if and only if the represented set contains value `null`. After removing the ghost field, this information has to be queried using `has(null)`.

Note that the method signatures in Figure 2 do not admit value `null` to be element of a `JMLObjectSet`-instance. This can be easily changed by adding the `nullable` modifier to method parameters that correspond to set-elements.

The other ghost field `elementType` is of type `\TYPE`, which is a type introduced by JML and represents the kind of all Java types [12]. The field gives an upper bound on the types of the set-elements, that is, all elements must be subtype of `\TYPE`. By dropping the field, this upper bound is lost and objects of arbitrary types are admitted as set-elements.

In summary, the removal of `containsNull` does not change the properties of the model class, while the removal of `elementType` does change them slightly. However, the main characteristics for which the model class is typically used in specifications remain unchanged.

To focus on the main ideas of this article, we decided not to handle methods that referred to non-primitive types other than `Object` and `JMLObjectSet`. For instance, constructors that take as argument a node of a singly-linked list from which a `JMLObjectSet` is created, or methods that convert `JMLObjectSet`s to other model and non-model types. The handling of these kinds of methods is possible once one has provided mappings for all types that are mentioned in their signatures.

4.2 Division of specifications

We analyzed the specification of `JMLObjectSet` and found that the equational theory and method specifications were highly redundant. We illustrate this redundancy by method `union`. The equation defining the method in the equational theory and its method specification is given in Figure 2. It is easy to see that after proper substitutions, the two specifications express the same property. Thus, we decided to split specifications into two parts: one containing only the equational theory and the other containing only the method specifications. This allowed us to analyze their relation, as discussed in Section 4.5.

We note that it is not always the case that the equational theory of a model class and its method specifications are redundant. For instance, model classes `JMLObjectToObjectRelation` and `JMLValueValuePair` specify the behavior of the class in great majority by method specifications. But one could as well specify a model class mainly or entirely by an equational theory. Thus, in general, faithfulness should be proven using both the equational theory and the method specifications together.

4.3 Specifying the mapping

The next step was to specify the mapping of the model class and its methods. The resulting mapping of the methods that we consider in this article is shown in Figure 2.

The mapping of the different methods of the model class was mostly straightforward. Here, we mention three interesting cases. Method `choose` yields an arbitrary element of the set in case it is not empty. Although Isabelle's set theory has no equivalent operation, the method directly corresponds to Hilbert's ϵ-operator, written as "`SOME` x. $P(x)$" in Isabelle, denoting some x for which $P(x)$ is true, provided one exists [10]. In our case, P simply needs to express set membership of x.

Another interesting case was the mapping of method `remove`. As discussed above, theory `HOL/Set` does not contain a corresponding function, thus the indirect mapping to term *this − (insert elem {})* had to be applied. As noted in Section 3.3, this indirect mapping leads to non-trivial backward mappings, but we can still prove that the specification of method `remove` is consistent and corresponds to set difference with a singleton set containing `elem`.

An important issue of the mapping is the handling of equality. In general, we use reference equality for objects [7]. However, instances of model classes are treated as terms of a mathematical structure; therefore, the equality of this structure applies. We achieve this by overloading Isabelle's `=` operator. Instances of non-model classes are represented in Isabelle by a designated sort. The `=` operator on this sort denotes reference equality. Consequently, we simply map Java's `==` operator to Isabelle's `=` operator when applied to instances of non-model classes, in particular, to the elements stored in a `JMLObjectSet`. Instances of model classes are represented in Isabelle by the sort specified in the `mapped_to` clause of the model class. When applied to instances of model classes, we replace the `==` operator by a call to `equals`. This call is then mapped to Isabelle as prescribed by the `mapped_to` clause for `equals`. For instance, the `==` operator on `JMLObjectSet` instances is mapped to set equality in Isabelle. During the isomorphism proof, one has to prove that the `equals` method corresponds to Isabelle's notion of set equality.

4.4 Consistency

In the next step, we proved that the specification of the model class is implied by the properties of Isabelle's `HOL/Set` theory. The proof was performed as described in Section 3.2.

We found one inconsistent equation in the equational theory. This equation intended to describe a relation between methods `remove` and `insert` as follows:

```
s.insert(e1).remove(e2).
  equals(e1 == e2 ? s : s.remove(e2).insert(e1))
```

where `s` is a `JMLObjectSet` instance, and `e1` and `e2` are two objects. The specification expresses that if `e1` and `e2` refer to the same object then inserting and removing the object into and from set `s` yields a set equal to `s`; otherwise, the order of performing the two operations is interchangeable.

Although this might look correct at first sight, the attempt to formally prove its correctness reveals that it is incorrect in case `s` contains `e2`, and `e1` and `e2` refer to the same object. In this case, the insertion yields some set s' that contains the same objects as `s` and the remove operation yields some set s'' that contains the same objects as s' except the object referenced by `e2` (and `e1`). Thus, this set cannot be equal to `s`.

This problem was directly pointed out by Isabelle via the open goal that remained after applying the automatic tactic **auto** on the corresponding lemma. The open goal was: $e2 : s \Rightarrow False$, expressing that the property does not hold in case `s` contains `e2`.

The buggy equation could be easily fixed after the problem was caught and all specifications of the equational theory and the method specifications could be proven trivially using the **auto** tactic of Isabelle. As a consequence, we proved that the (fixed) specification of the model class is consistent.

In general, the axiomatic JML specifications seem to be more error-prone than the conservative Isabelle specifications. Therefore, we expect other model classes to contain similar bugs, which our technique can reveal.

4.5 Equivalence of equational theory and method specifications

While it was easy to notice the large overlap of properties specified by the invariant and by method specifications, it was not trivial to see whether they are equivalent. Thus, after having proved that the specifications are consistent, we proved their equivalence formally using Isabelle.

The procedure of proving the equivalence was the following. First, we declared signatures of function symbols the same way as described in Section 3.3. When proving that the equational theory implies the method specifications, we stated axioms based on the equational theory and generated lemmas based on the method specifications. Finally, we attempted to prove the lemmas using the axioms. The other direction was proved analogously.

We found that the equational theory and the method specifications were not equivalent, and none of them contained stronger specifications than the other. That is, while proving either direction, some lemmas could not be proven without strengthening some of the axioms or adding new ones. Four additional equations had to be added to the equational theory and one postcondition had to be strengthened in the method specifications in order to prove their equivalence. Here we give one example for each direction.

