JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 05, No. 5, Special Issue: ECOOP 2005 Workshop FTfJP, June 2006

Reasoning About Method Calls in Interface Specifications

Ádám Darvas and **Peter Müller**, ETH Zurich, Switzerland

Interface specifications in languages such as Eiffel, the Java Modeling Language (JML), and Spec# are based on side-effect free expressions of the programming language. In particular, such specifications contain calls to side-effect free methods to describe the program behavior without exposing implementation details. Reasoning about such specifications requires an encoding of programming language expressions in a program logic. This encoding is non-trivial for method calls.
In this paper, we illustrate the subtle problems any encoding of method calls in specifications has to address. We present a sound encoding that allows side-effect free methods to create and initialize objects by explicitly modeling such modifications of the heap.

1 INTRODUCTION

Interface specifications in languages such as Eiffel [17], the Java Modeling Language (JML) [12], and Spec# [1] use expressions of the programming language in pre- and postconditions, object invariants, and other assertions. In particular, specifications are allowed to contain method calls, which are useful for two major purposes. First, methods are a means of abstraction. They allow one to express properties of a data structure without exposing its implementation details. Second, methods are a unit of reuse. Using methods in specifications avoids the duplication of specifications. For instance, using a square root method in specifications avoids repeating the properties of square root wherever they are needed.

Expressions in specifications must be side-effect free to prevent the evaluation of specifications from interfering with the program execution. For method calls, the absence of side-effects is enforced by requiring methods called in specifications to be pure. In Eiffel, JML, and Spec#, a method is considered *pure* if its execution does not modify objects that are allocated in the prestate of the method—additionally, JML requires pure methods to be deterministic. However, pure methods are allowed to allocate and initialize new objects. We call this notion of purity *weak purity*.

Allowing pure methods to allocate and initialize new objects is important for expressiveness [20, 24]. Object-oriented languages represent almost all data as objects. Therefore, methods that return strings, tuples, sequences, sets, etc. often create objects. Moreover, methods often create and manipulate auxiliary objects, for instance, iterators. Such methods are weakly-pure because they do not mod-

Cite this article as follows: : *Reasoning About Method Calls in Interface Specifications*, in Journal of Object Technology, vol. 05, no. 5, Special Issue: ECOOP 2005 Workshop FTfJP, June 2006, pages 59–85,
http://www.jot.fm/issues/issues_2006_06/article3

ify objects that exist in the method prestate, but they are not entirely free from side-effects.

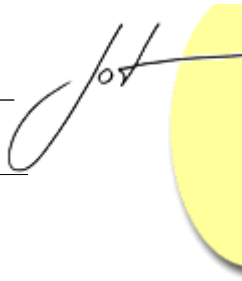
Class `Graph` in Fig. 1 illustrates how pure methods are used in JML. The class implements a graph with nodes and edges. Nodes are represented by natural numbers between 0 and the value of field `nodes`. An adjacency matrix, stored in field `amatrix`, defines the edges between the nodes. Since these fields give the internal representation of the graph, they are declared private. For brevity, we omit the invariant that expresses that the two-dimensional array is large enough for the number of nodes.

The methods `getNodes` and `getEdges` are used as abstraction functions [10] that describe a graph as a set of nodes and a set of edges. An edge between two nodes is represented by an instance of class `Pair`. The specifications of `getNodes` and `getEdges` are private because they are expressed in terms of private fields. Method `getEdges` is weakly-pure since it creates a new `Set` of `Pair`s. Like Java's interface specifies, the `equals` method of `Set` implements deep comparison. This makes `getEdges` useful in specifications. However, unlike Java's interface, we assume that method `add` of `Set` returns a new `Set`, that is, is a pure method.

Method `addEdge` is specified in terms of the pure methods. Thus, the specification can be public as it does not expose private implementation details. Furthermore, the use of the weakly-pure `getEdges` significantly shortens the specification. Note that the ensures clause contains seven calls of pure methods or constructors, four of which allocate and return new objects.

For verification purposes, the expressions of the specification language must be encoded in a program logic. 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 as expressed by their interface specifications. Generating these axioms is difficult for the following reasons. First, weak purity must be modeled explicitly in the logic because object allocation and initialization may have an effect on whether an assertion holds or not. For instance the expression `m()==m()` is not trivially true if `m` is a weakly-pure method because `m` might create and return a new object. Second, the axiomatization of the functions has to be consistent to avoid unsound reasoning. In particular, unsatisfiable specifications of pure methods must not lead to inconsistencies.

In this paper, we reveal the subtle problems in the encoding of pure methods and illustrate them by examples. We present an encoding of pure methods that handles weak purity by explicitly modeling the changes of the object store made by pure methods. Additional proof obligations guarantee that the encoding functions of pure methods are well-defined, that is, that their axiomatization is consistent. Our encoding is interesting for designers of Eiffel-style interface specification languages and for researchers working on program verifiers for programs with Eiffel-style specifications (for instance, Boogie [1], ESC/Java [11, 8], Krakatoa [16], or Jive). For concreteness, we present our work in terms of a subset of sequential Java and JML.



```
class Graph {

  private int nodes;
  private boolean[][] amatrix;

  /*@ private normal_behavior
    @   ensures \result == nodes;
    @*/
  /*@ pure @*/ public int getNodes() { return nodes; }

  /*@ private normal_behavior
    @   ensures (\forall int x,y;
    @              0 <= x && x <= nodes && 0 <= y && y <= nodes;
    @              \result.contains(new Pair(x,y)) == amatrix[x][y]);
    @*/
  /*@ pure @*/ public Set getEdges() { /*...*/ }

  /*@ public normal_behavior
    @   requires 0 <= x && x <= getNodes()
    @         && 0 <= y && y <= getNodes();
    @   ensures  getEdges().equals(\old(getEdges().add(new Pair(x,y))))
    @        &&  getNodes() == \old(getNodes());
    @*/
  void public addEdge(int x, int y) { amatrix[x][y] = true; }

  // invariant, constructors, and other methods omitted.
}
```

Figure 1: A JML specification using pure methods. Annotation comments start with an at-sign (@), and at-signs at the beginning of lines are ignored. We omit frame axioms (assignable clauses) since they are not relevant for this paper.

