Proving Consistency and Completeness of Model Classes Using Theory Interpretation

Ádám Darvas and Peter Müller

ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch

Abstract. Abstraction is essential in the formal specification of programs. A common way of writing abstract specifications is to specify implementations in terms of basic mathematical structures. Specification languages like JML offer so-called model classes that provide interfaces to such structures. One way to reason about specifications that make use of model classes is to map model classes directly to structures provided by the theorem prover used for verification. Crucial to the soundness of this technique is the existence of a semantic correspondence between the model class and the related structure. In this paper, we present a formal framework based on theory interpretation for proving this correspondence. The framework provides a systematic way of determining the necessary proof obligations and justifies the soundness of the approach.

1 Introduction

Abstraction is essential in the formal specification of programs because it allows one to write specifications in an implementation-independent way, which is indispensable for information hiding. Furthermore, abstraction facilitates the readability and maintainability of specifications. A common way of writing abstract specifications is to specify implementations in terms of well-known mathematical structures, such as sets and relations. This technique is applied, for instance, in VDM [10], Larch [8], and OCL [19]. While these approaches describe the mathematical structures in a language that is different from the underlying programming language, another approach is that of the Java Modeling Language (JML) [13], which simplifies the development of specifications by describing the structures through *model classes* [2]. Model classes are immutable and are used only for specification purposes. They provide object-oriented interfaces for essential mathematical structures through their side-effect free (*pure*) methods.

Specifications can be written in an abstract way by expressing properties in terms of model classes and their operations. Fig. 1 shows a class `SingletonSet` specified in JML using the model class `JMLObjectSet` (presented in Fig. 2), which represents a mathematical set of objects. To use the model class, we declare the public specification-only model field `_set`. The field represents the abstraction of an instance of type `SingletonSet` as specified by the private **`represents`** clause: a singleton set containing the object referenced by the private field `value`. Given model field `_set` and `JMLObjectSet`'s public pure method `has`, which checks

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

    //@ ensures _set.has(o);
    public void setValue(Object o)
    { value = o; }
}   // other constructors and methods are omitted for brevity
```

Fig. 1. Specifying class `SingletonSet` using model class `JMLObjectSet`. JML annotation comments start with an at-sign (`@`).

for set membership, one can specify `SingletonSet`'s `setValue` method in an abstract way, in particular, without referring to the private field `value`.

While model classes provide a powerful means for writing abstract specifications, they pose a problem for static verification: program verifiers have to encode specification expressions in the logic of the underlying theorem prover, in particular, calls to the pure methods of model classes.

Previous work proposes to map model classes and their methods directly to structures and function symbols provided by the theories of the underlying theorem prover [1, 11, 12, 4]. Calls to model-class methods are encoded as applications of these function symbols. For instance, if `JMLObjectSet`'s method `has` is mapped to symbol '$\in$' denoting set membership of a particular structure, then every call to `has` is encoded as an application of '$\in$'. Such an encoding leads to proof obligations that are handled well by theorem provers, which typically provide theories with numerous theorems for elementary structures.

Crucial to the soundness of this technique is to ensure that the mapping is *faithful*, that is, the semantics of related model classes and structures match. However, previous work mostly discusses the mapping of method signatures, but ignores their contracts. With this approach, for instance, the meaning of method `has` is given by the definition of symbol '$\in$' of the given theory, 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: (1) program verifiers might produce results that come unexpected for one who relies on the contract, (2) results may vary between different theorem provers, which define certain operations slightly differently, and (3) the result of runtime assertion checking might differ from that of static verification if the model-class implementation used by the checker is based on contracts.

Our previous work takes the contracts of model classes into account and describes the main ideas behind an approach that checks if the mapping of a model class to a mathematical structure is faithful [4]. In this paper, we present a formal framework for checking the faithfulness of mappings. This framework defines precisely the proof obligations needed to show faithfulness and guarantees soundness. Our framework applies the concept of *theory interpretation* [22, 24,

5], which allows one to compare the "strength" of two theories, T and T', whose language (*i.e.*, set of nonlogical symbols) possibly differ. A theory interpretation of T in T' is based on a syntactic notion, a *standard translation* Φ between the terms and formulas of T and T'. Φ is a *standard interpretation of* T *in* T' if $\Phi(\phi)$ is a theorem of T' for each theorem ϕ of T. A theorem important for our purposes is that if there is a standard interpretation of T in T', and T' is consistent, then T is consistent.

Our approach applies the concept of theory interpretation in three stages. In the first stage, we specify the mapping of a model class to an existing theory of the underlying theorem prover. In the second stage, we attempt to formally prove that the specified mapping of a model class defines a standard interpretation of the theory formed by the specification of the model class in the theory of the corresponding structure. If the proof attempt succeeds then *consistency* of the model-class specification is guaranteed. Although consistency is only relative to the consistency of the target theory, theorem provers are unlikely to contain inconsistent theories. We will refer to this stage as the *consistency proof*.

In the third stage, we attempt to "reverse" the specified mapping and attempt to prove that the reverse mapping defines a standard interpretation of the target theory in the specification of the model class. If the proof attempt succeeds then *completeness* of the model-class specification is guaranteed. Again, completeness is only relative to the corresponding theory, but theorem provers typically define theories with rich sets of properties. In contrast to our earlier work, we define a condition that ensures the existence of a suitable reverse mapping. We will refer to the third stage as the *completeness proof*.

We have presented the main idea of faithfulness proofs earlier [4]. However, the precise conditions that are necessary to ensure soundness are subtle, and our previous work did not contain a soundness argument. The advantage of building our approach on the well-studied concept of theory interpretation is that the correctness of our approach is guaranteed by the correctness of the concept. In particular, theory interpretation takes the universes of structures into account, which is crucial for the soundness of the mapping of model classes [3] and is not present in previous work.

Although we use JML as specification language and Isabelle [18] as theorem prover in this paper, the presented approach is applicable to any combination of specification language and theorem prover, for instance, Eiffel [17] and PVS [20].

Outline. The next section introduces a model class that will serve as running example and the specification means for mapping model classes. Sections 3, 4, and 5 present the formal details of faithfulness proofs: the way universe predicates are defined, the consistency proof, and the completeness proof. In Sec. 6, we discuss related work and conclude. We refer the reader to the PhD dissertation of Darvas [3] for a case study of the presented approach, an extension to mappings whose target structure is defined inductively, and practical considerations for the case when related model classes and structures do not match perfectly.