The equational theory contains two specifications that mention method `isEmpty`:

```
new JMLObjectSet().isEmpty()   and
!s.insert(e1).isEmpty()
```

These express that a newly-allocated set is empty and that a set into which an element is inserted is not empty. These specifications do not imply the property stated in the postcondition of method `isEmpty`:

```
\result == (\forall Object e; ; !this.has(e))
```

That is, `isEmpty` returns true if and only if the set does not contain any object. The postcondition could not be proven using the two equations because those just express **properties** of `isEmpty` (after construction and insertion) while the postcondition gives the **definition** of `isEmpty`. Adding this definition to the equational theory (and thus to the set of axioms used in the proofs) trivially solved the problem. In fact, the two original specifications could as well be removed (or marked as redundant) since the new one (together with other properties) implies them.

The specification of constructor `JMLObjectSet(e)` had to be strengthened because the original postcondition **`this`**`.has(e)` was not sufficient to prove two specifications from the equational theory, for instance, the equation that relates the two constructors of the class:

```
new JMLObjectSet(e1).
  equals(new JMLObjectSet().insert(e1))
```

The weakness of the constructor's postcondition was again revealed by the open goal while proving the above equation and suggested us to strengthen the postcondition to express that object `e` is the one and only object contained by the set after construction:

```
(\forall Object e1; ; this.has(e1) <==> (e == e1))
```

The strengthened postcondition allowed us to prove the two remaining specifications in the equational theory.

To make sure that the added and strengthened specifications did not introduce unsoundness, we proved their consistency the same way as in Section 4.4.

The result of having proved the equivalence of the equational theory and the method specifications is that one can use one or the other. For instance, one only needs to prove isomorphism of the method specifications and theory `HOL/Set`, while the equational theory can be marked as redundant.

4.6 Completeness

As the last step, we proved that the definitions of Isabelle's `HOL/Set` theory are implied by the (corrected and strengthened) specification of `JMLObjectSet`. The proof was performed both for the equational theory and for method specifications, and was carried out as described in Section 3.3. We note that due to the equivalence proof sketched above, it would have sufficed to perform this step either for the equational theory or for the method specifications. We carried out the proofs for both of them in order to gain more experience with our approach.

The most interesting part in this step was the mapping of Isabelle definitions to the signatures of the model class. Specifically, many of the definitions in Isabelle's `HOL/Set` theory use set comprehension (see Figure 3). This is a construct that cannot be expressed by a method in the model class. However, probably for this reason, JML supports set comprehension on the syntax level [12]. The JML Reference Manual [12] does not give a concrete definition for the semantics of the construct, thus we used the meaning that Isabelle defines via the axioms. This (1) ensured that we did not introduce unsoundness (provided that Isabelle's definition is sound), and (2) gave a connection between mathematical set comprehension and the methods of `JMLObjectSet` since the Isabelle definition refers to set membership which corresponds to the `has` method of the model class.

With the help of set comprehension, most Isabelle definitions could be easily mapped back to the "language" of the model class. The corresponding lemmas could be proven both by the corrected and strengthened equational theory and by the strengthened method specifications. This means that `JMLObjectSet`'s specification indeed captures the elementary properties of sets.

Most proofs were trivially discharged by giving hints to Isabelle's **auto** and **simp** tactics which axioms to use. When giving such hints, a first approximation is typically to include all axioms that refer to the function symbols that appear in a given proof obligation. However, this might make the tactics loop, requiring one to remove some of the axioms. To decide which ones to remove, one needs to analyze the proof obligation to see which properties may not be needed to complete the proof.

4.7 Mismatching methods and functions

Finally, we mention cases where the model class and the Isabelle structure cannot be related to each other.

As one can see in Figure 2, there is no `mapped_to` clause attached to method `int_size`, which yields the number of elements the set contains. The method cannot be mapped to any term in the target theory since the theory does not define set cardinality and it cannot be expressed by the combination of other functions either. As discussed in Section 3.5, this means that our approach can neither guarantee consistency of specifications that mention the method nor that the semantic meaning of the method is indeed set cardinality. Thus, if the author of the model class wanted to keep the method in the model class,

then the class would need to be mapped to another structure, for instance, Isabelle's `HOL/Finite_Set`, which provides the corresponding function, *card*.

Not surprisingly, due to the higher-order nature of the theory to which the class was mapped, there were definitions that could not be mapped back to the model class. An example is function *image* which takes a function f and a set A as parameters, and yields the image of A under f. The model class does not provide such functionality and it cannot be expressed by other methods of the class. As discussed in Section 3.5, this means that at most observational faithfulness can be shown between the model class and the structure, which is sufficient for the use of the mapping during verification.

5 Handling inductive structures

Many theorem provers allow one to introduce data types and sets inductively. For instance, data type *list* is typically introduced inductively by two type constructors: `Nil`, which creates the empty list; and `Cons` e ls, which creates a list by adding element e to list ls. The set of natural numbers *nat* is defined inductively by two introduction rules: 0 belongs to the set; and, if n belongs to the set then $Suc\ n$ belongs to the set, too.

Inductive definitions are natural to write and convenient to use because behind the scenes, theorem provers automatically generate proper definitions (for instance, a least fixed-point definition for *nat*) which ensure that the introduction of the data type is a conservative extension [28]. Furthermore, numerous theorems are derived from the definitions (for instance, the induction principle) available for the user and for the tactics [14, 28, 29].

As explained in Section 3, proving completeness requires deriving **all** definitions and axioms of the mathematical structure from the specification of the model class. The set of definitions and axioms should also include those that are implicitly generated for inductive structures by the theorem prover. However, the correctness of the implicit definitions and axioms follows from meta-theoretical results [28, 29] and are typically not derivable from the JML specification of the corresponding model class.

For instance, in Isabelle new functions are introduced and axiomatized for every inductive data type. These functions and axioms allow the automatic derivation of essential theorems, for instance, the induction principle. However, these implicit axioms are impossible to derive from the JML specifications, since the functions that Isabelle implicitly introduces do not even exist in the JML model classes.

Furthermore, even the most important implicitly derived Isabelle theorems are not deducible from the JML specification. Consider the induction principle for data type list, which states for any property P and list ys:

$$\frac{P\ \texttt{Nil} \qquad \forall x, xs.\ P\ xs \Rightarrow P\ (\texttt{Cons}\ x\ xs)}{P\ ys}$$

This principle cannot be expressed in JML, which is based on first order logic and, thus, does not allow one to quantify over predicate P. Note that this problem did not occur for model class `JMLObjectSet` because type $\alpha\ set$ to which it was mapped, is not defined inductively.

The above mentioned problems indicate that the technique presented in Section 3.3 to show completeness is not sufficient to handle inductively defined types because the proof would always fail for the generated definitions and axioms. In this section, we address this problem by using four characteristic properties of inductive structures, which imply the other implicit definitions and axioms, and by showing that these properties also hold for the model class. Three of the properties can typically be proved based on the model class specification. The fourth can be enforced by a simple syntactic check. In the rest of this section, we focus on inductive data types and omit inductive sets, which can be handled in a similar manner.

