Reasoning About Method Calls in JML Specifications

Ádám Darvas and Peter Müller

ETH Zurich, Switzerland
{Adam.Darvas,Peter.Mueller}@inf.ethz.ch

Abstract. The Java Modeling Language, JML, is an interface specification language that uses side-effect free Java expressions to describe program behavior. In particular, JML specifications can contain calls to side-effect free methods.
To verify programs w.r.t. JML specifications, JML expressions have to be encoded in a program logic. This encoding is non-trivial for method calls. In this paper, we illustrate several subtle problems any program verifier for JML has to address. We present an encoding of method calls that handles abrupt termination, allows methods to create and initialize objects, and is sound, even if the JML specification is not satisfiable.

1 Introduction

The Java Modeling Language, JML [10], is a behavioral interface specification language that is supported by a variety of verification tools [3]. The syntax of the JML expressions in invariants, pre- and postconditions, etc. is an extension of the side-effect free expressions of Java. For verification purposes, these expressions must be encoded in the underlying logic of a program verifier, in our case multi-sorted first order logic with recursive datatypes.

To handle method calls in specifications, side-effect free methods can be encoded as mathematical functions and axioms that specify these functions [4]. The axioms reflect the behavior of the methods expressed by their interface specifications. Generating these axioms is difficult for the following reasons: (1) According to the JML semantics, a method call that terminates abruptly yields an arbitrary value of the method's return type. Therefore, the axiom for the corresponding function must not specify a value for cases in which the method terminates abruptly, but leave the value unspecified. (2) JML requires methods that are called in specifications to be *pure*. Pure methods are not completely side-effect free. They are allowed to allocate and initialize new objects, but must not modify existing objects. We call this notion of purity *weak purity*. It is important to model weak purity explicitly in the logic because object allocation and initialization may have an effect on whether an assertion holds or not. (3) The axiomatization of the functions has to be consistent, even if the JML specification is not implementable. Inconsistent axioms lead to unsound reasoning.

In this paper, we illustrate these problems by examples and present a formalization of pure methods that addresses all of them. This formalization is still work in progress. In particular, we make a few simplifying assumptions we want to alleviate in the future. Nevertheless, our results are interesting to researchers working on program verifiers for JML and on JML-style specification languages. We plan to implement our approach in the JIVE tool [12], but the main ideas can be used in any program verifier.

The paper is structured as follows. Sec. 2 and 3 sketch the logical background used in the rest of the paper. The following three sections explain the three main problems described above as well as our solutions. Sec. 7 and 8 discuss related and future work, respectively.

2 Background on program logic

We use Poetzsch-Heffter and Müller's program logic [14] to formalize our encoding of methods. This logic uses an explicit object store, which is necessary to model the store changes performed by pure methods. In this section we describe those parts of the logic that are needed in the rest of the paper.

For brevity, we present our results for a small subset of Java and JML. In particular, we omit interfaces, arrays, and static class members. An extension to arrays and static class members is straightforward. Interfaces require additional proof obligations.

Types and Values. Java's types and values are modeled by the sorts *Type* and *Value*, respectively. Sort *Type* contains primitive types, the type of the `null` reference, and class types. The reflexive, transitive subclass relation is denoted by $\preceq$. A *Value* is a value of a primitive type, the `null` reference, or a reference to an object. The function $\mathit{typeof} : \mathit{Value} \rightarrow \mathit{Type}$ yields the type of a value.

Object States. Object states are modeled via *locations* (instance variables). For each field of its class, an object has a location. The sort *FieldId* is the sort of unique field identifiers of a program. The function $\mathit{loc}(X, f)$ yields the location of the object referenced by X for field f, or *undefined* if the object does not have a location for f. Conversely, $\mathit{obj}(L)$ yields a reference to the object a location L belongs to. For brevity, we write $X.f$ for $\mathit{loc}(X, f)$ in the following.

$$\begin{aligned}&\mathit{loc} : \mathit{Value} \times \mathit{FieldId} \;\rightarrow\; \mathit{Location}\\&\mathit{obj} : \mathit{Location} \;\rightarrow\; \mathit{Value}\end{aligned}$$

Object Stores. Object stores are modeled by an abstract data type with main sort *Store* and operations to read and update locations, to create new objects, and to test whether an object is allocated. Poetzsch-Heffter and Müller present these functions and their axiomatization [14].