```
//@ mapped_to("Isabelle", "HOL/Set", "α set");
/*@ immutable pure @*/ public class JMLObjectSet {
  //@ 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) { ... }

  /*@ 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) { ... }

  //@ mapped_to("Isabelle", "this Un s2");
  public JMLObjectSet union(JMLObjectSet s2) { ... }

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

Fig. 2. Signatures and mappings of `JMLObjectSet`'s constructors and methods that we consider in this paper. Implementations are omitted.

2 Encoding of Model Classes

Model Class `JMLObjectSet`. The class is part of JML's model library and encodes sets of objects: it provides the usual operations of mathematical sets; equality over the set-elements is based on Java's reference equality ("`==`"). Fig. 2 presents those constructors and methods that are discussed in the sequel.

The class is specified to be pure (meaning that all instance methods are pure) and immutable. It is specified by invariants and method specifications. The invariants of model classes are special in that they do not restrict the state space of model-class instances as invariants usually do. Instead, they give equational laws about their operations, thus they play a similar role as method specifications. Therefore, for brevity, we omit the handling of invariants here, but see [3].

A sample method specification is given in Fig. 2 for method `insert`. The proposed mapping of the class and its operations to one of Isabelle's set structures is given by `mapped_to` clauses that we introduce below.

Specifying the Mapping. In the first stage of a faithfulness proof, one specifies the mapping of the model class at hand. In our previous work [4], we introduced the `mapped_to` clause for this purpose. The mapping of a model class is specified by a `mapped_to` clause attached to the class. The first argument of the clause specifies the target theorem prover, the second the target theory, and the third the specific type (if any) in the theory to which the model class is mapped. Similarly, a method can be mapped to a term of the target theory of a given theorem prover by a `mapped_to` clause attached to it. The term must be well-typed and may only mention logical and nonlogical symbols of the target theory and parameters (including the explicit receiver) of the specified constructor or method. Only one clause per target theorem prover may be specified both for a model class and for a method.

For instance, in Fig. 2 class `JMLObjectSet` is mapped to type α *set* in the `HOL/Set` theory of Isabelle; and method `has` is mapped to the term *elem* : *this*, meaning that the method corresponds to Isabelle's set membership operator ":".

We permit one to write arbitrarily complex terms in `mapped_to` clauses, which allows us to support model methods with functionality that is not directly provided by the target prover. This flexibility is necessary to handle, for example, `JMLObjectSet`'s `remove` method, which removes a single element of a set. Theory `HOL/Set` does not provide a corresponding operation but provides set difference, which allows one to express the meaning of method `remove`.

As different theorem provers provide different theories with different symbols and semantics, we allow mappings to multiple provers. Thus, the faithfulness proof has to be carried out in every target prover specified in `mapped_to` clauses.

Contexts and Auxiliary Functions. The contexts in which the consistency and the completeness proofs are carried out are not the same. The context of the former is that of the target theory T, for instance, Isabelle's `HOL/Set` theory. The context of the latter will be denoted by $\widehat{M}$, which is the logical encoding of model class M's specification. The encoding allows one to carry out the completeness proof in a formal system, like Isabelle or PVS [20]. Note that merely analysing the encoded specification in context $\widehat{M}$ would not be sufficient for the sound use of the `mapped_to` clause for verification purposes, because only consistency of the specification could be shown; its semantic correspondence to target theory T could not be justified.

