Faithful mapping of model classes to mathematical structures

Ádám Darvas
ETH Zurich
adam.darvas@inf.ethz.ch

Peter Müller
Microsoft Research
mueller@microsoft.com

Abstract

Abstraction techniques are indispensable for the specification and verification of 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 set or relation.

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 paper 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.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications—Methodologies; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Programming by contract

General Terms Verification

Keywords specification, verification, abstraction, model classes, isomorphism, Java Modeling Language

1. Introduction

Abstraction is indispensable for the functional specification and verification of object-oriented programs. Without abstraction, types with no implementation (i.e. 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 corresponding mathematical structures such as sets and tuples. This approach was pioneered by the Larch project [10], which advocated two-tiered specifications consisting of a contract and a theory providing the mathematical structures.

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

public interface Set extends Collection {
  //@ public model instance JMLObjectSet _set;

  /*@ also
    @   public normal_behavior
    @     ensures contains(o);
    @*/
  public boolean add(Object o);

  /*@ also
    @   public normal_behavior
    @     ensures \result == _set.has(o);
    @*/
  /*@ pure @*/ public boolean contains(Object o);

  // other constructors and methods omitted
}
```

Figure 1. Specification of type `Set` using model class `JMLObjectSet` defined in JML's model library. JML annotation comments start with an at-sign (@). Keyword `also` expresses that the given specification extends the specification given in supertype `Collection`. The import declaration allows one to refer to the model class. We omit `nullable` annotations for brevity.

The Java Modeling Language (JML) unifies these tiers to simplify the development of specifications [4]. 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 interface `Set`. The model class, through its pure methods, provides access to a mathematical set that contains objects. In order to use the model class, a model field `_set` is declared. This field is used for specification purposes only and is supposed to represent the abstraction of an instance of type `Set`.

One can specify `Set`'s method `contains` in an abstract way using the model field and pure method `has` declared in model class `JMLObjectSet`. Given a concrete implementation of `Set` one would define the relation (using JML's `represents` clause) between the public model field `_set` and the private internal structure.

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 as uninterpreted function symbols and axioms, respectively [6, 5, 12]. However, this approach is not optimal for model classes because the tactics of theorem provers are optimized for the prover's theories rather than encodings of JML model classes. Moreover, it is difficult to

Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), September 3-4, 2007, Cavtat near Dubrovnik, Croatia.
Copyright © 2007 ACM ISBN 978-1-59593-721-6/07/0009...$5.00

ensure soundness of such encodings, especially in the presence of recursive specifications [6].

To overcome these problems, previous work [3, 13, 14] proposes to map model classes and their pure methods directly to theories of the theorem prover at hand. For instance, a method `contains` of a model class could be mapped to the $\in$ operator of the theorem prover. However, the existing work only discusses the mapping of method signatures, but ignores their contracts. With this approach, the meaning of `contains` is given by the definition of $\in$, and not by its contract. 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 relying on the model class contract. The results may also vary between different theorem provers, which define certain operations slightly differently. 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.

In this paper, 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.

Our approach leads to important results beyond semantical correspondence. 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, our case study 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 paper we will use JML [15] as specification language and Isabelle [18] as target theorem prover. This choice was made due to the characteristics of the static program verification tool JIVE [16] that we are working on. However, the presented approach is applicable to any combination of specification languages and theorem provers, for instance, Eiffel [21] and Coq [1].

Our approach does not yet have tool support. All steps in the case study were performed manually. In Section 6, as future work we briefly mention areas where tool support could greatly help.

The rest of the paper is structured as follows. Section 2 introduces model class `JMLObjectSet`, the class we use throughout this paper to illustrate 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 gives an overview of related work and in Section 6 we conclude.

2. Running example

As running example, we take model class `JMLObjectSet`, which is part of the model library of the JML distribution. It models a set 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 the constructors and methods that we discuss in this paper.

The class and, thus, all its methods are pure. Methods that return `JMLObjectSets` (for instance, `union`) do not mutate their receiver objects but return newly created `JMLObjectSets`. In accordance with the JML semantics, all reference type arguments and return values are considered to be non-null.

The class is specified by an *equational theory* and by *method specifications*:

1. The equational theory is an object invariant expressed in terms of the static pure model method `equational_theory` which has to return true for every non-null `JMLObjectSet` instance `s2`, and objects `e1` and `e2`. The method has a large normal behavior specification case containing equations written in the style of algebraic laws. Figure 2 shows a sample equation defining method `union`.
2. Method specifications consist of pre- and postconditions attached to constructors and methods of the model class. Modifies clauses are not needed since all methods are pure. As an example, the specification of method `union` is given on Figure 2.

We follow the proposal of Leavens et al. [13] and Charles [3], and consider model classes to be final and unrelated to Java's type hierarchy rooted in type `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.

3. Faithful mappings

In this section we present our solution for proving that the mapping of a model class M to a structure S (defined by some datatype or theory) is faithful. That is, there is a semantic correspondence between M and S, namely, they are *isomorphic*.

The process consists of three stages. In the first stage, we specify the mapping of M to S by a new JML clause, `mapped_to`. Then we prove consistency and completeness of the specified mapping in the second and third stages, 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.

Figure 2 demonstrates a possible mapping of model class `JMLObjectSet`. The model class is mapped to Isabelle's `HOL/Set` theory [19], specifically to type "*'a set*". In Isabelle, *'a* is a type variable which gives rise to polymorphic types [18]. This mapping is specified by the new specification construct `mapped_to`. 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.

The `mapped_to` clause attached to the class determines the context and type to which the methods of the model class will get mapped. The mapping of the methods is specified by `mapped_to` clauses attached to the methods. For instance, method `has` is mapped to Isabelle's set membership ":". The first parameter is again the target environment, the second specifies the way the model class method is mapped to some term in the target context. The second parameter typically mentions function symbols of the target context as well as parameters (including the receiver) of