In this paper, we need only two store operations: $OS(L)$ denotes the value held by location L in store OS. $\mathit{alive}(X, OS)$ yields true if and only if object X is allocated in OS. Values of primitive types are allocated in all stores.

$$\begin{aligned}&\_(\_) \;: \mathit{Store} \times \mathit{Location} \;\rightarrow\; \mathit{Value}\\&\mathit{alive} : \mathit{Value} \times \mathit{Store} \;\rightarrow\; \mathit{Bool}\end{aligned}$$

The constant symbol \$ of sort *Store* is used to refer to the current object store in formulas. The current object store \$ can be considered as a global variable.

3 Encoding of methods and method calls

In this section, we describe the encoding of methods and method calls by functions and function applications, respectively. The axiomatization of these functions is explained in the following sections.

Encoding of methods. As mentioned in the introduction, pure methods can be modeled as mathematical functions. These functions take one argument for each parameter of the method and the object store in which they are evaluated, and yield the result of the method. For instance, a method m with one implicit parameter (the receiver) and one explicit parameter is modeled by the following function:

$$\hat{m} : \mathit{Value} \times \mathit{Value} \times \mathit{Store} \rightarrow \mathit{Value}$$

JML does not specify whether the meaning of a call to a method m in a specification is determined by m's specification or its implementation. For our work on static verification, $\hat{m}$ is axiomatized based on m's specification because (1) m may be abstract, that is, have no implementation, and (2) the meaning of an interface specification should be defined independently of a concrete implementation.

Encoding of method calls. To transform JML expressions into formulas of our logic, we introduce an expression transformer, γ, which takes a JML expression and a term denoting a store, and yields a formula. The store argument is necessary to handle weak purity.

The function γ is defined inductively over the syntax of JML expressions. For brevity, we describe γ by examples instead of giving a full definition. For instance, for a field `f`, $\gamma(\texttt{f==5}, OS)$ yields "$OS(\texttt{this.f}) = 5$".

The expression transformer γ maps method calls to applications of the corresponding functions. For instance, if `x` and `p` are local variables, the call `x.m(p)` in the current object store is transformed to "$\hat{m}(\texttt{x}, \texttt{p}, \$)$".

4 Normal and abrupt termination

The axiomatization of the functions for pure methods has to take into account abrupt termination. JML's semantics for abrupt termination considers a JML expression e to yield an arbitrary value of e's static type if e terminates abruptly [8]. For instance, the expression `5/0` yields an arbitrary integer.

The same semantics is used for method calls. That is, in a JML specification, a method call that terminates abruptly is considered to yield an arbitrary value of the method's result type. To reflect this semantics, specification cases of a method m that permit abrupt termination must not introduce axioms for $\hat{m}$, that is, have to leave the definition of $\hat{m}$ unspecified for these cases. Consider for instance the following specification of a method `abrupt`:

```
/*@ behavior
  @   ensures \result == 5;
  @*/
/*@ pure @*/ int abrupt() { /* ... */ }
```

JML's default for an omitted signals clause in a behavior specification case allows the method to throw any exception. That is, even the following code is a correct implementation of `abrupt`'s specification:

```
/*@ pure @*/ int abrupt() { throw new RuntimeException(); }
```

This implementation throws an exception for all calls to `abrupt`. Therefore, any call to `abrupt` in a specification should yield an arbitrary value. Concluding from `abrupt`'s specification that the function $\widehat{abrupt}$ constantly yields 5 would not be faithful to the JML semantics. Consequently, we generate axioms only for those specification cases that forbid abrupt termination. In this paper, we use only normal behavior specification cases, but our approach can be extended to all specification cases that contain the signals clause `false`.

5 Weak purity

According to the design of JML, pure methods are weakly-pure. That is, they must not modify existing objects, but are allowed to allocate and modify new objects. This notion of purity is necessary to allow pure methods to return data structures such as tuples. For instance, many of JML's model classes create and return objects of basic datatypes such as sets and sequences. In this section, we illustrate that weak purity can be observed in interface specifications and show how it can be modeled in a program logic.