We introduce function γ that encodes JML specification expressions in context $\widehat{M}$. The function takes a JML expression and yields a first-order term or formula (denoted by *FOL*) in $\widehat{M}$. Its signature is γ: $Expr \rightarrow FOL_{\widehat{M}}$. Note that γ takes no argument for the state. This is because instances of model classes behave like mathematical values rather than heap-allocated objects. The definition of the function for a small but representative subset of JML is the following [3]:

$$\gamma(E \circledast F) \triangleq \gamma(E)\ \ Tr(\circledast)\ \ \gamma(F),\ \text{if } \circledast \in \{\texttt{\&\&}, \texttt{||}, \texttt{==>}, \texttt{==}, \texttt{!=}, \texttt{+}, \texttt{-}, \texttt{/}, \texttt{\%}\}$$

$$\gamma(\texttt{v}) \triangleq v \qquad \gamma(\texttt{new C}(E)) \triangleq \widehat{C}(\gamma(E))$$

$$\gamma(\texttt{!}E) \triangleq \neg\gamma(E) \qquad \gamma(\texttt{(\forall } T\ x.\ E\texttt{)}) \triangleq \forall\, x.\ \gamma(E)$$

$$\gamma(E\texttt{.m(}F\texttt{)}) \triangleq \widehat{m}(\gamma(E), \gamma(F)) \qquad \gamma(\texttt{(\exists } T\ x.\ E\texttt{)}) \triangleq \exists\, x.\ \gamma(E)$$

An application of a binary JML operator $\circledast$ is encoded by an application of the corresponding operator in the underlying logic yielded by function *Tr* to the encoding the two operands. *Tr* is a function that maps the binary operators of JML to their equivalents in first-order logic (*i.e.*, $\wedge$, $\vee$, $\Rightarrow$, $=$, $\neq$, etc.). Calls to methods and constructors are encoded by applications of uninterpreted function symbols. We will use the convention that a method `m` is encoded by symbol $\widehat{m}$. Note that the encoding of old-expressions (used in postconditions to refer to values in the pre-state of the specified method) is not given. This is because old-expressions are not meaningful in model-class specifications.

Next, we introduce function ν. A standard translation Φ of T in T' is a pair (U, ν), where U is a closed unary predicate, the *universe predicate*, and ν is a function that maps all nonlogical symbols of T to a λ-expression of T' [5]. Given U and ν, the translation of terms and formulas of T can be defined in a straightforward way [5].

Following this notation, we use function ν to map methods and constructors to λ-functions of the target theory. As `mapped_to` clauses contain exactly that information, function ν essentially captures the content of these clauses. For instance, in model class `JMLObjectSet` we have:[1]

$$\nu(\texttt{remove}) \;\equiv\; \lambda\{\, \mathit{this}, \mathit{elem}.\ \mathit{this} - (\mathit{insert}\ \mathit{elem}\ \{\})\,\}$$

The purpose of the universe predicate and the way it is specified in model classes is described in the next section.

3 Specifying the Universe

When relating two theories, it is possible that the set of possible elements (the *universe*) of the source and the target theory differ. In such cases, the scope of quantifiers and, therefore, the semantics of quantified formulas possibly differs in the two theories. The concept of theory interpretation solves this problem by introducing the unary universe predicate U, which yields true if and only if its argument denotes an element of the target structure that is meant to be in the scope of quantifiers, and thereby, in the scope of translation Φ. Given the universe predicate, translation Φ "relativizes" quantifiers:

$$\Phi(\forall x.\ \phi) \;\triangleq\; \forall x.\ U(x) \Rightarrow \Phi(\phi) \quad \text{and} \quad \Phi(\exists x.\ \phi) \;\triangleq\; \exists x.\ U(x) \wedge \Phi(\phi)$$

The dissertation of Darvas [3, Example 9.1] demonstrates the need for relativization in the context of mapping model classes to mathematical theories.

To allow users to specify the set of operations that should form the universe predicate of a model class, we introduce the **`constructing`** modifier that may be attached to constructors and methods. The universe predicate is the same for the consistency and for the completeness proofs, only the context in which the predicate is expressed differs: When proving consistency, the context is that of the target theory T; when proving completeness, the context is $\widehat{M}$. Accordingly, we will denote the predicates by U_T and $U_{\widehat{M}}$.

[1] As a second argument, ν should take the name of the target theorem prover. For simplicity, we omit this argument as the target prover will be Isabelle in the sequel.