5.1 Characteristic properties

In this subsection, we show how our approach enforces that model classes that are mapped to inductive data types indeed have the properties that follow from the inductive nature of the data type. We only cover non-nested, non-mutually-recursive definitions, which suffice to cover all model classes of the JML model library. The general form of a non-nested, non-mutually-recursive inductive data type definition is [28]:

$$(\alpha_1, \ldots, \alpha_n)rty \ ::= \ \mathtt{C}_1 \ ty_1^1 \ \ldots \ ty_1^{k_1} \mid \ldots \mid \mathtt{C}_m \ ty_m^1 \ \ldots \ ty_m^{k_m} \tag{2}$$

which defines type $(\alpha_1, \ldots, \alpha_n)rty$ with n type variables $\alpha_1, \ldots, \alpha_n$ where $n \geq 0$. The type has m constructors $\mathtt{C}_1$, ..., $\mathtt{C}_m$ where $m \geq 1$. Each constructor $\mathtt{C}_i$ takes k_i arguments where $k_i \geq 0$, and type expression ty_i^j for $1 \leq j \leq k_i$ either does not contain rty (i.e., is non-nested) or is equal to $(\alpha_1, \ldots, \alpha_n)rty$ (i.e., is recursive).

For example, the formal definition of type *list* is typically given as $(\alpha)list \ ::= \ \mathtt{Nil} \mid \mathtt{Cons}\ \alpha \ (\alpha)list$, where `Nil` takes no argument, the first argument of `Cons` is non-nested, and the second is recursive.

Inductive data types defined in the form of (2) can be fully characterized by three properties: the injectivity and distinctness of constructors, and the induction principle [21, 30]. That is, all other standard properties of inductive data types can be derived from these three properties. While we know that the three characteristic properties hold for the inductively defined data type, we have to show that the same properties hold for the corresponding model class too.

While the first two properties can be expressed in JML, the induction principle, as noted above, cannot. Thus, we capture other properties of data types that ensure that the set of possibly constructable values of the data type is the least set that can be finitely generated by the constructors [28], and thus, imply the validity of the induction principle. These properties are inclusiveness, no junk, and no loops.

Lastly, we will be interested in the property of non-emptiness. Although non-emptiness is not a necessary requirement in every logic, our approach enforces the property in order to make it independent of the logic.

In the following, we define formally the meaning of the above mentioned properties. These are the properties that we will require from model classes that are mapped to inductively defined data types.

1. injectivity of constructors. For every constructor $\mathtt{C}_i$ and their parameters, the following holds:

$$\mathtt{C}_i \ x_i^1 \ \ldots \ x_i^{k_i} \ = \ \mathtt{C}_i \ y_i^1 \ \ldots \ y_i^{k_i} \ \Leftrightarrow \ x_i^1 = y_i^1 \ \wedge \ \ldots \ \wedge \ x_i^{k_i} = y_i^{k_i} \tag{3}$$

2. distinctness of constructors. For every pair of constructors $\mathtt{C}_i$ and $\mathtt{C}_j$, where $i \neq j$, and for all parameters of $\mathtt{C}_i$ and $\mathtt{C}_j$, the following holds:

$$\mathtt{C}_i \ x_i^1 \ \ldots \ x_i^{k_i} \ \neq \ \mathtt{C}_j \ y_j^1 \ \ldots \ y_j^{k_j} \tag{4}$$

3. inclusiveness. Every value of the type S is denoted by some term of the form $\mathtt{C}_i \ x_i^1 \ \ldots \ x_i^{k_i}$:

$$\begin{array}{c} \forall\, v : S.\ (\exists\ x_1^1 \ \ldots \ x_1^{k_1}.\ v = \mathtt{C}_1 \ x_1^1 \ \ldots \ x_1^{k_1}\ ) \ \vee \ \ldots \ \vee \\ (\exists\ x_m^1 \ \ldots \ x_m^{k_m}.\ v = \mathtt{C}_m \ x_m^1 \ \ldots \ x_m^{k_m}\ ) \end{array} \tag{5}$$

4. no loops. For any non-empty sequence of nested constructors and value x, the following holds:

$$x \ \neq \ \mathtt{C}_1(\ldots, \mathtt{C}_j(\ldots, x, \ldots), \ldots) \tag{6}$$

5. no junk. The set of data-type values is the minimal set closed under the constructors. That is, every value of the type can be built by repeated applications of the constructors.

6. non-emptiness. The type has at least one element.

These six characteristic properties have to be enforced on the model class, that is, we need to show that (1) all model instances can be related to values that can be constructed by type constructors $\mathtt{C}_i$, and that (2) the characteristic properties hold for the model instances.

5.2 Enforcing characteristic properties

Note that the characteristic properties are expressed in the language of the structure; thus, the backward mapping from the structure to the model class needs to be applied. As noted in Section 3.3, deriving this backward mapping may be non-trivial. For the proof of the characteristic properties, we require that at least every constructor of the inductive data type can be mapped back. Otherwise, the proof fails. On the other hand, as before, if multiple backward mappings exist then all of them have to be taken into account.

We demonstrate how the characteristic properties are proven using model class `JMLObjectSequence`, which is mapped to data type $(\alpha)list$ as specified in Figure 5. Given the specified `mapped_to` clauses, we get the following backward mappings: `Nil` is trivially mapped back to $\hat{JMLObjectSequence}()$; `Cons` e ls is trivially mapped back to $\hat{insertFront}(ls, e)$, and to $\hat{JMLObjectSequence}(e)$ with ls matched with `Nil` and, consequently replaced by $\hat{JMLObjectSequence}()$.

Note that the backward mapping of data type constructors may not only yield model constructors, but also model methods. This is important because the implementation of a model class often has constructors that are not part of its public interface and which create instances that are not composable by the public constructors of the class. Such non-public constructors are typically "hidden" behind public model methods, that is, the implementations of the methods call these non-public constructors. An example is method `insertFront`, which creates instances that cannot be constructed by the two public constructors: `insertFront` creates lists with at least one element, whereas the constructors can only create empty lists and lists with exactly one element.

Based on the backward mapping, we can express the characteristic properties in the language of the model class. As we will see, some of the properties can be enforced syntactically and some of them follow from the other properties in our special setting of model classes.

1. injectivity of constructors. We have to show that equivalence (3) carries over to the model class. Thus, we generate proof obligations that correspond to this equivalence by substituting the type constructors using the backward mapping.