In particular, we omit interfaces, arrays, and static class members. Our results can be extended to full Java and are also applicable to Eiffel and Spec#.

Besides pure methods, JML provides model fields [3] to express data abstraction. In a program logic, the treatment of model fields is similar to parameterless pure methods [19]. The value of a model field is determined by applying a mapping to the concrete states of objects. For the set of edges in our `Graph` example, this mapping would yield a new `Set` object, just like method `getEdges`. That is, model field accesses are weakly-pure and lead to the same problems as weakly-pure methods. In the rest of the paper, we focus on methods, but our results are also applicable to model fields.

The paper is structured as follows. Sec. 2 sketches the logical background used in the rest of the paper. We illustrate the problems of weak purity and describe our solution in Sec. 3. In Sec. 4, we formalize our encoding of specification expressions. The axiomatization of the functions for pure methods and a soundness result are presented in Sec. 5. In the remaining sections, we explain the benefit of value types for the treatment of pure methods, discuss related work, and offer conclusions.

2 PRELIMINARIES

In this section, we describe the model of the object store that will be used to formalize weak purity and we present the foundations of our encoding of pure methods. Here and throughout the paper, we use the term "method" to refer to both methods and constructors. Pure methods and constructors are treated analogously by our encoding.

Store model

To formalize properties of the object store, we use the store model of Poetzsch-Heffter and Müller's program logic [23]. It is formalized in multi-sorted first order logic with recursive datatypes.

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 subtype 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 $typeof : Value \to Type$ 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 $loc(X, f)$ yields the location for field f of the object referenced by X, or *undefined* if the object does not have a location for f. Conversely, $obj(L)$ yields a reference to the object a location L belongs to. For brevity, we write $X.f$ for $loc(X, f)$ in the following.

$$\begin{aligned} loc &: Value \times FieldId \to Location \cup \{undefined\} \\ obj &: Location \to Value \end{aligned}$$

Since the properties of these functions are not needed in this paper, we refer the reader to [23] for their axiomatization.

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 [23].

In this paper, we need the following store operations: $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of class T. $new(OS, T)$ yields a reference to this object. $OS(L)$ denotes the value held by location L in store OS. $alive(X, OS)$ yields true if and only if object X is allocated in OS. Values of primitive types are allocated in all stores. The sort *ClassId* is the sort of unique class identifiers of a program.

$$\begin{array}{ll} \_\langle\_\rangle & : Store \times ClassId \rightarrow Store \\ new & : Store \times ClassId \rightarrow Value \\ \_(\_) & : Store \times Location \rightarrow Value \\ alive & : Value \times Store \rightarrow Bool \end{array}$$

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. The *prestore* of a method execution is the object store immediately after the arguments of the method are computed, but before the precondition is evaluated. Analogously, the *poststore* of a method execution is the object store after the execution of the return statement, but before the evaluation of the postcondition.

Encoding of pure methods

We encode pure methods by uninterpreted function symbols and axioms as described in the following.

Function symbols. Functions that model pure methods 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. (Note that a pure method used in specifications must have a return value.) For instance, a method m with one implicit parameter (the receiver) and one explicit parameter is modeled by the following function:

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

Axiomatization. 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.

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 [9]. 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. 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 exceptional postcondition `signals false`. We present the precise axiomatization in Sec. 5.

3 WEAK PURITY—PROBLEMS AND APPROACH

In this section, 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. We explain our approach to handling weak purity.

Examples

Class `Alloc` in Fig. 2 declares a weakly-pure method, `alloc`, which is used in the specification of method `foo`. If an encoding of methods assumed that pure methods are completely side-effect free then `foo`'s ensures clause `alloc()==alloc()` would translate to $\hat{alloc}(\texttt{this}, \$) = \hat{alloc}(\texttt{this}, \$)$, which is trivially true. However, according to the JML semantics, `foo`'s ensures clause does not hold because `alloc` returns a fresh object (as expressed by `alloc`'s ensures clause). In particular, the runtime assertion checker would evaluate `foo`'s ensures clause to *false*. This shows that an encoding of pure methods that is faithful to the JML semantics must encode the store changes performed by weakly-pure methods explicitly.

The second example (Fig. 3) shows that neglecting weak purity in the formalization can lead to unsoundness. Class `Unsound` has a field `f` and an invariant that requires `f` to be non-zero. `Unsound`'s constructor is declared as helper, which allows it to return an object that does not satisfy its invariant. In fact, the `f` 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 that are alive satisfy their invariants in the prestore of `divide`. If a formalization neglects the side-effects of a weakly-pure constructor, then one can conclude that after the constructor call still the invariants of all alive objects hold (since the store is assumed to be unchanged) and, therefore, `(new Unsound()).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.

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

  /*@ normal_behavior
    @   ensures \fresh(\result);
    @*/
  /*@ pure @*/ Alloc alloc() { return new Alloc(); }