Examples. In the following, we present two examples illustrating that weak purity has to be modeled in the formalization of methods and constructors to be faithful to JML's semantics and to avoid unsoundness.

Class `Alloc` in Fig. 1 declares a weakly-pure method, `alloc`, which is used in the specification of method `foo`. If an encoding of methods assumed that methods are completely side-effect free then `foo`'s ensures clause `alloc()==alloc()` would translate to $\widehat{alloc}(\mathtt{this}, \$) = \widehat{alloc}(\mathtt{this}, \$)$, which is trivially true. However, according to the JML semantics, `foo`'s ensures clause does not hold because `alloc` is specified to return a fresh object. This shows that the store changes performed by weakly-pure methods have to be encoded explicitly.

The example in Fig. 2 shows that neglecting weak purity in the formalization can lead to unsoundness. Class `Invariant` has a field `f` and an invariant that requires `f` to be non-zero. `Invariant`'s constructor is declared as helper, which allows it to return an object that does not satisfy its invariant. In fact, the `f`

```
class Alloc {

  /*@ pure @*/ Alloc()
     { /* ... */ }

  /*@ normal_behavior
    @   ensures \fresh(\result);
    @*/
  /*@ pure @*/ Alloc alloc() {
    return new Alloc();
  }
   /*@ normal_behavior
     @   assignable \nothing;
     @   ensures
     @       alloc () == alloc ();
     @*/
  void foo () { /* ... */ }
}
```

Fig. 1. The weakly-pure method `alloc` returns a fresh object.

```
class Invariant {
  int f;
  /*@ invariant f != 0; @*/

  /*@ private normal_behavior
    @   assignable f;
    @   ensures this.f == 0;
    @*/
  /*@ pure helper @*/ private Invariant()
    { f = 0; }

  /*@ private normal_behavior
    @   requires v == (new Invariant()).f;
    @   assignable \nothing;
    @*/
  int divide(int v) { return 5 / v; }

  int showIt() { return divide(0); }
}
```

Fig. 2. The weakly-pure constructor does not establish the invariant of the new object.

field of the new object is initialized to zero, as stated in the constructor's ensures clause. The constructor is pure since it modifies only the new object.

The constructor is called in the requires clause of method `divide`. According to the JML semantics, one can assume that all objects satisfy their invariants in the prestate of `divide`. If a formalization neglects the side-effects of a weakly-pure constructor, then one can conclude that after the constructor call still all object invariants hold (since the store is assumed to be unchanged) and, therefore, `(new Invariant()).f` evaluates to a non-zero value. By this reasoning, one can conclude that `v` is different from zero, which allows one to verify that `divide` does not terminate abruptly.

On the other hand, one can prove that the requires clause of the call `divide(0)` in method `showIt` is satisfied because, by the ensures clause of the constructor, `(new Invariant()).f` evaluates to zero. Therefore, method `showIt` verifies although it leads to a runtime exception. This unsoundness can be avoided by modeling weak purity in the encoding of methods.

Modeling store changes. To avoid the problems illustrated above, we make the potential store changes by pure methods explicit. For each pure method m, we introduce a function $\hat{mS}$ that takes the same arguments as $\hat{m}$ and yields the store after calling m. If m has one explicit parameter, $\hat{mS}$ has the signature:

$$\hat{mS} : \mathit{Value} \times \mathit{Value} \times \mathit{Store} \rightarrow \mathit{Store}$$

In the following, we call these functions *store functions.*

A pure method is guaranteed not to modify existing objects. We say that store OS' is a *pure successor* of store OS if all objects allocated in OS are still allocated and unchanged in OS'. We express this property by the predicate $psucc(OS, OS')$, where $psucc$ is defined as follows:

$$\begin{array}{rcl} psucc(OS, OS') & \equiv & (\forall X \bullet\ alive(X, OS) \ \Rightarrow\ alive(X, OS')\ ) \wedge \\ & & (\forall L \bullet\ alive(obj(L), OS) \ \Rightarrow\ OS(L) = OS'(L)\ ) \end{array}$$

For a pure method m, $\hat{mS}(t, p, OS)$ is a pure successor of OS.