As an example, we show how the substitution works when applied to type constructor `Cons` and symbol $\hat{insertFront}$. The signatures are:

$$\begin{aligned} &\texttt{Cons}: \ \alpha \ \times \ \alpha \ list \ \Rightarrow \alpha \ list \\ &\hat{insertFront}: \ \alpha \ list \ \times \ \alpha \ \Rightarrow \alpha \ list \end{aligned}$$

From the `mapped_to` clause of method `insertFront`, we know that during the substitution of `Cons` by $\hat{insertFront}$, we have to swap the positions of the parameters. This yields the following proof obligation:

$$\forall\, s_1, s_2, e_1, e_2. \hat{equals}(\hat{insertFront}(s_1, e_1), \hat{insertFront}(s_2, e_2)) \ \Leftrightarrow e_1 = e_2 \ \wedge \ \hat{equals}(s_1, s_2)$$

Note that when the operator = is applied to the data type, it is replaced by an application of $\hat{equals}$. This is a consequence of the specified mapping of method `equals` to operator = in the model class. In Section 6, we show an example where the substitution of parameters is not as trivial as in the example above.

Since multiple methods of the model class may be mapped to a type constructor $\texttt{C}_i$, $\texttt{C}_i$ may be substituted by multiple function symbols. To ensure that the property of injectivity carries over to the model class, we have to consider all possible substitutions of $\texttt{C}_i$ in equivalence (3). Moreover, we have to consider all possible combinations of substitutions since $\texttt{C}_i$ appears on both sides of the equivalence. In our example, this leads to four proof obligations since `Cons` can be mapped back in two different ways. One of the proof obligations is the formula above. All four are shown in Figure 6.

2. distinctness of constructors. To show that distinctness of constructors carries over to the model class, we have to perform substitutions on formula (4) in the same way as for injectivity. For instance, the distinctness property `Nil` $\neq$ `Cons` e s yields the following formula when the substitution is applied with $\hat{JMLObjectSequence}()$ on `Nil` and with $\hat{insertFront}$ on `Cons`:

$$\forall\, s, e. \ \neg \ \hat{equals}(\hat{JMLObjectSequence}(), \hat{insertFront}(s, e))$$

The substitution of `Nil` is trivial since neither `Nil` nor the corresponding symbol $\hat{JMLObjectSequence}()$ take any parameter. The substitution of `Cons` works as described above.

Analogously to injectivity, multiple proof obligations have to be proved if multiple model methods are mapped to the same type constructor. In this example, the second backward mapping of `Cons` yields a second proof obligation, see Figure 6.

3. inclusiveness. To enforce inclusiveness for the model class, we need to show that every model instance has a mapping to the structure. This is expressed by translating equivalence (5) to the model class:

$$\forall v : M.\ (\exists\ \overrightarrow{x_1}.\ \hat{equals}(v,\ \hat{g_1}(\overrightarrow{x_1}))\ )\ \vee\ \ldots\ \vee \\ (\exists\ \overrightarrow{x_k}.\ \hat{equals}(v,\ \hat{g_k}(\overrightarrow{x_k}))\ )$$

where the $\hat{g_i}(\overrightarrow{x_i})$ are the results of the backward mapping applied to the type constructors of S.

The quantification over all instances v of class M makes this condition difficult to prove. However, since model classes are immutable, we can prove it indirectly. If the property holds for each method of M that returns an instance of type M then it holds for all instances of M. The above proof obligation is adapted as follows when proven for some method m: the quantification over v is replaced by quantification over the parameters of m, and variable v is replaced by an application of function $\hat{m}$. For instance, the proof obligation for method `concat` is the following:

$$\begin{array}{l}\forall\, this,\ s2. \\ \quad \hat{equals}(\hat{concat}(this, s2),\ \hat{JMLObjectSequence}()) \vee \\ \quad \exists\, e.\ \hat{equals}(\hat{concat}(this, s2),\ \hat{JMLObjectSequence}(e)) \vee \\ \quad \exists\, s,\ e.\ \hat{equals}(\hat{concat}(this, s2),\ \hat{insertFront}(s, e)))\end{array}$$

4. no loops. Note that the expression on the right-hand side of (6) is an arbitrary sequence of constructor applications. Thus, proving (6) directly would lead to an infinite set of proof obligations. Therefore, we prove the property as follows.

We identify some measure μ in the data type S for which we can prove the following two properties. First, any two values with different measures are non-equal:

$$\forall\, s_1,\ s_2 : S.\ \mu(s_1) \neq \mu(s_2)\ \Rightarrow\ s_1 \neq s_2 \tag{7}$$

Second, the measure increases by every application of constructors that have recursive arguments. Formally, for some constructor $\mathtt{C}_i$ with one recursive argument (assumed to be on the first position):

$$\forall\, \overrightarrow{x},\ s : S.\ \mu(s) < \mu(\mathtt{C}_i(s, \overrightarrow{x})) \tag{8}$$

In data type $(\alpha)list$ (and in any other inductively defined data type in Isabelle), `size` is a candidate for the measure. One can show that it is indeed a measure by trivially proving the following two lemmas:

$$\forall\, s_1,\ s_2.\ (\texttt{size}\ s_1) \neq (\texttt{size}\ s_2)\ \Rightarrow\ s_1 \neq s_2$$

$$\forall\, e,\ s.\ (\texttt{size}\ s) < (\texttt{size}\ (\texttt{Cons}\ e\ s))$$

Note that property (8) only has to be proven for constructor `Cons`, as `Nil` is not recursive.

By a simple inductive argument on the number of constructor applications on the right-hand side of (6), one can trivially show that these two properties together imply the absence of loops.

In order to show that properties (7) and (8) hold for the model class as well, they have to be mapped back to the language of the model class. We give the precise proof obligations that have to be proven for the model class in the next section.

5. no junk. As the property of inclusiveness guarantees us that every data-type value can be related to a JML instance, and that every JML instance is constructable by a member that is mapped to a data-type constructor, we can deduce that the JML model class does not contain junk either.

Since this argument is valid for any model class that is mapped to an inductively defined data type, no proof obligation has to be proven.

6. non-emptiness. We have to show that there is at least one model instance that can be created by the model class and that has a counterpart in the structure. Since the property of inclusiveness ensures that every model instance has a counterpart in the structure, it suffices to show that at least one model instance can be created.

For a model class M, the existence of an instance can be shown by syntactically checking that at least one of the model constructors that are mapped to a type constructor does not contain parameters of type M or of any other model class. We exclude other model classes since there is potential mutual recursion between M and that class. Since methods of non-model classes cannot have parameters whose type is a model class, we do not have to exclude those.

For instance, in Figure 5, we can see that method `insertFront` has an explicit parameter of the enclosing type `JMLObjectSequence`; thus, it does not guarantee non-emptiness. However, constructor `JMLObjectSequence()` does not have any parameters; thus, there is at least one instance of model class `JMLObjectSequence`.

The result of having proven these six characteristic properties is that we may safely assume that the model class possesses the standard properties of inductively defined types. In particular, we may assume that the induction principle holds for the model class, which therefore can be added as an axiom to the set of axioms that is extracted from the specification of the model class during the completeness proof (see step 2. in Section 3.3).