  /*@ normal_behavior
    @   assignable \nothing;
    @   ensures    alloc() == alloc();
    @*/
  void foo() { /* ... */ }
}
```

Figure 2: The weakly-pure method `alloc` returns a fresh object.

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 Unsound()).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.

It is important to understand that the problems illustrated above cannot simply be fixed by disallowing test for reference equality in specifications. Such a requirement would rule out the specification of `foo` in Fig. 2, but would not solve the soundness problem illustrated by class `Unsound`. Reference equality is needed for many examples, for instance, collections of references. Using reference equality on objects that exist already before the evaluation of a specification does not cause any problem. It would, therefore, be overly restrictive to completely forbid it.

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:

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

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

A pure method is guaranteed not to modify existing objects. We say that store

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

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

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

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

Figure 3: The weakly-pure constructor does not establish the invariant of the new object.

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 $OS \trianglelefteq OS'$, and define it as follows:

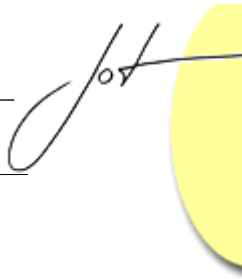
$$\begin{aligned} OS \trianglelefteq OS' \;\equiv\; & (\forall X \bullet \; \mathit{alive}(X, OS) \;\Rightarrow\; \mathit{alive}(X, OS')\;) \;\wedge \\ & (\forall L \bullet \; \mathit{alive}(\mathit{obj}(L), OS) \;\Rightarrow\; OS(L) = OS'(L)\;) \end{aligned}$$

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

Examples revisited

By using the store functions for pure methods, we can encode expressions such that each subexpression refers to the store resulting from the evaluation of the previous subexpression. In particular, after each method call, the corresponding store function is used in the encoding of the following subexpression.

In our first example, this encoding of the ensures clause of method `foo` yields "$\hat{alloc}(\texttt{this}, \$) = \hat{alloc}(\texttt{this}, \hat{allocS}(\texttt{this}, \$))$". Whether this equality holds depends