Expression transformer. The expression transformer γ is defined such that each subexpression refers to the store resulting from the evaluation of the previous subexpression. In particular, after each method call, γ uses the corresponding store function for the transformation of the following subexpression.

In our first example, the transformation of the ensures clause of method `foo`, $\gamma(\texttt{alloc()==alloc()}, \$)$, yields "$\widehat{alloc}(\texttt{this}, \$) = \widehat{alloc}(\texttt{this}, \widehat{allocS}(\texttt{this}, \$))$". Whether this equality holds depends on the specification of method `alloc`. In our example, the specification of `alloc` implies that the equality does not hold. This reflects the JML semantics and also the behavior of JML's runtime assertion checker.

Analogously, the transformation of `divide`'s requires clause in our second example uses the store function for `Invariant`'s constructor, $\widehat{InvariantS}$: the field `f` is read in store $\widehat{InvariantS}(\texttt{this}, \$)$ rather than \$. The fact that all object invariants hold in the prestate of `divide`, \$, does not imply that the invariant of the new object holds in $\widehat{InvariantS}(\texttt{this}, \$)$. This prevents the soundness problem described above.

Modeling store changes is necessary to handle weak purity, but makes formulas more complicated. For many common expressions, the evaluation of one subexpression cannot observe store changes made by preceding subexpressions. For instance, the evaluation of `e2` in the conjunction `e1 && e2` cannot observe the store changes made by `e1`.[1] In such cases, it is possible to omit the store functions and transform the expression as if `e1` was strongly-pure, that is, $\gamma(\texttt{e1 \&\& e2}, OS) = \gamma(\texttt{e1}, OS) \wedge \gamma(\texttt{e2}, OS)$. We proved an equivalence result for this optimization for a small, but representative subset of Java and JML.

Summary. The axiomatization of the functions $\hat{m}$ and $\hat{mS}$ for a method m is based on m's normal behavior specification cases and weak purity.

Let m be a method declared in class C that takes one explicit parameter, p. $PRE^{C}_{m,i}(\texttt{this}, p, \$)$ denotes the (γ-converted) conjunction of the requires clauses of the i-th normal behavior specification case for m in C and, unless m is a helper method, the predicate $INV(\$)$ that expresses that all allocated objects satisfy their invariants in the current object store, \$. Analogously, $POST^{C}_{m,i}(\texttt{this}, p, \texttt{\textbackslash result}, \$)$ denotes the (γ-converted) conjunction of the ensures clauses and, unless m is a helper method, $INV(\$)$. The index i ranges

[1] unless `e2` contains a quantification over all allocated objects, even over those allocated by the evaluation of `e1`. JML does not precisely define the range of quantifiers.

over all normal behavior specification cases for m in C, including specifications inherited from C's superclasses, which ensures behavioral subtyping [5].

In the axiomatization of $\hat{m}$ and $\hat{mS}$ we express that if the arguments of a call to method m satisfy m's precondition then the result value satisfies the postcondition. Moreover, we know that the result value is alive in the poststate and that the poststate is a pure successor of the prestate. These properties are formalized by the predicate

$$\mathit{spec}^C_m(\ t,\ p,\ OS,\ \hat{m}(t,p,OS),\ \hat{mS}(t,p,OS)\ )$$

where t, p, and OS are the receiver, the explicit parameter, and the object store of the prestate of the call to m. The predicate spec^C_m is defined as follows:

$$\mathit{spec}^C_m(t,p,OS,r,OS') \ \equiv$$
$$\bigwedge_i \ (PRE^C_{m,i}(t,p,OS) \ \Rightarrow\ POST^C_{m,i}(t,p,r,OS') \ \wedge\ \mathit{alive}(r,OS') \ \wedge\ \mathit{psucc}(OS,OS'))$$

Although spec^C_m expresses the essential properties of the functions $\hat{m}$ and $\hat{mS}$, we do **not** simply state

$$(\forall t,\ p,\ OS \bullet\ \mathit{spec}^C_m(t,p,OS,\hat{m}(t,p,OS),\hat{mS}(t,p,OS))\ )$$

as an axiom because such a naive axiomatization can easily lead to unsoundness, as we explain in the next section.