6 Case study: mapping `JMLObjectSequence` to `HOL/List`

In this section, we present our second case study: the mapping of model class `JMLObjectSequence` to Isabelle's `HOL/List` theory [31], more specifically, to the inductive data type $(\alpha)list$. We only present the proof obligations that need to be proven to show that the class possesses the characteristic properties of the inductive data type $(\alpha)list$, but we do not discuss how consistency and the other properties of completeness were proven. The procedure for those proofs (that is, simplifications as well as generation and proving of proof obligations) is analogous to the previous case study.

The mapping of the class and those methods that we consider in this article is presented in Figure 5.

1. injectivity of constructors. Injectivity of `Nil` is trivial since the type constructor does not take any parameters. This is also the case with the corresponding model member `JMLObjectSequence`().

The injectivity property for `Cons` is that for all lists s_1, s_2 and elements e_1, e_2, the following holds:

$$\texttt{Cons}\ e_1\ s_1 = \texttt{Cons}\ e_2\ s_2 \ \Leftrightarrow\ e_1 = e_2\ \wedge\ s_1 = s_2$$

To show that the property holds for the model class, four proof obligations need to be proven, since there are two possible backward mappings for `Cons`. In Section 5.2, we described how `Cons` is substituted by symbol $\hat{insertFront}$. Here, we explain how `Cons` is substituted by $\hat{JMLObjectSequence}(e)$. Their signatures are as follows:

$$\begin{aligned}&\texttt{Cons}: \ \alpha\ \times\ \alpha\ list\ \Rightarrow \alpha\ list\\ &\hat{JMLObjectSequence}(e): \ \alpha\ \Rightarrow \alpha\ list\end{aligned}$$

As we can see, there is a mismatch between their arguments: `Cons` has an additional $\alpha\ list$ argument. From the `mapped_to` clause of constructor `JMLObjectSequence(e)`, which specifies `Cons e Nil`, we can derive that this $\alpha\ list$ argument is `Nil`. Since we know that the backward mapping for `Nil` yields $\hat{JMLObjectSequence}()$, we can conclude that when `Cons` is to be substituted by $\hat{JMLObjectSequence}(e)$, the first argument of `Cons` gives the only argument of $\hat{JMLObjectSequence}(e)$, while the second argument of `Cons` is to be replaced by the constant $\hat{JMLObjectSequence}()$. Applying the substitutions for the property in all combinations yields the upper four proof obligations in Figure 6.

The proof obligations have to be proven using the axioms that are extracted from the specification of the model class as described in Section 3.3. We proved the first three proof obligations by giving hints to Isabelle's **auto** tactic which axioms to use. The fourth proof obligation additionally required a quantifier instantiation and a case split.

```
package org.jmlspecs.models;
//@ immutable
//@ mapped_to("Isabelle", "HOL/List", "α list");
public /*@ pure @*/ class JMLObjectSequence {
  /*@ public invariant
    @   (\forall JMLObjectSequence s2; s2 != null;
    @     (\forall Object e1, e2; ;
    @       equational_theory(this, s2, e1, e2))); */
  /*@ public normal_behavior
    @   ensures \result <==>
    @     s.insertFront(e1).concat(s2).equals(
    @        s.concat(s2).insertFront(e1));
    @ static public pure model boolean
    @   equational_theory(JMLObjectSequence s,
    @     JMLObjectSequence s2, Object e1, Object e2); */

  //@ mapped_to("Isabelle", "Nil");
  public JMLObjectSequence();

  //@ mapped_to("Isabelle", "Cons e Nil");
  public JMLObjectSequence(Object e);

  //@ mapped_to("Isabelle", "size this");
  public int int_size();

  //@ mapped_to("Isabelle", "elem mem this");
  public boolean has(Object elem);

  //@ mapped_to("Isabelle", "this = obj");
  public boolean equals(Object obj);

  /*@ public normal_behavior
    @   ensures \result == (int_size() == 0); */
  //@ mapped_to("Isabelle", "null this");
  public boolean isEmpty();

  //@ mapped_to("Isabelle", "Cons item this");
  public JMLObjectSequence insertFront(Object item);

  //@ mapped_to("Isabelle", "this @ s2");
  public JMLObjectSequence concat(JMLObjectSequence s2);

  //@ mapped_to("Isabelle", "rev this");
  public JMLObjectSequence reverse();
}
```

Figure 5: Model class `JMLObjectSequence` containing the signatures of methods we consider in this article.

$$\forall e_1,\, e_2.\ \hat{equals}(\hat{JMLObjectSequence}(e_1),\ \hat{JMLObjectSequence}(e_2)) \ \Leftrightarrow\ e_1 = e_2 \ \wedge\ \hat{equals}(\hat{JMLObjectSequence}(), \hat{JMLObjectSequence}())$$

$$\forall s,\, e_1,\, e_2.\ \hat{equals}(\hat{JMLObjectSequence}(e_1),\ \hat{insertFront}(s, e_2)) \ \Leftrightarrow\ e_1 = e_2 \ \wedge\ \hat{equals}(\hat{JMLObjectSequence}(), s)$$

$$\forall s,\, e_1,\, e_2.\ \hat{equals}(\hat{insertFront}(s, e_1),\ \hat{JMLObjectSequence}(e_2)) \ \Leftrightarrow\ e_1 = e_2 \ \wedge\ \hat{equals}(s, \hat{JMLObjectSequence}())$$

$$\forall s_1,\, s_2,\, e_1,\, e_2.\ \hat{equals}(\hat{insertFront}(s_1, e_1),\ \hat{insertFront}(s_2, e_2)) \ \Leftrightarrow\ e_1 = e_2 \ \wedge\ \hat{equals}(s_1, s_2)$$

$$\forall e.\ \neg\ \hat{equals}(\hat{JMLObjectSequence}(),\ \hat{JMLObjectSequence}(e))$$

$$\forall s,\, e.\ \neg\ \hat{equals}(\hat{JMLObjectSequence}(),\ \hat{insertFront}(s, e))$$

Figure 6: Proof obligations for injectivity and distinctness for model class `JMLObjectSequence`.

2. distinctness of constructors. Type $(\alpha)list$ has two constructors. Thus, it has only one distinctness property: for every list s and element e, `Nil` $\neq$ `Cons` e s holds. Since the backward mapping yields two model class methods for `Cons` and one for `Nil`, there are two combinations of substitutions for the type constructors. Thus, two proof obligations are to be proven in order to show that `JMLObjectSequence` preserves the property.