```
Expr  ::=  Expr && Expr
        |  Expr == Expr
        |  Expr + Expr
        |  Expr . FieldId
        |  Expr . MethId ( Expr )
        |  new ClassId( Expr )
        |  ParId | this | null | true | false | -1 | 0 | 1 | ...
        |  \old ( Expr )
        |  \result
```

Figure 4: Specification expression subset. *MethId* and *ParId* are the sorts for method and formal parameter names, respectively.

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 `Unsound`'s constructor, $\hat{UnsoundS}$: the field `f` is read in store $\hat{UnsoundS}(new(\$, Unsound), \$\langle Unsound\rangle)$ rather than \$, that is, in the store after allocating a new `Unsound` object in store \$ and executing the constructor on this object. The fact that all object invariants hold in the prestore of `divide`, \$, does not imply that the invariant of the new object holds in the modified store. This prevents the soundness problem described above.

4 ENCODING OF SPECIFICATION EXPRESSIONS

In this section, we formalize the encoding of expressions in two steps. In the first step, we describe the general encoding including the treatment of weak purity. In the second step, we show how the encoding can be simplified for expressions of primitive types.

We present the encoding for a small subset of JML that allows us to demonstrate the most interesting aspects of weak purity. The expression syntax is given in Fig. 4. The subset contains the operators "`&&`", "`==`", and "`+`", field reads, method calls, and object creation with constructor calls. Without loss of generality, we require that methods and constructors take exactly one explicit parameter. The subset also contains formal parameters (including `this`) as well as the literals `null`, `true`, `false`, and integer literals. We also support JML's `\old` and `\result` expressions.

General encoding

We formalize the encoding of expressions by two functions. The *expression transformer* γ translates specification expressions to terms of multi-sorted first order logic. It has the following signature:

$$\gamma : Expr \times Store \times Store \rightarrow Term$$

where *Expr* is the sort for specification expressions as defined in Fig. 4 and *Term* is the sort for first order logic terms. The first store argument of γ is the store in which the expression is evaluated. The second store argument is used to transform ensures clauses that contain `\old` expressions. It is used to pass the prestore of the method to γ.[1]

Since we make the store modifications of weakly-pure expressions explicit, the stores used to encode an expression may change from subexpression to subexpression. For instance, if the first subexpression is a call to a method m then the store for the second subexpression is obtained by using the store function $\hat{m}S$. This tracking of store changes is also necessary when encoding `\old` expressions. For instance, consider the encoding of the ensures clause `\old(alloc())==\old(alloc())`, where OS is used to refer to the prestore of the method. Analogously to the `Alloc` example in Fig. 2, this encoding must not yield "$\hat{alloc}(\texttt{this}, OS) = \hat{alloc}(\texttt{this}, OS)$". That is, the two `\old` expressions must not be encoded using the same store. Instead, the store function $\hat{allocS}$ has to be used for the encoding of the second subexpression.

To formalize store changes, we use a function ω that yields the stores after the evaluation of an expression. It takes the same parameters as γ and returns two stores referring to the (possibly modified) pre- and poststores after the evaluation of the expression:

$$\omega : Expr \times Store \times Store \rightarrow Store \times Store$$

The inductive definitions of γ and ω are presented in Fig. 5. Most cases of the definition are straightforward. We explain the most interesting cases in the following.

The function γ encodes a call to method m as a function application of $\hat{m}$. The successor store of a method call is expressed using the store function $\hat{m}S$ as explained earlier. The evaluation of `\old` expressions in the actual parameters of the call to m potentially modifies the method's prestore. The ω function for method calls changes the second store component of its result accordingly.

Constructor calls are encoded similarly to method calls. The receiver parameter of the function application is the newly created object and the store parameter is the store after the object allocation.

[1] Since γ works on syntactic entities, it would be more accurate to say that it takes two terms of sort *Store* rather than two stores. We do not make this distinction here for simplicity.

An expression `\old( E )` is encoded by applying γ recursively to expression E. To express that E is evaluated in the prestore of the method, the second store argument of γ is passed as first store argument to the recursive γ application. The ω function expresses that the store changes made by an `\old` expression effect only the prestore, that is, the second store of the resulting tuple.

Examples and discussion

To illustrate our encoding of pure methods, we revisit the `Graph` example from Fig. 1 and encode the most interesting assertion, namely the ensures clause of `addEdge`. We show the result of the following application of the expression transformer γ:

$$\gamma(\texttt{getEdges().equals(\old(getEdges().add(new Pair(x,y)))) \&\& getNodes() == \old(getNodes())}, OS', OS)$$

OS and OS' denote the stores before and after the execution of `addEdge`, respectively. In the presentation of the encoding, we use the following abbreviations:

$$\begin{array}{ll} G' & = \hat{getEdges}(this, OS') \\ G & = \hat{getEdges}(this, OS) \\ P & = \hat{Pair}(new(\hat{getEdgesS}(this, OS), Pair), x, y, \hat{getEdgesS}(this, OS)\langle Pair\rangle) \\ PS & = \hat{PairS}(new(\hat{getEdgesS}(this, OS), Pair), x, y, \hat{getEdgesS}(this, OS)\langle Pair\rangle) \end{array}$$

G' and G denote the results of the pure method `getEdges` executed in stores OS' and OS, respectively. P denotes the result of the `Pair` constructor. Since the call of this constructor occurs within an `\old` expression, it is evaluated in the store that is obtained by executing `getEdges` in `addEdge`'s prestore, OS, and after allocating a new `Pair` object. These changes of the store are reflected in the last argument of the application of the function for the pure constructor, $\hat{Pair}$. PS denotes the store after the execution of this constructor. Using these abbreviations, we get the following encoding of `addEdge`'s ensures clause:

$$\begin{array}{l} \hat{equals}(\ G', \hat{add}(G, P, PS), \hat{getEdgesS}(this, OS')\ ) \ \wedge \\ \hat{getNodes}(\ this, \hat{equalsS}(G', \hat{add}(G, P, PS), \hat{getEdgesS}(this, OS'))\ ) = \\ \hat{getNodes}(this, \hat{addS}(G, P, PS)) \end{array}$$

The `equals` method is executed in the poststore of the first call to `getEdges`. Analogously, `getNodes` is executed in the poststore of `equals`. Note that the second call to `getNodes` is executed in the poststore of `add` because this call occurs within an `\old` expression, and the preceding store change to the prestore was made by the execution of method `add`, which also occurs within `\old`.

This example shows that our encoding makes all changes of the object store explicit. This solves the problems described in Sec. 3. However, the example also reveals several problems of the encoding presented so far:

CONJUNCTION, EQUALITY, ADDITION

$$\gamma(E \texttt{ op } F, OS', OS) = \gamma(E, OS', OS) \;\; FOL(\texttt{op}) \;\; \gamma(F, \omega(E, OS', OS))$$
$$\omega(E \texttt{ op } F, OS', OS) = \omega(F, \omega(E, OS', OS))$$

where `op` is one of `&&`, `==`, or `+`, and $FOL(\texttt{\&\&}) = \text{“}\wedge\text{”}$, $FOL(\texttt{==}) = \text{“}=\text{”}$, $FOL(\texttt{+}) = \text{“}+\text{”}$.

FIELD READ

$$\gamma(E.\texttt{f}, OS', OS) = \omega_1(E, OS', OS)(\gamma(E, OS', OS).\texttt{f})$$
$$\omega(E.\texttt{f}, OS', OS) = \omega(E, OS', OS)$$

METHOD CALL

$$\gamma(E.\texttt{m}(F), OS', OS) = \hat{m}(\ \gamma(E, OS', OS),\ \gamma(F, \omega(E, OS', OS)),\ \omega_1(F, \omega(E, OS', OS))\ )$$
$$\omega(E.\texttt{m}(F), OS', OS) = (\ \hat{mS}(\ \gamma(E, OS', OS),\ \gamma(F, \omega(E, OS', OS)),\ \omega_1(F, \omega(E, OS', OS))\ ),\ \omega_2(F, \omega(E, OS', OS))\ )$$

OBJECT CREATION

$$\gamma(\texttt{new C}(E), OS', OS) = \hat{C}(\ new(\omega_1(E, OS', OS), \texttt{C}),\ \gamma(E, OS', OS),\ \omega_1(E, OS', OS)\langle\texttt{C}\rangle\ )$$
$$\omega(\texttt{new C}(E), OS', OS) = (\ \hat{CS}(\ new(\omega_1(E, OS', OS), \texttt{C}),\ \gamma(E, OS', OS),\ \omega_1(E, OS', OS)\langle\texttt{C}\rangle\ ),\ \omega_2(E, OS', OS)\ )$$

FORMAL PARAMETER

$$\gamma(\texttt{v}, OS', OS) = \texttt{v}$$
$$\omega(\texttt{v}, OS', OS) = (OS', OS)$$

CONSTANT

$$\gamma(\texttt{c}, OS', OS) = \texttt{c}$$
$$\omega(\texttt{c}, OS', OS) = (OS', OS)$$

where `c` can be one of `this`, `null`, integer literal or boolean constant.

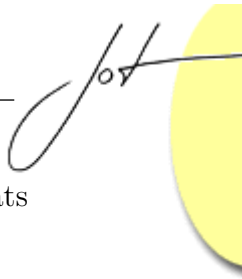
OLD EXPRESSION

$$\gamma(\texttt{\textbackslash old}(E), OS', OS) = \gamma(E, OS, OS)$$
$$\omega(\texttt{\textbackslash old}(E), OS', OS) = (OS', \omega_1(E, OS, OS))$$

RESULT EXPRESSION

$$\gamma(\texttt{\textbackslash result}, OS', OS) = resV$$
$$\omega(\texttt{\textbackslash result}, OS', OS) = (OS', OS)$$

Figure 5: Definition of functions γ and ω. We use ω_1 and ω_2 to denote the first and second store component of the result of ω, respectively. $resV$ is a special program variable to represent the result value of a method.



1. The application of a store function for each method or constructor call bloats the resulting formula, which makes it difficult to read and reason about.
2. Due to the store functions, commutativity of operators gets lost. For instance, the expressions `m()==n()` and `n()==m()` are encoded by different formulas.
3. The store functions make it difficult to match specifications. For instance, if clients of method `addEdge` are interested in the second conjunct of the ensures clause, but not in the first one, they still have to deal with the store changes potentially made by the first conjunct because the corresponding store functions appear in the encoding of the second conjunct.

We cannot avoid these complications in the most general setting such as the problem cases shown in Sec. 3. However, in many practical cases, the store changes made by weakly-pure methods are not observable by succeeding subexpressions and need, therefore, not be modeled explicitly. For instance, the second conjunct of `addEdge`'s ensures clause cannot refer to values of subexpressions of the first conjunct. Therefore, the value of the second conjunct is independent of the store changes potentially made by the first conjunct. In the next subsection, we use this observation to simplify the encoding of expressions.

Simplified encoding

The simplification we make is based on the observation that the evaluation of a subexpression E of a primitive type cannot create observable store changes for a successor subexpression F. In such cases, it is possible to transform F as if E was strongly-pure, that is, for the transformation of F we can omit the store functions for E and "roll back" to the original pre- and poststores.

In our small expression syntax, the simplification is applicable to conjunction and addition because their operands are of primitive types. The simplification can also be applied to equality in case the operands are of a primitive type. Furthermore, we do not have to track the store changes made by methods with primitive result types, because objects created by such methods cannot be referred to by subsequent subexpressions.

To formalize this simplification, we only have to adapt the definition of γ (Fig. 5) for binary operators and the definition of ω for calls to methods with primitive result types. For conjunction, addition, and equality on primitive types, the γ function need not carry over the store changes made by expression E when transforming the successor expression F (`op` and FOL are as in Fig. 5):

$$\gamma(E\ \texttt{op}\ F, OS', OS) = \gamma(E, OS', OS)\ FOL(\texttt{op})\ \gamma(F, OS', OS)$$

For calls to a method m with a primitive result type, the ω function need neither carry over the store changes made by the parameter expressions (E and F) nor by

the method itself. Neither of these store changes can be observed by subsequent expressions:

$$\omega(E.\texttt{m}(F), OS', OS) = (OS', OS)$$

For instance, if `m` is an integer method then the evaluation of `n()` in the expression `m(new Pair(x,y)) == n()` cannot refer to the new `Pair` object, but only to `m`'s result value.

Applying the simplified encoding to the ensures clause of method `addEdges` yields the following formula:

$$\hat{equals}(G', \hat{add}(G, P, PS), \hat{getEdgesS}(this, OS')) \wedge$$
$$\hat{getNodes}(this, OS') = \hat{getNodes}(this, OS)$$

The simplification addresses the complications discussed in the previous subsection:

1. The resulting encoding is significantly simpler than the general encoding, even for the ensures clause of `addEdges`, which contains nested method calls with non-primitive result types. The effect is even larger for simpler specifications.

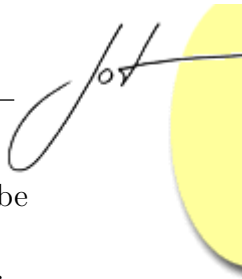