6 Consistent axiomatization

Our approach is based on an axiomatization of the functions $\hat{m}$ and $\hat{mS}$ for each pure method m. Soundness requires that this axiomatization is consistent, that is, that there are functions that satisfy the axioms. Inconsistencies occur when one can derive *false* from a single axiom or when several axioms contradict each other. In this section, we discuss a JML example for both cases and show how inconsistencies can be avoided. We present our axiomatization and prove soundness.

6.1 Single axioms

With the naive axiomatization described above, method `wrong` in Fig. 3 leads to the following axiom (we omit the conjunct $INV(\$)$ for simplicity):

$$(\forall t,\ OS \bullet\ \widehat{wrong}(t,OS) = 0 \ \wedge\ \widehat{wrong}(t,OS) = 1 \ \wedge$$
$$\mathit{alive}(\widehat{wrong}(t,OS), \widehat{wrongS}(t,OS)) \ \wedge\ \mathit{psucc}(OS, \widehat{wrongS}(t,OS))\ )$$

Since `wrong`'s specification is not satisfiable, the axiom is equivalent to *false* (as $\widehat{wrong}(t,OS) = 0 = 1$). The axiom for a method m is part of the background theory used to verify methods that use m in their specification. If this background theory is inconsistent, the reasoning is potentially unsound. For instance, the

```
abstract class Inconsistent {

  /*@ normal_behavior
    @   ensures \result == 0;
    @   ensures \result == 1;
    @*/
  /*@ pure @*/ abstract int wrong();

  /*@ normal_behavior
    @   assignable \nothing;
    @   ensures \result == 6 + wrong();
    @   ensures \result == 5 + wrong();
    @*/
  int bar() { return 6; }
}
```

Fig. 3. The specification of `wrong` is not satisfiable.

```
class Cycle {

  /*@ normal_behavior
    @   ensures
    @      \result == direct() +1;
    @*/
  /*@ pure @*/ int direct() {
    return 5;
  }
}
```

Fig. 4. The recursive specification is not satisfiable by a pure method.

above axiom is part of the background theory used to verify method `bar` and allows one to verify `bar`, although its specification is obviously not satisfiable. Note that this unsoundness occurs even though `wrong` is not called from `bar`'s implementation.

To eliminate this source of unsoundness, we use axioms that are weaker than the naive axiomatization. These axioms require one to prove that the specification of a pure method m is satisfiable in order to assume the properties of $\hat{m}$ and $\hat{mS}$. One can assume that all arguments of these functions are allocated and that the receiver object of the call is a non-null instance of the enclosing class. This leads to the following axiom for a method m declared in class C with one explicit parameter:

$$\begin{aligned}(\forall t,\ p,\ OS \bullet\ & \mathit{alive}(t,OS)\ \wedge\ \mathit{alive}(p,OS)\ \wedge\ t \neq \mathit{null}\ \wedge\ \mathit{typeof}(t) \preceq\ C\ \wedge \\ & (\,\exists\, r,\ OS' \bullet\ \mathit{spec}^C_m(t,p,OS,r,OS')\,) \\ & \Rightarrow \\ & \mathit{spec}^C_m(t,p,OS,\hat{m}(t,p,OS),\hat{mS}(t,p,OS))\,)\end{aligned}$$

For method `wrong`, this axiom is void since there is no r that satisfies $r = 0\ \wedge\ r = 1$. Therefore, the left-hand side of the implication does not hold, and the overall implication is trivially true.

6.2 Recursive specifications

Recursive specifications occur when a method is either directly or indirectly specified in terms of itself. Class `Cycle` in Fig. 4 shows an example where a recursive specification leads to unsoundness.

Method `direct` is specified in terms of itself. It leads to the following (slightly simplified) axiom. To make the unsoundness more noticeable, we use the optimized expression transformer γ described in Sec. 5: since method `direct` returns an integer, potential store changes cannot be observed by subsequent calls. Therefore, we can use OS instead of $\widehat{directS}(t, OS)$.