```
package org.jmlspecs.org;

//@ mapped_to("Isabelle","HOL/Set", "'a 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));
     @  also
     @    ...
     @ static public pure model boolean
     @   equational_theory(JMLObjectSet s,
     @     JMLObjectSet s2, Object e1, Object e2);
     @*/

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

   //@ mapped_to("Isabelle","{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 - {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);
}
```

Figure 2. Model class JMLObjectSet containing the signatures of members we consider in this paper. The proposed mapping of the class and its members to Isabelle is given by the mapped_to clause. The object invariant, a fragment of the equational theory and two sample method specifications are given too. Other specification elements are omitted.

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.

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 the mappings need not be specified by programmers who are typically not familiar with theorem provers and their theories and syntax. The mappings can be specified by the author of a model class or by the team which performs the verification.

3.2 Consistency

Once the mappings are specified, their 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 mappings to handle model classes in specifications. In this section, 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, definitions, theorems etc.).

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 [15]. Other method specifications describe situations when the method might throw exceptions which is not of interest for the isomorphism proof.

In the sequel, we use the term *relevant specification element* to refer either to an invariant or to a normal-behavior method specification of a model class. Every relevant specification element s_M in M needs to be translated and proved as follows:[1]

1. (*a*) If s_M is a method specification of some method m with precondition pre and postcondition $post$ then it is treated as a formula of the form "$pre \Rightarrow post$" which is universally quantified over all parameters (including the implicit receiver) of m.
   (*b*) Occurrences of every method call to some method m have to be 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).[2]
   (*c*) If s_M is a method specification of some method m, then in the postcondition all occurrences of \result (and this if m is a constructor) have to be replaced by the term prescribed in the mapped_to clause of method m.

2. The formula has to be 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 method specification is presented on Figure 2.

[1] We ignore ghost fields in this paper. They can be handled by mapping a model class M with n ghost fields to a $n + 1$-tuple, where the first component represents the structure for M and the other components represent the state of the ghost fields [17].

[2] Proving correspondence of logical operators is out of the scope of this paper.

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 on `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 elementary structures.

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 can be proven using S, too.

An important consequence of this result is that the specification of M is consistent (i.e., free of contradictions) provided S is consistent. Since structures like Isabelle's `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 `Set` theory is a *model* for model class `JMLObjectSet`. By exhibiting this model we prove that using `JMLObjectSet`'s specification does not lead to unsoundness.

This is also an interesting result concerning the use of pure methods in specification expressions. As we have shown earlier [6], the use of pure methods in specifications can easily lead to unsoundness. The solution we proposed in our earlier work [6] to prevent unsoundness is to exhibit a witness for showing that the specification of the method is satisfiable. However, the solution is not applicable for recursive specifications. With the approach presented above, recursive specifications do not pose any problems.

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.

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 member 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 an axiom as follows:
(a) If s_M is a method specification with precondition pre and postcondition $post$ attached to method m then it is treated as formula "$pre \Rightarrow post$" universally quantified over all parameters of m.
(b) Occurrences of method calls on some method m have to be replaced by function applications of the corresponding function symbol $\hat{m}$. Additionally, JML's logical connectives have to be replaced by the corresponding connectives of the theorem prover.
(c) If s_M is a method specification of some method m, then in the postcondition all occurrences of `\result` (and `this` if m is a constructor) have to be replaced by function applications of function symbol $\hat{m}$.
3. A lemma is generated from every axiom and definition s_S of S by replacing all occurrences of function symbols in s_S by the corresponding function symbols declared in step 1. Correspondence is based on the `mapped_to` clauses.
4. The lemma has to be 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 step, the signature of $is\widehat{ProperSubset}$ is declared based on the signature of method `isProperSubset`:

$$is\widehat{ProperSubset} :\ 'a\ set\ \times\ 'a\ set \Rightarrow bool$$

The second step is based on the specification of the method. For demonstration purposes, this time we take the specification prescribed by the invariant, i.e. 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 on `isProperSubset`, `isSubset` and `equals` are turned into the function applications $is\widehat{ProperSubset}(s, s2)$ $is\widehat{Subset}(s, s2)$ and $\widehat{equals}(s, s2)$, respectively. Additionally, the logical operators are mapped. The resulting formula is turned into the following axiom:

$$\forall\, s, s2, e1, e2.\ is\widehat{ProperSubset}(s, s2) =\\ (is\widehat{Subset}(s, s2) \wedge \neg \widehat{equals}(s, s2)) \tag{1}$$

In step 3, we take the definition of proper subsets from Isabelle's theory [19]:

```
psubset_def: "A < B == (A::'a set) <= B & ¬ A=B"
```

and translate it to the following lemma:[3]

$$\forall\, A, B.\ is\widehat{ProperSubset}(A, B) =\\ (is\widehat{Subset}(A, B) \wedge \neg \widehat{equals}(A, B))$$

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

Note that theorems of S need not be turned into lemmas since theorems are properties that are derived from definitions and axioms of S. However, it is important that all axioms and definitions are turned into lemmas, including the ones that do not appear in the textual representation of S. For instance, Isabelle supports inductively defined sets for which the tool generates fixed point definitions and proves several lemmas about them [18]. In such cases the artefacts introduced "under the hood" need to be turned into lemmas too. Theorem provers typically make these artefacts available for users, for instance, Isabelle can be queried to show them and PVS [22] generates separate files for them.

Proving this stage guarantees that whatever can be proved using the axioms, definitions and theorems of Isabelle's `Set`, can also be proved using JML's `JMLObjectSet`. An interesting consequence is that we have proved that the axiom system extracted from the specifications of `JMLObjectSet` is complete relative to Isabelle's theory of `Set`. Since Isabelle structures like `Set` are heavily used

[3] Isabelle definitions are implicitly universally quantified over variables that are not bound by quantifiers that appear explicitly.

in formalizations and proofs, one can be sure that they contain the most important properties of the structure.

The generation of lemmas (step 3) for this stage is not as trivial as for the consistency proof. The `mapped_to` clauses specify the mappings from M to S, which is the opposite direction of this stage. The mapping from S to M is not necessarily unique. For instance, the `=` operator of S is typically used in the `mapped_to` clauses of several methods of M, which makes it difficult to choose automatically the appropriate mapping.

Furthermore, proving the lemmas (step 4) is also less trivial than in 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, the specifications 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 for its completion. However, the effort is justified by the increased quality of the model class specification.

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 mappings prescribed by the `mapped_to` clauses were 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. 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 proved, method calls can be directly translated to the corresponding function applications without being worried about soundness issues or differences in the semantics of related methods and function symbols.

An interesting side-effect of the proposed proof technique is that redundant specifications can be discovered in the model class. 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.

4. Case study

In this section, we demonstrate our approach for the model class `JMLObjectSet` by describing in detail the process of proving faithfulness with Isabelle's `HOL/Set` theory. We highlight the interesting observations and results of the case study.

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 members that remained after the simplifications described in the next section. All proofs were carried out in Isabelle. The proof scripts contained a total of ca. 380 LOC without comments and empty lines. Consistency of the mapping was proven in ca. 100, completeness in ca. 110 LOC. Equivalence of the equational theory and the method specifications (see Section 4.5) was proven in ca. 170 LOC. All proof scripts were written manually.

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 a next step we removed all implementation details. This included all method bodies, and members and specifications not visible for clients. Additionally, we removed all public members that were only used in informal specifications or provided only syntactic sugar. As mentioned in Section 3.2, only method specification 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.

To focus on the main ideas of this paper, we decided not to handle members that referred to non-primitive types other than `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 `JMLObjectSets` to other model or non-model types. The handling of these kinds of members is possible once one has provided a mapping for the types mentioned in their signatures.

4.2 Division of specifications

We analyzed the specification of `JMLObjectSet` and found that the equational theory and method specifications contained a lot of redundancy. Many properties of the model class were attempted to be expressed both by the equational theory and by method specifications. We illustrate this by method `union`. The equation defining the method in the equational theory and its method specification is given on 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 necessarily the case that the equational theory of a model class and its method specifications contain so much redundancy. There are, for instance, JML model classes that specify the behavior of the class in great majority by method specifications (e.g., `JMLObjectToObjectRelation` and `JMLValueValuePair`). Thus, in general, faithfulness of a model class to some structure should be proven using both the equational theory and the method specifications together.

4.3 Specifying the mapping

The next step was the specification of the mapping of the model class and its methods. The resulting mapping to Isabelle's higher-order set theory `HOL/Set` 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. This 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 [18].

Another interesting case to mention was the mapping of method `remove` that takes an object `elem` as argument. Isabelle's theory contains no operation that removes a single element from the set. Thus, `remove` had to be mapped to two other set operations: creation of a singleton set and set difference: `this - {elem}`.