2. Since the simplified encoding omits the store functions when operators are applied to expressions of primitive types, commutativity is preserved in these cases. For instance, the expressions `m()==n()` and `n()==m()` are now encoded by equivalent formulas if `m` and `n` are boolean or integer methods.

3. Since the simplified encoding rolls back to the original stores at each boolean connective, matching of specifications is no longer a problem. In the `addEdge` example, the store changes of the first conjunct are ignored in the encoding of the second conjunct, which makes it simple to match the second conjunct with other assertions.

In order to see how frequently the simplified encoding actually applies in practice, we studied the Dutch Internet voting tally system that was specified and verified using JML and ESC/Java2 [11]. To our knowledge, that is the largest open-source system that has been specified with JML. We found that 42 per cent (77 out of 182) of all pure methods have primitive result type. This result shows that the simplified encoding makes a difference for practical examples.

5 AXIOMATIZATION

The expression transformer γ encodes method calls as function applications and models weak purity by store functions. In this section we present the axiomatization of the functions for pure methods. The axiomatization has to be done with care



because inconsistent axioms can lead to unsound reasoning. Consistency has to be preserved even if the specifications of pure methods are not satisfiable.

As pointed out in Sec. 2, the axiomatization of the functions $\hat{m}$ and $\hat{mS}$ for a method m is based on m's normal behavior specification cases. Let m be a method declared in class C that takes one explicit parameter, par. We define the predicate $PRE^C_{m,i}(OS)$ to denote the conjunction of (1) the requires clause of m's i-th normal behavior specification case, $req^C_{m,i}$, evaluated in store OS and (2) the predicate $INV(OS)$, which expresses that all allocated objects satisfy their invariants in store OS. Conjunct (1) is obtained by applying γ to $req^C_{m,i}$. The first store argument for the encoding of the precondition is OS. The second store argument is arbitrary (denoted by _) because preconditions must not contain `\old` expressions. Conjunct (2) is omitted if m is a helper method.

$$PRE^C_{m,i}(OS) \equiv \gamma(req^C_{m,i}, OS, \_) \ \wedge \ INV(OS)$$

Analogously, we define $POST^C_{m,i}(OS', OS)$ to denote the conjunction of (1) the ensures clause of m's i-th normal behavior specification case, $ens^C_{m,i}$, evaluated in prestore OS and poststore OS' and (2) the predicate $INV(OS')$ (unless m is a helper method):

$$POST^C_{m,i}(OS', OS) \equiv \gamma(ens^C_{m,i}, OS', OS) \ \wedge \ INV(OS')$$

The index i ranges 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 following properties hold:

1. The result value r of m satisfies m's postcondition.
2. The result value r of m is alive in m's poststore OS'.
3. The poststore OS' of m is a pure successor of the store that is obtained by evaluating the `\old` expressions of m's ensures clause in m's prestore OS. This intermediate store is obtained by $\omega_2(ens^C_{m,i}, \_, OS)$.

Property 1 describes the result of m. Property 2 is guaranteed by the semantics of Java. Property 3 is necessary to guarantee that the objects created during the evaluation of `\old` expressions are considered to be alive when the postcondition of m is evaluated. These objects are not alive in m's prestore. The corresponding property for objects created during the evaluation of the precondition is not needed because it is not possible to refer to these objects from the postcondition. That is, these store changes can safely be ignored. Note that the definition of ω guarantees that $\omega_2(ens^C_{m,i}, \_, OS)$ is a pure successor of OS and, since $\trianglelefteq$ is transitive, OS' is a

pure successor of OS. The three properties are formalized by the following predicate:

$$
\begin{aligned}
&spec^C_m(t, p, OS, r, OS') \;\equiv \\
&\bigwedge_i \left(PRE^C_{m,i}(OS)[t/\texttt{this}, p/par] \;\Rightarrow POST^C_{m,i}(OS', OS)[t/\texttt{this}, p/par, r/resV] \wedge \right. \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad alive(r, OS') \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \omega_2(ens^C_{m,i}, \_, OS) \trianglelefteq OS'
\end{aligned}
$$

where t, p, and OS are the receiver, the explicit parameter, and the prestore of the execution of m; r is the result value and OS' is the poststore of m. $e[x/y]$ denotes the term e with all free occurrences of y substituted by x.

The property we are ultimately interested in is the following:

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

By passing $\hat{m}(t, p, OS)$ for r, each occurrence of `\result` in m's ensures clause is replaced by $\hat{m}(t, p, OS)$. That is, $\hat{m}$ is constrained by the specification of m's result. Passing $\hat{mS}(t, p, OS)$ for OS' accounts for the potential store changes performed by the weakly-pure method m. This prevents one in particular from trivially proving possibly invalid assertions such as "alloc() == `\old`(alloc())" even if the pre- and poststore is identical.

However, we do **not** simply state the above property as an axiom because such a naive axiomatization can easily lead to unsoundness, as we explain in the following.

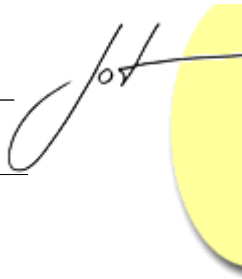
Consistency

Soundness requires that the axiomatization of functions $\hat{m}$ and $\hat{mS}$ is consistent, that is, that there are functions that satisfy the axioms. Inconsistencies occur when unsatisfiable specifications are turned into axioms in a naive way. An unsatisfiable specification of a pure method m is not necessarily detected when m's implementation is verified. Partial correctness logics allow one to verify m w.r.t. an unsatisfiable specification if m's implementation does not terminate. Moreover, if the inconsistent axiomatization of the functions for method m allows one to derive *false*, one can prove anything, even that m's implementation satisfies an unsatisfiable specification. In this subsection, we show how the inconsistencies can be avoided for non-recursive specifications. Recursion is addressed in the next subsection.

With the naive axiomatization described above, method `wrong` in Fig. 6 leads to the following axiom (we omit the conjunct $INV(OS)$ for simplicity):

$$
\begin{aligned}
(\,\forall\, t,\ OS \bullet\ & \hat{wrong}(t, OS) = 0 \ \wedge\ \hat{wrong}(t, OS) = 1 \wedge \\
& alive(\hat{wrong}(t, OS), \hat{wrongS}(t, OS)) \ \wedge\ OS \trianglelefteq \hat{wrongS}(t, OS)\,)
\end{aligned}
$$

Since `wrong`'s specification is not satisfiable, the axiom is equivalent to *false* (as $\hat{wrong}(t, OS) = 0 = 1$). The axiom for a method m is part of the background