We have already seen how to do the substitutions for `Cons`, and that `Nil` is to be substituted by $\hat{JMLObjectSequence}()$. After performing the substitutions of type constructors, we get the two formulas at the bottom in Figure 6. We proved both proof obligations by giving hints to the prover which axioms to use.

3. inclusiveness. As mentioned in Section 5.2, the proof of inclusiveness requires a proof obligation for each method of the model class that yields an instance of the class. We have shown an instance of the proof obligation for method `concat` in Section 5.2.

In our case study, there were 15 methods of class `JMLObjectSequence` that returned a sequence instance. Our experience was that this kind of proof obligation was the most difficult to prove because the specifications of numerous methods, and thus the corresponding axioms interact. For instance, the proof obligation for method `concat` contains function applications of $\hat{JMLObjectSequence}()$, $\hat{JMLObjectSequence}(e)$, $\hat{insertFront}$, $\hat{equals}$, and $\hat{concat}$. Thus, in order to discharge the proof obligation, most of the axioms for these symbols have to be used in a non-trivial way. Furthermore, the existential quantifications typically require manual instantiations.

Often certain subgoals were unprovable from the specification of the model class. In such cases, we extended the specification of the class with additional properties. For instance, while proving inclusiveness for method `concat`, a subgoal could not be proven because commutativity of concatenation and insertion to the front could not be derived from the specification of the model class. Thus, we added the property to the equational theory of the class, as shown in Figure 5.

The proof for method `concat` required several non-trivial manual steps, including two case splits, four quantifier instantiations, and three substitutions.

4. no loops. As mentioned above, a proper measure in data type $(\alpha)list$ is `size`, which trivially maps back to method `int_size`. Thus, the proof obligation instantiated for property (7) is:

$$\forall s_1,\, s_2.\ \hat{int\_size}(s_1) \neq \hat{int\_size}(s_2) \ \Rightarrow\ \neg \hat{equals}(s_1, s_2)$$

Although data type constructor `Cons` maps back both to symbol $\hat{JMLObjectSequence}(e)$ and $\hat{insertFront}$, property (8) only has to be instantiated for the latter, because the former does not correspond to a member that has any parameter of type `JMLObjectSequence`. The resulting proof obligation is:

$$\forall\, e,\, s.\ \hat{int\_size}(s) < \hat{int\_size}(\hat{insertFront}(s, e))$$

Both proof obligations were trivially proven by **auto** with hints on which axioms to use.

5. no junk. As argued above, the property follows from inclusiveness.

6. non-emptiness. There are two constructors in `JMLObjectSequence` that do not contain recursive parameters. Thus, non-emptiness is ensured.

7 Related work

The idea of using function symbols that are understood by the back-end theorem prover directly on the specification level is already present in ESC/Java [32]. The special construct `\dttfsa` (*Damn The Torpedos, Full Speed Ahead!*) allows users to refer to function applications on the level of Simplify, the theorem prover of ESC/Java. The corresponding function symbols are defined directly on the level of the prover. While this construct is a powerful means for specification, one has to be careful with its usage since on the specification level, the definitions of the function are hidden. The verification system does not give support for showing that the definitions are free from inconsistencies.

The Caduceus tool is a static verification system for C programs [33]. For specification and verification purposes, the tool allows one to declare types and predicates as well as to define or axiomatize these predicates on the C source level. One can also define "hybrid" predicates, predicates that refer both to elements of the C program and elements of these specification-only types and predicates. Definitions of predicates can also be postponed on the source level and given directly in Coq, the back-end prover of the tool. This concept eases the task of specifying and verifying programs since, for instance, it prevents the use of method calls in specifications and leads to definitions that are more suitable for provers than JML specifications. Case studies demonstrate the power of this approach [34, 35]. The drawback of the approach is the absence of a consistency proof for definitions and axioms given on the source or prover level. This might lead to soundness issues.

Schoeller [36] roughly sketches the idea of the faithful mapping of model classes to mathematical structures. However, no details are given on how one would prove faithfulness.

Schoeller et al. developed a model library for Eiffel [13, 37]. They address the faithfulness issue by equipping methods of model classes with specifications that directly correspond to axioms and theorems taken from mathematical textbooks. A shortcoming of this approach is that the resulting model library has to follow exactly the structure of the mimicked theory. This limits the design decisions one can make when composing the model library and it is unclear how one can support multiple theorem provers. Furthermore, user-defined model classes cannot be supported since there is no corresponding theory. Our approach allows more flexibility in the construction of model classes and libraries by using `mapped_to` clauses that can go beyond direct mappings since arbitrary terms of the target context can be specified. In turn, our approach requires one to prove faithfulness of the mapping.

Charles [7] proposes the introduction of the `native` keyword to JML in the context of work on the program verifier Jack [38]. The keyword can be attached to methods with a similar meaning to ESC/Java's `\dttfsa` construct: methods marked as `native` introduce uninterpreted function symbols, and their definitions can be directly given on the level of Coq, the back-end prover of Jack. Charles carries the idea over to model classes: the `native` keyword may also be attached to types with the meaning that such types get mapped to corresponding Coq data types. The mapping of `native` types is also defined on the Coq level. This approach differs from ours in two ways. First, our approach ensures faithfulness of the mapping. There is no attempt to do so in the work of Charles. Second, the `mapped_to` clause we propose in this article allows one to specify the mapping on the specification language level. Furthermore, properties of model classes are specified in JML, which typically provides easier understanding (for programmers) of the semantics than definitions given directly on the level of a theorem prover.

Leavens et al. [9] identify the problem of specifying model types as a research challenge. They propose four possible solution approaches and summarize the open problems for each of them. Solution 1 uses

equational theories. One of the problems of this solution is that the resulting invariants quantify over all instances of a model class, which makes modular verification difficult. As we have illustrated in Section 4.5, our technique helps in proving the equivalence of equational theories and method specifications, which are typically easier to verify. The other problem of solution 1 mentioned by Leavens et al. is the possible semantic mismatch between the `equals` method of the model class and the built-in equality of the theorem prover. We solved this problem by mapping the `equals` method explicitly to an operation of the theorem prover and by proving faithfulness of this mapping.

Solution 2 uses method specifications. Leavens et al. observe that this solution makes it difficult to verify the implementations of model methods, a problem that we do not consider in this paper. Moreover, solution 2 requires a sound treatment of pure methods, which is exactly what we provide through proving faithfulness. As solution 3, Leavens et al. discuss automatic translations between model classes and mathematical structures and argue why such translations are difficult. We deal with these problems by specifying the mapping manually and proving faithfulness of the mapping. One of the problems mentioned by Leavens et al. is how to deal with model class specifications that invoke methods of the program, in particular, `Object.equals`. We do not address this problem here; a possible solution might be to also map those methods to operations of the theorem prover and to prove faithfulness. Leavens et al.'s solution 4 is similar to the work by Schoeller and Charles discussed above.