Finally, we mention 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. We discuss the consequences and solutions of such cases in Section 4.7.

An important issue of the mapping is the handling of equality. In general, we use reference equality for objects [3]. 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, == operator on `JMLObjectSet` instances is mapped to set equality in Isabelle.

4.4 Consistency

We proved that the specifications of the model class are implied by the properties of Isabelle's `Set` theory. The proof was performed as described in Section 3.2.

We found one unsound 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 in and from set `s` yields a set equivalent 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 `e1`, 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 equivalent 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 patched 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 (patched) specifications of the model class are consistent.

4.5 Equivalence of equational theory and method specifications

While it was easy to notice the large overlap of properties specified by the class invariant and by the 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 by 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 in 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) solved the problem. In fact, the two original specifications could as well be removed since the new one (together with other properties) implies them.

The example where the method specifications had to be strengthened is the postcondition of `JMLObjectSet`'s constructor which takes an object `e` as argument and yields a set that contains `e`. 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 do 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 either 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.

An interesting side-effect of this proof technique is that one can check whether specifications marked as redundant are indeed redundant. For instance, to check if a method specification marked as redundant is indeed implied by other method specifications, one needs to generate a lemma out of the specification marked as redundant and axioms from the non-redundant method specifications. If the lemma is provable, the specification is indeed redundant.

4.6 Completeness

As the last step we proved that the definitions of Isabelle's `Set` theory are implied by the (corrected and strengthened) specifications 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 `Set` theory use set comprehension. 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 [15]. The JML Reference Manual does not give a concrete definition for the semantics of the construct, thus we used the same meaning that Isabelle defines. This (1) ensured that we did not introduce unsoundness (provided the Isabelle 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 both kinds of specifications are strong enough to imply the elementary properties of sets.

However, there were definitions that could not be mapped back to the model class in a straightforward way. An example is function *image* which takes a function *f* and a set *A* as parameters, and yields the image of set *A* under *f*. The model class does not provide such functionality and it cannot be expressed by the use of other methods of the class. Such cases lead us to the notion of *observational faithfulness*, discussed in the next session.

4.7 Mismatches between model class and structure

So far we only dealt with situations where each method of M had a direct correspondence in S and vice versa. However, this is not necessarily the case. If there is no direct correspondence, one can try to express the operation in terms of other operations (either of M or of S) that could already be mapped (directly or indirectly). As an example, in Section 4.3 we mentioned `JMLObjectSet`'s `remove` method, which could be expressed in terms of two functions of Isabelle's `Set`. In such cases the isomorphism result still holds.

However, there might be situations when no mapping exists and the operation cannot be expressed in terms of other ones. In such cases, there is a mismatch between M and S that cannot be bridged, that is, 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 that specifications referring to the method are indeed consistent and that the method semantically corresponds to some mathematical operation. An example for this situation is method `int_size` in `JMLObjectSet`. It has no counterpart in Isabelle's `HOL/Set` theory and cannot be expressed by other methods of the model class. This means that if we use theory `HOL/Set` we can neither guarantee consistency of specifications mentioning the method nor that the semantic meaning of the method is the intended one, namely set cardinality. In such situations, one needs to pick a different target theory where the mapping is possible. In our case Isabelle's `HOL/Finite_Set` could be picked as it provides function *card* to express set cardinality.

The situation is better if an operation of S cannot be translated to M. In this case the consistency of all methods in M can still be shown and the mappings prescribed by the `mapped_to` clauses can be safely used. That is, although isomorphism of M and S cannot be proven, isomorphism of all operations accessible in M and the corresponding operations in S can be shown. We call this kind of isomorphism *observational faithfulness* which is a sufficient result for the sound use of `mapped_to` clauses.

As mentioned above, `HOL/Set`'s function *image* cannot be mapped to `JMLObjectSet`. This means that the model class and the theory are not isomorphic. However, they are observationally faithful since isomorphism can still be shown for all methods of `JMLObjectSet` that may appear in specifications (apart from method `int_size`, as mentioned above).

5. Related work