Given a model class M with l methods and constructors marked with modifier **`constructing`**, the resulting universe predicate is a disjunction with l disjuncts. The disjunct of the universe predicate $U_{\widehat{M}}(x)$ for a **`constructing`** method m with one implicit and n explicit parameters, and precondition P is:

$$\exists\, s, e_1, \ldots, e_n.\ U_{\widehat{M}}(s) \wedge U_{\widehat{M}}(e_1) \wedge \ldots \wedge U_{\widehat{M}}(e_k) \wedge \gamma(P) \wedge \widehat{equals}(x, \widehat{m}(s, e_1, \ldots, e_n))$$

where $k \leq n$ and where we assume for simplicity that parameters $e_1, \ldots, e_k$ are of the enclosing type, while the others are not. The construction of predicate U_T is analogous in the context of T:

$$\exists\, s, e_1, \ldots, e_n.\ U_T(s) \wedge U_T(e_1) \wedge \ldots \wedge U_T(e_k) \wedge \Phi_M(P) \wedge \Phi_M(x\texttt{.equals(}s\texttt{.m(}e_1, \ldots, e_n\texttt{))})$$

where Φ_M is the translation function between JML expressions and the target context. The function is precisely defined in the next section.

The treatment of constructors is analogous. As an example, if the parameterless constructor and method `insert` of model class `JMLObjectSet` are marked as **`constructing`** then we get the following universe predicates:

$$U_{\widehat{M}}(x) \triangleq \widehat{equals}(x, \widehat{JMLObjectSet}()) \vee (\exists\, s, e.\ U_{\widehat{M}}(s) \wedge \widehat{equals}(x, \widehat{insert}(s, e)))$$

$$U_T(x) \triangleq x = \{\} \vee (\exists\, s, e.\ U_T(s) \wedge x = (insert\ e\ s))$$

If no method is marked as **`constructing`** then the universe predicate is *true*. This is the case when the source and the target universe is the same.

4 Proving Consistency of a Model Class

In the second stage of the faithfulness proof, we prove consistency of the mapping: we show that there is a standard interpretation of M's theory in theory T. To do so, first we define the translation of JML expressions in the context of T based on `mapped_to` clauses. The resulting translation function will be denoted by Φ_M. Second, we attempt to prove that Φ_M is a standard interpretation.

Definition of Φ_M. The function takes a JML expression and yields a term or formula in the target context. Its signature is Φ_M: $Expr \rightarrow FOL_T$ and its definition is presented in Fig. 3. For simplicity, the definition for method and constructor calls, for keyword **`\result`**, and for keyword **`this`** in the postcondition of constructors is presented for methods with one explicit parameter p. Note that terms $\nu(\texttt{m})(this, p)$ and $\nu(\texttt{C})(p)$ denote the terms that are defined by the `mapped_to` clause of the corresponding method and constructor.

Proving that Φ_M is a Standard Interpretation. To prove that translation function Φ_M is a standard interpretation of M's theory in theory T, we need to prove that three sufficient obligations hold [5].

$$
\begin{array}{lcl}
\Phi_M(E \circledast F) & \triangleq & \Phi_M(E)\ Tr(\circledast)\ \Phi_M(F),\ \text{if } \circledast \in \{\texttt{\&\&}, \texttt{||}, \texttt{==>}, \texttt{==}, \texttt{!=}, \texttt{+}, \texttt{-}, \texttt{/}, \texttt{\%}\} \\
\Phi_M(\texttt{!}E) & \triangleq & \neg \Phi_M(E) \\
\Phi_M(E\texttt{.m(}F\texttt{)}) & \triangleq & \nu(\texttt{m})(\Phi_M(E), \Phi_M(F)) \\
\Phi_M(\texttt{new C(}E\texttt{)}) & \triangleq & \nu(\texttt{C})(\Phi_M(E)) \\
\Phi_M(\texttt{\textbackslash result}) & \triangleq & \nu(\texttt{m})(\mathit{this}, p), \qquad \text{where m is the enclosing method} \\
\Phi_M(\texttt{this}) & \triangleq & \nu(\texttt{C})(p)\,, \qquad \text{if this occurs in the postcondition of constructor C} \\
\Phi_M(\texttt{v}) & \triangleq & v\,, \text{ if v is a parameter or literal other than \textbackslash result and this in postconditions of constructors}
\end{array}
$$

$$
\begin{array}{lcl}
\Phi_M(\texttt{(\textbackslash forall}\, T\ x\texttt{.}\ E\texttt{))} & \triangleq & \forall\, x.\ \ U_T(x) \Rightarrow\ \Phi_M(E) \\
\Phi_M(\texttt{(\textbackslash exists}\, T\ x\texttt{.}\ E\texttt{))} & \triangleq & \exists\, x.\ \ U_T(x) \wedge\ \Phi_M(E)
\end{array}
$$

where the shaded parts are added only if the quantified variable is of a model type.

Fig. 3. Definition of translation Φ_M.