$$
\begin{aligned}
(\forall t,\ OS \bullet\ & \mathit{alive}(t, OS)\ \wedge\ t \neq \mathit{null}\ \wedge\ \mathit{typeof}(t) \preceq \texttt{Cycle} \wedge \\
& (\exists r, OS' \bullet\ r = \widehat{\mathit{direct}}(t, OS') + 1 \wedge \mathit{alive}(r, OS')\ \wedge\ \mathit{psucc}(OS, OS')\ ) \\
& \Rightarrow \\
& \widehat{\mathit{direct}}(t, OS) = \widehat{\mathit{direct}}(t, OS) + 1 \wedge \ldots\ )
\end{aligned}
$$

By choosing $r = \widehat{\mathit{direct}}(t, OS') + 1$ and $OS' = OS$, the existentially quantified formula holds because values of primitive types are always allocated and *psucc* is reflexive. Therefore, the axiom allows one to prove $\widehat{\mathit{direct}}(t, OS) = \widehat{\mathit{direct}}(t, OS) + 1$ and, thereby, *false*.

To prevent this soundness problem, we take a rather drastic approach here: we completely forbid recursive specifications. That is, we define a *depends graph* and require it to be acyclic. The graph is defined as follows: (1) The nodes of the graph are the pure methods of a program. (2) There is an edge from node m to node n if a normal behavior specification case of method m mentions method n. (3) There is an edge from node m to node n if m is not a helper method and any invariant of the program mentions method n. This kind of edge is due to the fact that invariants are implicitly conjoined to the requires and ensures clauses of all non-helper methods.

While recursive specifications are often an indication that a specification is redundant or flawed, completely forbidding them is too restrictive. For instance, recursive methods such as a method that computes the factorial or the `equals` method on a recursive data structure are typically specified recursively. For our axiomatization, it is sufficient to require that such a recursive specification is well-founded.

As future work, we plan to investigate different means of guaranteeing well-foundedness, for instance, by using JML's measured_by clause to map method parameters to a partially-ordered well-founded set and enforcing that each recursive call uses parameters that are less according to this ordering. Such an order is easily defined for values of primitive types. For objects, we plan to use the partial order defined by the Universe type system (the ownership relation) [6].

6.3 Soundness

For programs that do not contain recursive specifications, our axiomatization of pure methods is consistent.

Theorem 1 (Consistency). *Let* $\boldsymbol{P}$ *be a specified program with an acyclic depends graph. There is a model for the axioms generated from the specifications of* $\boldsymbol{P}$*'s pure methods.*

Proof sketch. The proof runs by induction on the depth of a method m in the depends graph. The induction hypothesis is that there are well-defined functions $\hat{m}$ and $\hat{mS}$ for each method m with a depth up to N. These functions satisfy the axioms for all methods with a depth up to N. The induction base ($N = 0$, that is, the leaves of the graph) and induction step are proved by the same arguments, which we present in the following.

Let m be a pure method with depth N that takes one explicit parameter and is declared in class C, and consider any non-null C object t, value p, and store OS. We show that there are values r, OS' for $\hat{m}(t, p, OS)$ and $\hat{mS}(t, p, OS)$ that satisfy the axioms for m. The axioms for another method n with depth less or equal N do not mention $\hat{m}$ and $\hat{mS}$ because n is not specified in terms of m. Therefore, these axioms are satisfied independently of the definition of $\hat{m}$ and $\hat{mS}$. For the axioms for m, we continue as follows.

The axiom for each subclass S of C that is not a superclass of t's dynamic type holds trivially because $\mathit{typeof}(t) \preceq S$ does not hold. Since we do not consider interfaces, we can assume single subtyping. Therefore, it remains to show that there are values that satisfy the axioms for the superclasses of t's dynamic type.

Let σ be the set of all superclasses D of t's dynamic type ($\mathit{typeof}(t) \preceq D \preceq C$) such that $(\,\exists\, r, OS' \bullet\ \mathit{spec}_m^D(t, p, OS, r, OS')\,)$ holds. For superclasses of t's dynamic type that are not in σ, the axiom is trivially satisfied. In particular, if σ is empty, the proof is completed.

If σ is not empty, let T be the smallest class in σ w.r.t. the subclass relation. We define $\hat{m}(t, p, OS)$ and $\hat{mS}(t, p, OS)$ to yield any values r and OS' such that $\mathit{spec}_m^T(t, p, OS, r, OS')$ holds. Since T is in σ, such values exist.