```
abstract class Inconsistent {

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

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

Figure 6: The specification of `wrong` is not satisfiable.

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 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.

In practice, unsatisfiable specifications are far less obvious than in the example of method `wrong`, because they typically involve several normal behavior specification cases including inherited specifications. A verification technique has to ensure that unsatisfiable specifications do not lead to unsound reasoning.

To eliminate this source of unsoundness, we use axioms that are weaker than the naive axiomatization above. These axioms require one to prove, by giving a witness, that the specification of a pure method m is satisfiable in order to assume the properties of $\hat{m}$ and $\hat{mS}$. That is, the axioms for $\hat{m}$ and $\hat{mS}$ are guarded by the following antecedent:

$$(\exists r,\ OS' \bullet\ spec_m^C(t, p, OS, r, OS'))$$

The existence of a witness has to be proven in order to employ the corresponding axiom. For method `wrong`, one cannot give a witness r that satisfies $r = 0 \ \wedge\ r = 1$. Therefore, the antecedent of the corresponding axiom is *false*, and the axiom is void.

```
class Cycle {

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

Figure 7: The recursive specification is not satisfiable by a pure method.

Recursive specifications

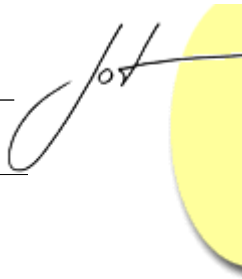
Recursive specifications occur when a method is either directly or indirectly specified in terms of itself. Class `Cycle` in Fig. 7 shows an example where a recursive specification leads to an inconsistent axiom.

Method `direct` is specified in terms of itself. The specification is clearly not satisfiable by a pure method. However, one can give witnesses r and OS' that satisfy the specification, namely by choosing $r = \gamma(\texttt{direct()+1}, OS, \_)$ and $OS' = OS$. By providing this witness, the axiomatization enables one to derive $\hat{direct}(t, OS) = \hat{direct}(t, OS) + 1$ and, thereby, *false*.

JML allows directly-recursive specifications of pure methods if the recursion is well-founded [12]. To describe a termination argument for the recursion, methods can be annotated with a measured_by clause. This clause specifies an integer expression that must always be at least zero. The measure for recursive calls in the specification must strictly decrease [21].

We follow JML's rules for recursive specifications of pure methods. That is, we permit direct recursion in the ensures clauses of a pure method, provided that an appropriate measured_by clause is satisfied. However, we do not allow indirect recursion as well as recursion in requires clauses, measured_by clauses, and `\old` expressions. We also disallow method calls in invariants. Invariants can be regarded as a conjunct of each pre- and postcondition. Since invariants have to hold for all allocated objects [22], they introduce a form of recursive specification that is not decreasing according to the measure.

To simplify the formalization and, in particular, the soundness proof, we impose two further requirements: (1) All measured_by clauses in the specification of a pure method m are identical. That is, it is not allowed to use different measured_by clauses in different specification cases. This restriction corresponds to the measures used in theorem provers [21], but is not required by JML. (2) For each encoding of an ensures clause $ens^C_{m,i}$, we require that each occurrence of $\hat{m}$ in $\gamma(ens^C_{m,i}, OS', OS)$ takes OS' as store argument. This restriction allows us not to track store changes