Axiom Obligation. The obligation requires that the translation of every axiom of M is a theorem of T. The "axioms" of a model class are its method specifications. Their translation is straightforward, only the free variables have to be bound by universal quantifiers since these quantifications are implicit in method specifications. The specification of a method of class C with one explicit parameter p of type T, precondition P, and postcondition Q is translated to:

$$\Phi_M(\texttt{(\textbackslash forall}\ C\ \texttt{this. (\textbackslash forall}\ T\ p.\ P\ \texttt{==>}\ Q\texttt{)))}$$

which is equivalent to:

$$\forall\ \mathit{this}, p.\ (U_T(\mathit{this}) \Rightarrow (\ U_T(p) \Rightarrow\ \Phi_M(P\ \texttt{==>}\ Q)\ )),$$

where the shaded part is only added if p is of a model type. The formulas are turned into lemmas and have to be proved in the target theory.

Universe Nonemptiness Obligation. The obligation requires that the universe of the translation is nonempty: $\exists x.\ U_T(x)$. This is usually trivial to prove. For instance, for class `JMLObjectSet`, picking $\{\}$ for x trivially discharges the obligation for the universe predicate presented on the preceding page.

Function Symbol Obligation. The obligation requires that for each symbol f of the source theory, the interpretation of f is a function whose restriction to the universe takes values in the universe. When applying the obligation to methods of model classes, the only difference is that preconditions have to be taken into account. We have to prove for each model-class method m with n explicit parameters and precondition P that the following holds in the target theory:

$$
\begin{array}{l}
\forall t, x_1, \ldots, x_n.\ U_T(t) \Rightarrow U_T(x_1) \Rightarrow \ldots \Rightarrow U_T(x_k) \Rightarrow \\
\qquad\qquad \Phi_M(P(t, x_1, \ldots, x_n)) \Rightarrow U_T(\Phi_M(t\texttt{.}m\texttt{(}x_1, \ldots, x_n\texttt{)}))
\end{array}
\tag{1}
$$

where $k \leq n$ and where we assume that $x_1, \ldots, x_k$ are of model types, while the others are not. The proof obligations for constructors are analogous.

The second stage of the faithfulness proof is successfully completed if the three obligations can be proven. Based on the concept of theory interpretation, we can then conclude that the specification of the model class at hand is consistent provided that the target theory is consistent.

Having proved consistency of the specification of a model class ensures that it can be safely used for reasoning about client code. However, the consistency proof does not ensure that specified `mapped_to` clauses can be used for verification purposes [4]. Assume method m of a model class was mapped to symbol f, which was specified to possess properties that m did not. The verification of specifications that rely on m may lead to results that are not justified by the model-class specifications because, after having mapped method m to symbol f, the method would be endowed with all the additional properties that f possessed. The results may also diverge between different theorem provers, which define certain operations slightly differently. Furthermore, the results of runtime assertion checking might diverge from the results of static verification if the model class implementation used by the runtime assertion checker is based on the model class contract.

To fix this issue, we need to show that method m indeed possesses all endowed properties. Thus, proving completeness of a model class with respect to a theory does not just show that the specification of the class is strong enough relative to the theory, but is crucial for the sound use of `mapped_to` clauses during the verification of client code.

5 Proving Completeness of a Model Class

In the third stage of the faithfulness proof, we prove completeness of the mapping, that is, we show that there is a standard interpretation of theory T in M's theory. To do so, first we define function Φ_S that translates terms and formulas of the target theory in the context of the model class. Second, we attempt to prove that Φ_S is a standard interpretation.

Issues of Reverse Mappings. The `mapped_to` clauses provide the basis for the translation of JML expressions to terms and formulas of the target context. However, for the completeness proof, we need a translation in the other direction. In the following, we show that translation Φ_S may not be an arbitrary translation for which we can show that it is a standard interpretation. The translation should be one that is derived from the mapping prescribed by `mapped_to` clauses. That is, we need a way to *reverse* the specified mapping, which is not trivial.

Assume that in the above example not only m, but another method n was mapped to symbol f. When proving completeness of the mapping, we would need to show that not only m, but also n possesses all properties that f has. Otherwise, n might be endowed with properties that it does not possess when the method is mapped to f.

For instance, consider symbol *insert* of theory `HOL/Set`, which is mapped to both by method `insert` and by the one-argument constructor of model class

`JMLObjectSet`. Therefore, the translation of a formula that contains an application of the symbol should consider mapping the symbol both to the method and to the constructor. Although this seems to be doable by defining Φ_S such that all possible reverse mappings of a symbol must be taken into account, clearly, such a translation would not be *standard* anymore (as ν would not be a function). Furthermore, since $\nu(\texttt{JMLObjectSet(e)})(e) \equiv \mathit{insert}\ e\ \{\}$,[2] the translation of the general term $\mathit{insert}\ x\ Y$ to the constructor is only valid *under the condition* that Y corresponds to the empty set. This condition would need to be added to the translated formula, again showing that the translation would not be standard. Consequently, the concept of theory interpretation would not apply.

Moreover, the problem is not merely that the resulting reverse translation would not be standard: the conditions under which certain mappings are valid may alter the semantics and satisfiability of the original formula. It is well-known that a condition over a universally bound variable has to be added as the premise of an implication, otherwise the condition has to be added as a conjunct. However, if a condition contains both an existentially and a universally quantified variable then the condition can be added neither as a premise, nor as a conjunct.