Several approaches have been developed to prove the consistency of design specifications written in languages such as Alloy [39] or OCL [40]. These approaches typically generate a model for the specification, whereas our work shows consistency indirectly by proving isomorphism to a structure that is assumed to be consistent. Our approach allows us in particular to replace model class specifications by terms of the theorem prover during program verification, which is one of the main motivations behind our work.

Our approach is based on the equivalence proof of two structures. Thus, it can be seen as stating that the structures are in a bi-directional refinement relation. This can be expressed in existing specification languages. For instance, in ESC/Java by the combination of the above mentioned `\dttfsa` construct together with JML's **`refining`** clause, which takes two arguments and specifies that one refines the other. An advantage of the newly proposed `mapped_to` clause is that it allows one to additionally specify the target theorem prover, possibly multiple ones.

There is a large body of work on the **runtime** discovery and checking of algebraic specifications. We give an incomplete list of references and refer the reader to [41] for a more detailed account on previous work. Sankar [42], Antoy and Hamlet [43], and Nunes et al. [44] developed systems that allow one to test the implementation of algebraic data types against their implementations. Henkel and Diwan [41, 45] developed an approach for the discovery of algebraic specifications as well as for their debugging. These approaches are orthogonal to ours. First, they inspect the relation between the specification and implementation of data types. Our approach inspects the relation between the specifications and definitions of two structures. Second, the listed approaches are dynamic, that is, based on executions of the inspected data types. Thus, these approaches cannot guarantee that the result of the specification discovery and checking is sound. Our approach is based on static verification techniques.

8 Conclusion

For the static verification of programs, model classes have to be encoded in the underlying logic of the program verifier. Previous work [7, 8, 9] proposed the mapping of model classes to mathematical structures of the underlying theorem prover. In this article, we proposed an approach to show that such mappings are faithful by proving isomorphism between the model classes and the structures.

The proposed approach improves on previous work in three ways. First, previous work that proposed the direct translation of model-class methods to functions of a theorem prover did not ensure any actual semantic relationship between the mapped entities. This can easily lead to semantic mismatch between what was intended to be specified and what was actually verified.

Second, our approach leads to better specifications for model classes by ensuring their (relative) consistency and completeness. The identification and checking of redundant specifications further improves the quality of the specifications.

Third, previous work for ensuring the consistency of recursive pure-method specifications either does not provide a satisfying solution [3], or proposes to explicitly prove well-foundedness [6]. The solution

proposed by this article solves this problem: by proving that a certain mathematical structure is a model for the specifications of a model class, we get the guarantee that the specifications are consistent. This result is independent of the presence of recursion and requires no explicit proof for well-foundedness.

To demonstrate our approach, we reported on case studies with two model classes from JML's model library and two theories from Isabelle's theory library. The case studies were successful in that observational faithfulness could be proved (except for method `int_size` of class `JMLObjectSet`) and interesting observations could be made on the model classes: an incorrect specification was revealed, missing specifications were identified, and a precise relation between the equational theory and method specifications was identified. Additionally, we did a case study, which mapped a model class encoding a stack as described in [16] to theory `HOL/List`, and proved observational faithfulness. Since stack is a rather simple structure, the case study was rather simple to complete. Still, it shows that our approach also works for model classes that do not have a directly corresponding theory in the target theorem prover.

Future work. Future work remains to provide tool support for the proposed mapping process described in this article. Tools could support the typechecking of `mapped_to` clauses; the (partial) generation of proof scripts for faithfulness proofs; and the actual use of mappings for static verification of programs.

We plan to carry out further case studies to cover most of the model classes of the JML model library. This will also give us a better understanding of the strengths and weaknesses of our approach.

Acknowledgments. We are grateful to David Basin, Stefan Berghofer, Nikolaj Bjørner, Farhad Mehta, Tobias Nipkow, and Burkhart Wolff for helpful discussions, and to the anonymous reviewers of IET Software and the SAVCBS 2007 Workshop for their insightful comments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Müller's work was partly done at Microsoft Research, Redmond.

References

[1] Guttag, J.V., and Horning, J.J.: 'Larch: Languages and Tools for Formal Specification' (Texts and Monographs in Computer Science. New York, NY, USA, Springer-Verlag, 1993)

[2] Cheon, Y., Leavens, G.T., Sitaraman, M., and Edwards, S.: 'Model variables: cleanly supporting abstraction in design by contract', Software: Practice and Experience, 2005, 35, (6), pp. 583–599

[3] Darvas, Á., and Müller, P.: 'Reasoning About Method Calls in Interface Specifications', Journal of Object Technology, 2006, 5, (5), pp. 59–85

[4] Jacobs, B., and Piessens, F.: 'Verification of Programs with Inspector Methods', Int. Workshop Formal Techniques for Java-like Programs, Nantes, France, 2006

[5] Darvas, Á., and Leino, K.R.M.: 'Practical reasoning about invocations and implementations of pure methods', in Dwyer, M.B., and Lopes, A. (Eds.): Proc. Int. Conf. Fundamental Approaches to Software Engineering. Vol. 4422 of LNCS (Springer-Verlag). Braga, Portugal, 2007, pp. 336–351

[6] Rudich, A., Darvas, Á., and Müller, P.: 'Checking well-formedness of pure-method specifications', in Cuellar, J., and Maibaum, T. (Eds.): Proc. Int. Symp. Formal Methods. Vol. 5014 of LNCS (Springer-Verlag). Turku, Finland, 2008, pp. 68–83

[7] Charles, J.: 'Adding Native Specifications to JML', Int. Workshop Formal Techniques for Java-like Programs. Nantes, France, 2006

[8] Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., and Cok, D.R.: 'How the design of JML accommodates both runtime assertion checking and formal verification', Science of Computer Programming, 2005, 55, (1–3), pp. 185–205

[9] Leavens, G.T., Leino, K.R.M., and Müller, P.: 'Specification and verification challenges for sequential object-oriented programs', Formal Aspects of Computing, 2007, 19, (2), pp. 159–189

[10] Nipkow, T., Paulson, L.C., and Wenzel, M.: 'Isabelle/HOL — A Proof Assistant for Higher-Order Logic' (Vol. 2283 of LNCS, Springer-Verlag, London, UK, 2002)

[11] Shankar, N., Owre, S., and Rushby, J.M.: 'A Tutorial on Specification and Verification Using PVS (Beta Release)', Technical Report, Computer Science Laboratory, SRI International, 1993

[12] Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D. et al.: 'JML Reference Manual', Last revised October 2007, Technical Report, Department of Computer Science, Iowa State University