The idea of using function symbols that are understood by the backend theorem prover directly on the specification level was already present in ESC/Java [9]. The special construct `\dttfsa` (*Damn The Torpedos, Full Speed Ahead!*) allowed users to refer to function applications on the level of Simplify, the theorem prover of ESC/Java. The corresponding function symbols were defined directly on the level of the prover. While this construct was a powerful means for specification, one had to be careful with its usage since on the specification level the definitions of the function symbols were hidden. The verification system did not give support for showing that the definitions were free of inconsistencies.

The Caduceus tool is a static verification system for C programs [7]. 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 backend 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 [8, 11]. The drawback of the approach is the lack of consistency proof for definitions and axioms given on the source or prover level. This might lead to soundness issues.

Leavens et al. [14] identify the problem of specifying model types as a challenge for the specification and verification of programs. As a solution they propose the direct translation of model classes to mathematical theories, however, their proposal does not include details on how the translation would work and the issue of faithfulness is not mentioned.

Schoeller [20] 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 [21]. 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 [3] proposes the introduction of the `native` keyword to JML in the context of work on the program verifier Jack [2]. 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 backend 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 datatypes. The mapping of `native` types is defined on the Coq level, too. This approach differs mainly in two ways from ours. First, our approach ensures faithfulness of the mappings. There is no attempt to do so in the work of Charles. Second, the `mapped_to` clause we propose in this paper allows one to specify the mappings on the specification language level. Furthermore, properties of model classes are specified in JML which typically provide easier understanding (for programmers) of the semantics than definitions given directly on the level of a theorem prover.

6. Conclusion

For the static verification of programs, model classes have to be mapped to mathematical structures of the underlying theorem prover. In this paper, we proposed an approach to show that this mapping is 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 does 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 specifications of pure methods does not provide a satisfying solution in the presence of recursion [6]. The solution proposed by this paper 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.

To demonstrate our approach, we did a case study with a model class from JML's model library and a theory from Isabelle's library. The case study was successful in that observational faithfulness could be proved (except for method `int_size`) and interesting observations could be made on the model class: an incorrect specification was revealed, missing specifications were identified, and a precise relation between its equational theory and method specifications was identified.

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

For a better understanding of the strengths and weaknesses of our approach, further case studies with more complex model classes need to be done. In particular, it would be interesting to see how well our approach works for model classes that do not have a directly corresponding theory in the theorem prover, e.g., a stack.

Acknowledgments. We are grateful to Vijay d'Silva and Farhad Mehta for interesting discussions, and to the reviewers for helpful comments. Müller's work was done at ETH Zurich and 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.

References

[1] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[2] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *FME*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[3] J. Charles. Adding Native Specifications to JML. In *Formal Techniques for Java-like Programs*, 2006.

[4] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract: Research articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[5] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.

[6] A. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *JOT*, 5(5):59–85, 2006.

[7] J.-C. Filliâtre, T. Hubert, and C. Marché. The Caduceus verification tool for C programs. Tutorial and Reference Manual. 2007.

[8] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *ICFEM*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, J. B. S. G. Nelson, and R. Stata. Extended static checking for Java. In *PLDI*, volume 37, pages 234–245. ACM Press, 2002.

[10] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[11] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM*. IEEE Comp. Soc. Press, 2005.

[12] B. Jacobs and F. Piessens. Verification of programs with inspector methods. In *Formal Techniques for Java-like Programs*, 2006.

[13] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–205, 2005.

[14] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007. To appear.

[15] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. *JML Reference Manual*. Iowa State University, Last revised February 2007.

[16] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. 2000.

[17] M. Miragliotta. Specification model library for the interactive program prover JIVE. ETH Zurich, Semester Thesis, 2004.

[18] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[19] T. Nipkow, L. C. Paulson, and M. Wenzel. Theory HOL/Set from "The Isabelle Library". `isabelle.in.tum.de/library/HOL/Set.html`, 2005.

[20] B. Schoeller. Strengthening Eiffel contracts using models. In *Formal Aspects of Component Software*, 2003.

[21] B. Schoeller, T. Widmer, and B. Meyer. Making specifications complete through models. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*. Springer, 2006.

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