To sum up, the general reversal of translation Φ_M would not be standard, would considerably change the structure of translated formulas, and (in certain cases) would alter the semantics of translated formulas. Thus, it would be difficult to reason that the resulting translation is indeed the one we are looking for.

Our Pragmatic Approach. To resolve the problem, we take a pragmatic approach and pose a requirement on the user-defined mappings. In practice, the requirement typically does not constrain the way model classes may be written and mapped, but it ensures that the "reverse" translation of Φ_M is a standard translation and can be easily derived from Φ_M.

Besides the requirement, a number of proof obligations will be posed on the operations of the model class at hand. In the remainder of this section, we formalize the requirement, the translation Φ_S, and the necessary proof obligations.

Requirement. The requirement we pose on specified `mapped_to` clauses is that each symbol of the target theory T should be mapped to by at least one model method *unconditionally*. Formally:

> For each n-ary function and predicate symbol f of T and variables $x_1, \ldots, x_n$ there is at least one method m or constructor C, and expressions $e_1, \ldots, e_k$ with free variables $x_1, \ldots, x_n$ such that either $\Phi_M(e_1.m(e_2, \ldots, e_k)) = f(x_1, \ldots, x_n)$ or $\Phi_M(\texttt{new}\ C(e_1, e_2, \ldots, e_k)) = f(x_1, \ldots, x_n)$ holds. (2)

Although the requirement does not hold for arbitrary mappings, it typically holds for model classes. Conditional mappings are typically needed when a model

[2] We will write `JMLObjectSet(e)` to refer to the one-argument constructor even when only a method or constructor name is expected, like the argument of function ν.

$$\Phi_S(Var) \triangleq Var$$

$$\Phi_S(f(t_1,\ldots,t_n)) \triangleq \begin{cases} \gamma(e_1\texttt{.m(}e_2,\ldots,e_k\texttt{)}), \text{ if there is a method m and} \\ \qquad\qquad \text{expressions } e_1,\ldots,e_k \text{ such that:} \\ \qquad\qquad \Phi_M(e_1\texttt{.m(}e_2,\ldots,e_k\texttt{)}) = f(t_1,\ldots,t_n) \\ \gamma(\texttt{new C(}e_1,e_2,\ldots,e_k\texttt{)}), \text{ if there is a constructor C and} \\ \qquad\qquad \text{expressions } e_1,\ldots,e_k \text{ such that:} \\ \qquad\qquad \Phi_M(\texttt{new C(}e_1,e_2,\ldots,e_k\texttt{)}) = f(t_1,\ldots,t_n) \end{cases}$$

$\Phi_S(t_1 = t_2) \triangleq \Phi_S(t_1) = \Phi_S(t_2)$, if t_1 and t_2 are not of model type; otherwise handled the same way as predicate symbols

$\Phi_S(\neg\,\phi) \triangleq \neg\,\Phi_S(\phi)$ $\quad \Phi_S(true) \triangleq true$ $\quad \Phi_S(false) \triangleq false$

$\Phi_S(\phi_1 \circ \phi_2) \triangleq \Phi_S(\phi_1) \circ \Phi_S(\phi_2)$, if $\circ \in \{\wedge, \vee, \Rightarrow\}$

$\Phi_S(\forall x.\ \phi) \triangleq \forall x.\ U_{\widehat{M}}(x) \Rightarrow \Phi_S(\phi)$ $\quad \Phi_S(\exists x.\ \phi) \triangleq \exists x.\ U_{\widehat{M}}(x) \wedge \Phi_S(\phi)$

where the shaded parts are added only if the quantified variable is of the type to which the model class was mapped.

Fig. 4. Definition of translation Φ_S.

class offers methods that are redundant in the sense that they are equivalent to some compound expression consisting of calls to more basic methods. For instance, method `remove` is equivalent to set difference with a singleton set as second argument. Such methods make the use of model classes more convenient, whereas mathematical structures typically avoid this redundancy.

The requirement would not hold, for instance, if class `JMLObjectSet` provided method `remove`, but not method `difference`.

Definition of Φ_S. The reverse translation Φ_S is a transformer between terms and formulas of context T and $\widehat{M}$. Its signature is Φ_S: $FOL_T \rightarrow FOL_{\widehat{M}}$. Given requirement (2), it can be easily defined. The definition of translation Φ_S for the standard syntax of first-order logic is presented in Fig. 4. Translation Φ_S is identical to translation Φ described in the literature [22, 5], except that the translation of function and predicate symbols is not based on function ν but on the reversal of translation Φ_M, as expressed by the condition.

If there are multiple methods or constructors that satisfy the condition then any of them can be selected since their equivalence has to be formally proven, as we will see below.