[13] Schoeller, B., Widmer, T., and Meyer, B.: 'Making specifications complete through models', in Reussner, R.H., Stafford, J.A., Szyperski, C.A. (Eds.): Int. Seminar Architecting Systems with Trustworthy Components. Vol. 3938 of LNCS (Springer-Verlag). Dagstuhl, Germany, 2006, pp. 48–70

[14] Bertot, Y., and Castéran, P.: 'Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions', (Texts in Theoretical Computer Science. Springer-Verlag, 2004)

[15] Darvas, Á., and Müller, P.: 'Faithful mapping of model classes to mathematical structures', Int. Workshop Specification and Verification of Component-Based Systems, Dubrovnik, Croatia, 2007, pp. 31–38

[16] Meyer, B.: 'Object-Oriented Software Construction' (Prentice Hall, Upper Saddle River, NJ, USA, 1988, 2nd edn. 1997)

[17] Müller, P., Poetzsch-Heffter, A., and Leavens, G.T.: 'Modular Invariants for Layered Object Structures', Science of Computer Programming, 2006, 62, pp. 253–286

[18] Hoare, C.A.R.: 'Proof of Correctness of Data Representations', Acta Informatica, 1972, 1, (4), pp. 271–281

[19] JML Distribution. `http://sourceforge.net/projects/jmlspecs`, accessed May 2008. Model library available in package `org.jmlspecs.models`

[20] Paulson, L.C.: 'ML for the working programmer', (Cambridge University Press, New York, NY, USA, 1991, 2nd edn. 1996)

[21] Nipkow, T., Paulson, L.C., and Wenzel, M.: 'Isabelle's Logics: HOL', 2007. Manual available at `http://isabelle.in.tum.de/doc/logics-HOL.pdf`, accessed May 2008

[22] Gordon, M., Milner, A., and Wadsworth, C.: 'Edinburgh LCF: A Mechanized Logic of Computation', (Vol. 78 of LNCS, Springer-Verlag, 1979)

[23] Shoenfield, J.R.: 'Mathematical Logic', (Addison-Wesley, Reading, MA, USA, 1967)

[24] Nipkow, T., Paulson, L.C., and Wenzel, M.: 'Theory HOL/Set from "The Isabelle Library"', 2005. Available at `isabelle.in.tum.de/library/HOL/Set.html`, accessed May 2008

[25] Owre, S., and Shankar, N.: 'The PVS Prelude Library', Technical Report SRI-CSL-03-01, Computer Science Laboratory, SRI International, 2003

[26] Coq theory of Classical_sets. Available at `http://pauillac.inria.fr/coq/V8.1/stdlib/Coq.Sets.Classical_sets.html`, accessed May 2008

[27] Miragliotta, M.: 'Specification Model Library for the Interactive Program Prover JIVE', Semester Thesis, ETH Zurich, 2004

[28] Melham, T.F.: 'Automating recursive type definitions in higher order logic', in Birtwistle, G., Subrahmanyam, P.A. (Eds.): 'Current trends in hardware verification and automated theorem proving' (Springer-Verlag, New York, NY, USA, 1989), pp. 341–386

[29] Berghofer, S., and Wenzel, M.: 'Inductive Datatypes in HOL – Lessons Learned in Formal-Logic Engineering', Proc. Int. Conf. Theorem Proving in Higher Order Logics, (Springer-Verlag, London, UK), 1999, pp. 19–36

[30] Berghofer, S.: 'Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL', Master's thesis (In German), Institut für Informatik, Technische Universität München, 1998

[31] Nipkow, T.: 'Theory HOL/List from "The Isabelle Library"', 2005. Available at `isabelle.in.tum.de/library/HOL/List.html`, accessed May 2008

[32] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R.: 'Extended static checking for Java', Proc. Int. Conf. Programming Language Design and Implementation (ACM Press, New York, NY, USA), Berlin, Germany, 2002, pp. 234–245

[33] Filliâtre, J.C., Hubert, T., and Marché, C.: 'The Caduceus verification tool for C programs', Tutorial and Reference Manual, 2007

[34] Filliâtre, J.C., and Marché, C.: 'Multi-Prover Verification of C Programs', in Clarke, E., Agha, G., and Kroening, D. (Eds.): Proc. Int. Conf. Formal Engineering Methods. Vol. 3308 of LNCS (Springer-Verlag). Seattle, USA, 2004, pp. 15–29

[35] Hubert, T., and Marché, C.: 'A case study of C source code verification: the Schorr-Waite algorithm', in Aichernig, B.K., and Beckert, B. (Eds.): Proc. Int. Conf. Software Engineering and Formal Methods (IEEE Computer Society Press), Koblenz, Germany, 2005, pp. 190–199

[36] Schoeller, B.: 'Strengthening Eiffel Contracts using Models', Int. Workshop Formal Aspects of Component Software, Pisa, Italy, 2003

[37] Schoeller, B.: 'Making classes provable through contracts, models and frames'. Ph.D. Thesis, ETH Zurich, 2008

[38] Burdy, L., Requet, A., and Lanet, J.L.: 'Java Applet Correctness: A Developer-Oriented Approach', in Araki, K., Gnesi, S., and Mandrioli, D. (Eds.): Proc. Int. Symp. Formal Methods Europe. Vol. 2805 of LNCS (Springer-Verlag). Pisa, Italy, 2003, pp. 422–439

[39] Jackson, D.: 'Software Abstractions', (MIT Press, 2006)

[40] Wahler, M.: 'Using Patterns to Develop Consistent Design Constraints'. Ph.D. Thesis, ETH Zurich, 2008

[41] Henkel, J., and Diwan, A.: 'Discovering Algebraic Specifications from Java Classes', in Cardelli, L., (Ed.): European Conf. on Object-Oriented Programming. Vol. 2743 of LNCS (Springer-Verlag). Darmstadt, Germany, 2003, pp. 431–456

[42] Sankar, S.: 'Run-time consistency checking of algebraic specifications', Proc. Int. Symp. Testing, analysis, and verification (ACM Press, New York, NY, USA), Victoria, British Columbia, Canada, 1991, pp. 123–129

[43] Antoy, S., and Hamlet, D.: 'Automatically Checking an Implementation against Its Formal Specification', IEEE Trans. on Software Engineering, 2000, 26, (1), pp. 55–69

[44] Nunes, I., Lopes, A., Vasconcelos, V.T., Abreu, J., and Reis, L.S.: 'Checking the Conformance of Java Classes Against Algebraic Specifications', in Liu, Z., and He, J. (Eds.): Proc. Int. Conf. Formal Engineering Methods. Vol. 4260 of LNCS (Springer-Verlag). Macao, China, 2006, pp. 494–513

[45] Henkel, J., and Diwan, A.: 'A Tool for Writing and Debugging Algebraic Specifications', in Proc. Int. Conf. Software Engineering (IEEE Computer Society), Washington, DC, USA, 2004, pp. 449–458