```
class Factorial {

  /*@ normal_behavior
    @   requires 0 <= i && i < 2;
    @   ensures \result == 1;
    @ also
    @ normal_behavior
    @   requires 2 <= i;
    @   ensures \result == i * factorial(i-1);
    @   measured_by i;
    @*/
  /*@ pure @*/ int factorial(int i) { /* ... */ }
}
```

Figure 8: The measured_by clause allows one to prove that the recursive specification of method `factorial` is well-defined.

in the formalization of measured_by clauses. It is met by all practical examples we have considered so far, but disallows recursive calls where the actual parameters are in turn method calls.

The requirement of well-founded recursion can be formalized as follows. Assume method m in class C takes one explicit parameter, par. $mb_m(t, p, OS)$ denotes the result of applying γ to the measured_by expression of method m—with t and p substituted for `this` and par, respectively—, OS, and an arbitrary store. We add another antecedent to the axioms for $\hat{m}$ and $\hat{mS}$ that requires one to prove for each recursive call $o.m(q)$ in a normal behavior specification case for m in C that the corresponding measure is non-negative and decreasing. We formalize this by the following *measure condition*:

$$PRE^C_{m,i}(OS) \;\Rightarrow 0 \leq mb_m(o, q, OS) < mb_m(\texttt{this}, par, OS)$$

We use $wfn^C_m(OS)$ to denote the conjunction of the measure conditions for each recursive call to m in an ensures clause for m in class C.

The measure condition rules out the specification of method `direct`. Since the recursive call takes exactly the same parameter, `this`, it is not possible to give a decreasing measure.

The `factorial` method in Fig. 8 illustrates the use of measured_by clauses. The measure condition for this specification can be proved easily. The requires clause of the second specification case, $2 \leq \texttt{i}$, implies that the measure, `i`, is non-negative and decreasing ($0 \leq \texttt{i} - 1 < \texttt{i}$).

For recursive methods that traverse object structures, ownership [6] can be used

to describe measures. More specifically, we can use the height of an object in the ownership hierarchy as a measure. For instance, the `equals` method of a binary tree can be specified recursively, provided that the children of a tree node are owned by the parent node.

Summary

To summarize the previous subsections, we present the axiom for a pure method m in class C with one explicit parameter par. To use the properties of $\hat{m}$ and $\hat{mS}$, one has to show that: (1) the arguments of these functions are allocated and that the receiver object t is a non-null instance of the enclosing class; (2) the specification of m is satisfiable (by giving a witness); (3) each recursion in m's specification is well-founded (by giving a measure). These three conditions are the antecedents of the axiom for m:

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

Soundness

In this subsection, we show that the axiomatization of the functions for pure methods is consistent. That is, we prove the following theorem:

Theorem. Let **P** be a specified program where pure methods are not used mutually in their specifications. There is a model for the axioms generated from the specifications of **P**'s pure methods.

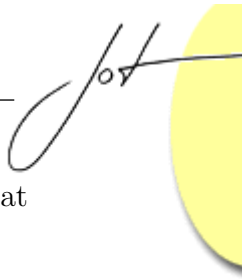
We prove this theorem in two steps. In Step 1, we prove an auxiliary lemma that there is a model for each individual axiom. In Step 2, we prove the theorem.

Step 1: Individual axioms.

We prove the following auxiliary lemma:

Lemma. Let **P** be a specified program where pure methods are not used mutually in their specifications. Let m be a pure method declared in a class C of **P**, where the functions for all methods other than m that are used in m's specification are well-defined. Then there is a model for the axiom generated for m and C.

Proof of lemma. We recursively define $\hat{m}(t, p, OS)$ and $\hat{mS}(t, p, OS)$ as follows. Let Δ be the set of all indices i such that $PRE_{m,i}^C(OS)[t/\texttt{this}, p/par]$ holds.



Now $\hat{m}(t, p, OS)$ and $\hat{mS}(t, p, OS)$ are defined to yield any values r, OS' such that $POST^{C}_{m,i}(OS', OS)[t/\texttt{this}, p/par, r/resV]$ holds for all $i \in \Delta$.

This definition of $\hat{m}$ and $\hat{mS}$ is well-defined for the following reasons: (1) Any potential recursion is well-founded. For direct recursion, this is guaranteed by the antecedent $wfn^{C}_{m}(OS)$ of the axiom and the fact that OS' is a pure successor of OS, which implies that the measure yields the same result in both stores. Indirect recursion is not allowed. (2) There actually are values r and OS' to be chosen, which follows from the antecedent of the axiom that requires the existence of a witness.

Step 2: Theorem.

We define a *depends graph* as follows: (1) The nodes of the graph are the pure methods of $\mathbf{P}$. (2) There is an edge from node m to node n if a normal behavior specification case of method m mentions method n. By the restrictions explained in Sec. 5, we know that the depends graph is acyclic except for direct cycles of just one node.

The proof of the theorem 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 the 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 $typeof(t) \preceq S$ 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 ($typeof(t) \preceq D \preceq C$) such that $(\exists r, OS' \bullet spec^{D}_{m}(t, p, OS, r, OS'))$ 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 $\hat{mS}(t, p, OS)$ such that they satisfy the axiom for m in T. According to the auxiliary lemma, 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: $spec_m^T \Rightarrow spec_m^S$. Therefore, r and OS' also satisfy $spec_m^S(t, p, OS, r, OS')$. □

Note that this soundness proof assumes single subtyping. With multiple subtyping (that is, interfaces), there can exist superinterfaces of $typeof(t)$ 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 generating a proof obligation for each type with several supertypes to show that the inherited specifications are satisfiable.

6 A PLEA FOR VALUE TYPES

In the previous sections we have shown that weak purity can be handled in a sound way. However, the resulting formulas are complicated. We have also shown that the encoding of specification expressions can be simplified significantly for expressions of primitive types. Further simplifications are possible in programming languages that provide non-primitive value types such as expanded types in Eiffel or structs in C#. Since instances of value types do not have an identity, value types qualify for the same simplifications as primitive types.

Eiffel's and C#'s value types are rather restricted because it is not possible to implement recursive data structures as value types. However, Spec# offers built-in value types for sets and sequences. The existence of such value types enables a dramatic simplification of the encoding of pure methods. It seems practical to completely prevent pure methods from returning new reference type objects. Data abstraction can still be expressed using (built-in) value types.

If pure methods must not return new reference type objects, they are effectively strongly-pure. With this requirement, specification expressions can be encoded without using store functions for pure methods or the ω function. Another possibility is to explicitly specify methods as weakly-pure or strongly-pure to simplify the encoding of the latter.

To check how frequently value types are used in specifications, we counted the number of pure methods with result type `String` in the Internet voting tally system [11]. Although strings are not value types in Java and C#, their immutability and the semantics of equality on strings in C# [7] suggest that strings behave very much like value types.

We found that strings are returned by 34 per cent (63 out of 182) of the pure methods. In fact, only 5 per cent (9 out of 182) of the pure methods have a reference result type different from `String` and are actually used in specifications. This result underlines the importance of supporting value types for specification purposes. It also shows that the model library of a specification language should consist of value

types or should use value type semantics.

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 [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 [11]. We have verified all unsound examples presented in this paper with ESC/Java2. ESC/Java [8] does not permit method calls in specifications.

For pure methods with restricted ensures clauses, the KRAKATOA tool [16] generates function definitions rather than axioms. 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.

Model fields are similar to parameterless methods. Model fields are specification-only fields whose values are typically not looked up in the store when accessing them, but are determined by applying mappings to the concrete states of objects. In general, such mappings may be based on weakly-pure methods, thus a sound handling of model fields has to cope with the problems of weak purity.

Breunesse and Poll [2] address the consistency problem for model fields. 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.

Work by Leino and Nelson [13, 15], and Müller [19] provides a sound and modular handling of model fields using different techniques. However, neither of the two approaches allows representation functions for model fields to include method calls, thus preventing the need of handling side-effects.

Recent work by Leino and Müller [14] extends the Boogie methodology to handle model fields in a sound, modular, and practical way. One of the main novelties of the approach is that the values of model fields are stored in the heap, just like the values of ordinary fields. The representation functions are only applied at certain well-defined points of the program to compute the values to be stored. Therefore, reading a model field does not have side-effects because the representation function

is not applied. For instance for a model field m, $m == m$ yields true, even if the representation for m allocates a new object. Leino and Müller's approach handles weak purity for model fields, but not for general method and constuctor calls in specifications.

Naumann [20] 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 is possible by weakening the $\trianglelefteq$ relation such that modifications of invisible fields are permitted.

8 CONCLUSIONS

In this paper, we presented a formalization of pure methods that allows one to reason about method calls in JML specifications. Pure methods are encoded by uninterpreted function symbols and axioms. This encoding can be expressed in a variety of logics. The axiomatization of the functions handles weak purity and is consistent, even if the JML specification is not satisfiable.

Our encoding can also be applied to JML's model fields [3], which face the same problems as pure methods because computing the value of a model field may involve the allocation and initialization of objects.

The main restriction of our approach is that method calls are disallowed in invariants as pointed out in Sec. 5. As part of future work, we will investigate ways to relax this restriction.

Our main conclusion from the work presented here is that value types simplify the treatment of pure methods dramatically. Our recommendation for language designers is to provide at least built-in value types such as the sets and sequences in Spec#. Because of the observations described in this paper, the Boogie program verifier requires pure methods not to return references to newly allocated objects, which allows Boogie to use a simplified encoding of pure methods. As future work, we plan to implement our results also in Jive [18] based on a core model library of value types.

Acknowledgments

We are grateful to David Cok, Werner Dietl, Gary Leavens, Rustan Leino, Farhad Mehta, Nicole Rauch, Burkhart Wolff, and the reviewers of the ECOOP FTfJP workshop version of this paper 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] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, 2005.

[4] D. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, October 2005.

[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*, 4(8):5–32, October 2005.

[7] C# Language Specification. Standard ECMA-334, 3rd Edition, June 2005.

[8] 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.

[9] 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.

[10] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Inf.*, 1(4):271–281, 1972.

[11] 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.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Iowa State University, 2005.

[13] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[14] K. R. M. Leino and P. Müller. A verification methodology for model fields. In P. Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *LNCS*. Springer-Verlag, 2006.

[15] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[16] 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.

[17] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[18] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system—implementation description. Available from sct.inf.ethz.ch/publications, 2000.

[19] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[20] D. Naumann. Observational purity and encapsulation. In M. Cerioli, editor, *Fundamental Aspects of Software Engineering (FASE)*, volume 3442 of *LNCS*. Springer-Verlag, 2005.

[21] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 1995.

[22] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[23] 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.

[24] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.

ABOUT THE AUTHORS

Ádám Darvas is PhD student at ETH Zurich. He works on semantics of modular specification languages and on the development of the JIVE verification tool. Email: adam.darvas@inf.ethz.ch.

Peter Müller is assistant professor and head of the Software Component Technology Group at ETH Zurich. His research focuses on specification and verification of object-oriented software. Email: peter.mueller@inf.ethz.ch.