Note that the translation of operator "=" is different if the operands are of model types and if they are of some other type. In the former case, the definition over function and predicate symbols apply: to which model method the operator is mapped depends on the user-specified mapping. In practice, it is typically (but not necessarily) the `equals` method.

If the operands are not of model type, then "=" is translated to "=" (or the equivalent symbol of the target prover). Although this is in line with the definition of function Φ, it might not be the desired translation: one might want

to define equality over the elements of a model class by the `equals` method of the specific element type at hand, and not by reference equality. For brevity, we omit this issue here and refer to the dissertation of Darvas [3] for a solution.

Proof Obligations. The requirement on mappings prescribes that there should be at least one unconditional mapping for each symbol of T. However, it does not rule out methods with mappings that can be reversed only conditionally, such as the reverse mapping of symbol *insert* to the one-argument constructor. Therefore, what remains to be shown is that the functionalities of methods that are mapped to the same symbol of T are equivalent provided that the condition (if any) under which their mapping can be reversed holds.

For instance, we need to prove that the functionality of a call to the one-argument constructor `JMLObjectSet(e)` is equivalent with that of method `insert` provided that the receiver object of the method denotes the empty set.

This kind of proof obligations can be formalized as follows. Assume that for some symbol f, method m fulfills requirement (2). Then for each method n that is also mapped to symbol f (even if n also fulfills the requirement), we have to show that the following holds in context $\widehat{M}$:

$$\forall\, x_1, \ldots, x_p,\ y_1, \ldots, y_q.$$
$$\Phi_S(t_m^1 = t_n^1) \wedge \ldots \wedge \Phi_S(t_m^k = t_n^k) \ \Rightarrow\ \widehat{m}(x_1, \ldots, x_p) \stackrel{eq}{=} \widehat{n}(y_1, \ldots, y_q)$$

where (1) symbol $\stackrel{eq}{=}$ denotes operator "=" if the operands are not of model type, otherwise an application of the hat-function to which symbol "=" is translated by Φ_S (*i.e.*, typically function $\widehat{equals}$); and (2) the $t_m^i = t_n^i$ equalities are derived by applying translation function Φ_M on methods m and n, and taking pairwise the i-th arguments of the resulting function applications. Formally:

$$\Phi_M(x_1\,.\,m(x_2, \ldots, x_p)) = f(t_m^1, \ldots, t_m^k)$$
$$\Phi_M(y_1\,.\,n(y_2, \ldots, y_q)) \ = f(t_n^1, \ldots, t_n^k)$$

Proving that Φ_S is a Standard Interpretation. It remains to prove that Φ_S is a standard interpretation. The procedure is the same as for translation Φ_M: we have to show that the three sufficient obligations hold for the standard translation Φ_S.

First, the context and theory in which the obligations are to be proven needs to be constructed. As noted above, the context is denoted by $\widehat{M}$, and the theory is formed by the axiom system that is extracted from the specification of model class M. In the sequel, we will call this theory the *model theory* and assume that method signatures in M only refer to the enclosing type and type `Object`. In practice, this is typically the case for methods and constructors that correspond to the operations of the mathematical structure that M represents.

The model theory is obtained in three simple steps for a model class M:

1. Two new types are declared: *Object* and M.
2. Each method m of M is turned into a function symbol $\widehat{m}$ and its signature is declared based on m's signature using the two newly declared types.

3. Each method specification of M is turned into an axiom. For the specification of method m with parameter p, precondition P, and postcondition Q, the axiom is: $\forall\, this, p.\ \gamma(P$ `==>` $Q[$`this.`$m(p)/\$`result`$])$.
   For a constructor C, the substitution to perform on Q is $C(p)/$`this`.

Once the model theory is created, we have to show that the formulas that correspond to the three sufficient obligations for Φ_S are theorems of the model theory. The obligations are analogous to those of the consistency proof. To prove the axiom obligation, we have to show that for every axiom and definition ϕ of T, formula $\Phi_S(\phi)$ is a theorem of the model theory.

The universe nonemptiness obligation requires one to prove that universe $U_{\widehat{M}}$ is nonempty: $\exists\, x.\ U_{\widehat{M}}(x)$. As for the consistency proof, the obligation is typically trivially provable. The function symbol obligation is analogous to the corresponding obligation (1) on page 8 for the consistency proof. Predicate P corresponds to the domain restriction (if any) of the function at hand.

The third stage of the faithfulness proof is successfully completed if the three obligations can be proven. Based on the concept of theory interpretation, we can then conclude that all theorems of the target theory follow from the specification of the model class. That is, the specification is complete relative to the target theory. As discussed before, completeness allows a program verifier to prove properties in the target theory without creating results that cannot be explained by the model class specification. Moreover, failing to prove completeness typically indicates that the model-class specification is not complete. By adding the missing cases, the quality of the model-class specification improves.