It remains to show that these values satisfy the axioms for all classes in σ. Since T is the smallest class in σ, each member S of σ is a superclass of T. Specification inheritance guarantees that the subclass specification is stronger than the superclass specification: $\mathit{spec}_m^T \Rightarrow \mathit{spec}_m^S$. Therefore, r and OS' also satisfy $\mathit{spec}_m^S(t, p, OS, r, OS')$. $\square$

Note that this soundness proof assumes single subtyping. With multiple subtyping (that is, interfaces), there can exist superinterfaces of $\mathit{typeof}(t)$ that are neither sub- nor supertypes of the class T. The axioms for these interfaces can lead to inconsistencies. This source of unsoundness can be avoided by appropriate proof obligations, which are beyond the scope of this paper.

7 Related work

According to our knowledge, our work is the first encoding of methods that addresses abrupt termination, weak purity, and consistency.

The work closest to ours is that of Cok's [4], which also uses axiomatized functions to model pure methods. However, his formalization does not handle weak purity and does not prevent inconsistent axiomatizations and, therefore, unsoundness. For specification cases other than normal behavior, Cok uses signals clauses to generate axioms, which leads to a stronger axiomatization than ours, but in general requires strong purity for soundness. Cok's approach has been implemented in ESC/Java2 [9] and Boogie [1]. We have verified all unsound examples presented in this paper with ESC/Java2. ESC/Java [7] does not permit method calls in specifications.

Breunesse and Poll [2] address the consistency problem for model fields, which are similar to parameterless methods. They propose two solutions. Like ours, their first solution uses existential quantification to ensure that the representation relation of a model field is satisfiable. However, their encoding yields *false* for every JML expression e that contains a model field whose representation cannot be satisfied, even if e is a tautology. The second solution transforms model fields into pure methods. This solution requires a sound encoding for methods, which we presented in this paper. Breunesse and Poll do not address weak purity and recursive specifications.

For pure methods with restricted ensures clauses, the Krakatoa tool [11] generates function definitions rather than an axiomatization. Marché *et al.* do not discuss the requirements that are necessary to ensure that these functions are well-defined. They do not consider weak purity either.

Naumann [13] proposes a notion of purity that is more liberal than JML's weak purity and allows certain modifications of existing objects, for instance, updates of encapsulated caches. Extending our approach to this notion of purity seems possible.

8 Conclusion and future work

In this paper, we presented a formalization of pure methods as mathematical functions. This formalization allows one to reason about method calls in JML specifications. It handles abrupt termination and weak purity. The axiomatization of the functions is consistent, even if the JML specification is not satisfiable.

As future work, we plan to investigate ways to guarantee that recursive specifications of pure methods are well-founded, for instance, by using ownership. This will allow us to use more liberal rules for recursive specifications and, thereby, to handle more specifications. We also plan to implement our approach in Jive and to assess its practicality in case studies.

Acknowledgments. We are grateful to David Cok, Gary Leavens, Rustan Leino, Farhad Mehta, Nicole Rauch, Burkhart Wolff, and the reviewers for helpful comments and discussions.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005.

2. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs*, pages 51–60, 2003. Technical Report 408, ETH Zurich.
3. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80, pages 73–89. Elsevier, 2003.
4. D. Cok. Reasoning with specifications containing method calls in JML and first-order provers. In *Formal Techniques for Java Programs. Proceedings of the ECOOP'2004 Workshop*, 2004. An extended version of this paper will appear in the Journal of Object Technology (JOT).
5. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *International Conference on Software Engineering (ICSE)*, pages 258–267. IEEE Computer Society Press, 1996.
6. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.
7. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, volume 37, pages 234–245, 2002.
8. D. Gries and F. B. Schneider. Avoiding the undefined by underspecification. In *Computer Science Today*, volume 1000 of *LNCS*, pages 366–373. Springer-Verlag, 1995.
9. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 108–128. Springer-Verlag, 2005.
10. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, 2005.
11. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
12. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from sct.inf.ethz.ch/publications, 2000.
13. D. Naumann. Observational purity and encapsulation. In M. Cerioli, editor, *Fundamental Aspects of Software Engineering (FASE)*, volume 3442 of *LNCS*. Springer-Verlag, 2005.
14. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.