6 Related Work and Conclusion

The concept of theory interpretation has already been used for formal program development. For instance, Levy applied theory interpretation to formally show the correctness of compiler implementations [14]; the work of Maibaum *et al.* (*e.g.*, [15]) and the Specware tool [23] applies the concept together with other formal machinery for the construction of formal specifications and their refinement into programs; and the theorem prover Ergo applies the concept to maximize theory reuse [9].

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 [7], which uses such function symbols instead of pure-method calls in specifications. However, the meaning of the symbols is hidden on the specification level, and the tool does not give support for showing consistency of their definitions.

Similarly, Caduceus [6] allows one to declare predicates that can be defined or axiomatized either on the source level or in the back-end prover [16]. However, there is no consistency proof for the user-provided definitions and axioms.

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. Our approach allows more flexibility by allowing `mapped_to` clauses to contain arbitrary terms of the target context.

Charles [1] proposes the introduction of the `native` keyword to JML with the meaning that methods marked as `native` introduce uninterpreted function symbols that can be defined on the level of the underlying theorem prover. Furthermore, the `native` keyword may also be attached to classes meaning that such classes get mapped to corresponding data types of the underlying prover.

Charles' 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, `mapped_to` clauses allow 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.* [12] identify the problem of specifying model classes as a research challenge. They propose two possible solution approaches that are related to our work and summarize the open problems for both of them. One approach considers automatic translations between model classes and mathematical structures, and the authors argue why such translations are difficult. We deal with these problems by specifying the mapping manually and proving faithfulness of the mapping. The other approach is similar to the work by Schoeller and Charles.

Conclusion. We presented a formal framework for faithfulness proofs based on theory interpretation. Proving faithfulness of model classes ensures consistency of model class specifications, prevents unexpected results from program verifiers, and also improves the overall quality of model class specifications.

Acknowledgments. We are grateful to Peter H. Schmitt for directing us to the concept of theory interpretation as the appropriate framework for our approach. We thank Reiner Hähnle, Gary T. Leavens, and the anonymous reviewers for helpful comments. This work was funded in part by the IST-2005-015905 MOBIUS project.

References

1. Charles, J.: Adding native specifications to JML. In: FTfJP (2006)
2. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Software: Practice and Experience 35(6), 583–599 (2005)
3. Darvas, Á.: Reasoning About Data Abstraction in Contract Languages. Ph.D. thesis, ETH Zurich (2009)

4. Darvas, Á., Müller, P.: Faithful mapping of model classes to mathematical structures. IET Software 2(6), 477–499 (December 2008)
5. Farmer, W.M.: Theory interpretation in simple type theory. In: Higher-Order Algebra, Logic, and Term Rewriting. LNCS, vol. 816, pp. 96–123 (1994)
6. Filliâtre, J.C., Hubert, T., Marché, C.: The Caduceus verification tool for C programs (2007), tutorial and Reference Manual
7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI. vol. 37, pp. 234–245. ACM Press (2002)
8. Guttag, J.V., Horning, J.J.: Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science, Springer (1993)
9. Hamilton, N., Nickson, R., Traynor, O., Utting, M.: Interpretation and instantiation of theories for reasoning about formal specifications. In: ACSC, Australian Computer Science Communications 19. pp. 37–45 (1997)
10. Jones, C.B.: Systematic software development using VDM. Prentice Hall (1986)
11. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Science of Computer Programming 55(1–3), 185–205 (2005)
12. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing 19(2), 159–189 (2007)
13. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems. pp. 175–188. Kluwer Academic Publishers (1999)
14. Levy, B.: An Approach to Compiler Correctness Using Interpretation Between Theories. Ph.D. thesis, University of California, Los Angeles (1986)
15. Maibaum, T.S.E., Veloso, P.A.S., Sadler, M.R.: A theory of abstract data types for program development: bridging the gap? In: TAPSOFT. pp. 214–230. Springer-Verlag (1985)
16. Marché, C.: Towards modular algebraic specifications for pointer programs: a case study. In: Rewriting, Computation and Proof. LNCS, vol. 4600, pp. 235–258. Springer (2007)
17. Meyer, B.: Eiffel: The Language. Prentice Hall (1992)
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Object Constraint Language, OMG Available Specification, Version 2.0, `http://www.omg.org/docs/formal/06-05-01.pdf`, 2006
20. Owre, S., Shankar, N.: A brief overview of PVS. In: TPHOLs. LNCS, vol. 5170, pp. 22–27. Springer (2008)
21. Schoeller, B., Widmer, T., Meyer, B.: Making specifications complete through models. In: Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 48–70. Springer (2006)
22. Shoenfield, J.R.: Mathematical Logic. Addison-Wesley (1967)
23. Srinivas, Y.V., Jüllig, R.: Specware: Formal support for composing software. In: Mathematics of Program Construction. pp. 399–422. Springer-Verlag (1995)
24. Turski, W.M., Maibaum, T.S.E.: The Specification of Computer Programs. Addison-Wesley (1987)