Welldefinedness and Expressiveness of JML Specifications

Adrian Moos
adrian.moos@student.ethz.ch

October 24, 2005

Contents

1 Introduction

The Java Modeling Language [4] (JML) is a behavioral interface specification language for Java. In order to be easily learnable by ordinary programmers, its specification expression syntax is an extension of the side-effect free subset of Java expressions. Among these extensions is the ability to call methods, provided they are pure, which means that they cannot have any side-effect on previously allocated objects. The use of pure methods makes specifications more compact and expressive, but their translation to logic also introduces difficulties.

In this report we explore the expressive potential and the dangers inherent to this extension by using pure methods to specify a class from JML's model library. We also make some observations about difficulties introduced by the use of pure methods and explore potential solutions.

1.1 Background

According to Preliminary Design of JML [4], a method may appear in a specification if and only if it is annotated **pure**. A method annotated pure is restricted

with the intent to let it behave like the evaluation of a mathematical function[1]. Specifically, a pure method

- is provably free of observable side-effects, since it is not permitted to modify any location that was allocated in its pre-state, which is checked syntactically.
- provably terminates given its precondition is met
- is deterministic

Unfortunately, the restrictions above fall short of that intent: Darvas and Müller show that the store changes by a pure method must in general be modeled to prevent unsoundness[1]. Also, I show in section 4.2 that side effects not observable by the program can be observed in specifications and used to affect the return value of another specification expression, contradicting the claim that the evaluation order of specification expressions does not matter. We hope, but did not prove, that the specifications we use are not affected by such side effects.

1.2 Quantifier Semantics

In JML, a quantified specification expression ranges over all *potential* values of the specified type. If it is a reference type, this includes references to objects not allocated so far [4]. The semantics of specification expressions referring to the state accessible by these references are not defined. In this report, we assume that properties enforced by the programming language (such as the type of the reffered object) can be assumed for these objects, but properties enforced by the program (i.e. conformance to its specification) can not – after all, the latter are established by the execution of the constructor, which happens after object allocation. In order to assume an object's specification for a quantified variable, we therefore need to restrict the quantification range. Gary Leavens suggested[2] to test for allocation using a newly defined predicate

\allocated(o)

which yields true if and only if the object pointed to by o has already been allocated. Thus, \allocated(null) will always yield false.

To make full use of not-yet-allocated references, it is neccessary that pure methods can be invoked on them. This peculiar notion is beneficial to the expressiveness of specifications, but complicates attempts to achieve sound verification. We show the former by using that feature in a case study. We discuss the latter later in this report.

We think it is counter-intuitive that this can not be assumed to point to an allocated object in a specification. We therefore define that, in this report, a

[1] i.e. for every pure non-static method m with one parameter p, there is a mathematical function f such that the return value of $r.m(p)$, executed in state OS, is $f(OS, r, p)$.

[2] on the mailing list JML interest, see http://sourceforge.net/mailarchive/forum.php?thread_id=7244394&forum_id=13320

method's receiver can always be assumed to be allocated. Also, we extend the meaning of the annotation **non_null** to exclude not-yet-allocated objects. For instance, the method specification

```
/*@ public normal_behavior
  @ requires P;
  @ ensures Q;
  @*/
abstract /*@ non_null @*/ Object foo(/*@ non_null @*/ Object p);
```

desugars to

```
/*@ public normal_behavior
  @ requires
  @     \allocated(this) && // since the receiver is always allocated
  @     \allocated(p) &&    // since p is non_null
  @     P;
  @ ensures
  @     \allocated(\result) && // since the return value is non_null
  @     Q;
  @*/
abstract Object foo(Object p);
```

This mirrors the treatment of null references in Java and should therefore be easy to understand for programmers. Also, it permits to exclude non-allocated references with little notational overhead.

It should be noted that these syntactical conventions have no bearing on the additional need for checking the allocation status of references prior to dereferencing them that is caused by including not-yet-allocated references in quantification range.

1.3 Overview

In section 2, we briefly discuss the improvements to JML's expressiveness permitted by the use of pure methods, then spend quite some time discussing the useful properties they jeopardize and how these could be reestablished. In section 3, we study the questions thus raised by analyzing and then respecifying a class from JML's model library using two different approaches. In section 4, we revisit some problems encountered in greater generality and suggest potential solutions. In section 5, we conclude.

2 Method Calls in Specifications – Opportunities and Difficulties

2.1 Desirability of Method Calls in Specifications

JML extends the syntax of JML specification expressions to include calls to pure methods. This introduces a highly flexible way to reuse specification expressions.

Also, it adds a more powerful means to reason about object state: Without method calls, the only way a specification can refer to object state is by using concrete or model fields. In either case, it is neccessary to give a representation of object state. By specifying the method (or the set of methods) that accesses it, we can specify modification of object state. If this method (or this set of methods) is abstract, we do not need to give a representation of object state. For instance, in the following skeleton[3] specification of a hash map, the declaration of get() postulates a mapping from Object → Object, which is modified by set().

```
//@ pure
abstract Object get(Object key);

/*@ public normal_behavior
  @ ensures get(key)==value;
  @*/
abstract void set(Object key, Object value);
```

Using calls to pure methods in specification therefore enables us to talk about object state even in the absence of a representation. In contrast, if calls to pure methods in specifications can not be used, we are required to give a representation.

2.2 Encapsulation of Object State

We can use data groups to infer noninterference for concrete and model fields. Therefore, we can infer noninterference for object state if a representation is provided. However, there is no syntactic construct permitting this for representationless state. A more thorough presentation of this problem and potential solutions can be found in section 4.3.

2.3 Reestablishing Modularity

Consider a class with two methods m and n with associated specifications s, t. Formally, we represent a method by the semantic function describing its execution[4], and a specification as predicate over the set of methods that yields true iff the methods satisfy the specification. Permitting method calls in specifications shatters an important independence result: If method calls are not permitted in specifications, s does not depend on n and t does not depend on m:

$$\forall m, n, x; s(m, n) = s(m, x) \wedge t(m, n) = t(x, n)$$

This property is used to reason about safe implementation exchange, to show behavioral subtyping permits safe application of the substitution principle and other useful properties. Unless additional restrictions are applied, if method

[3] *skeleton*, since it is incomplete
[4] thereby avoiding the Fragile Base Class problem

calls are permitted in specifications, it is not true. For instance, in the skeleton specification above, replacing the implemenation of get() can invalidate the method specification of set(), even while still conforming to get()'s method specification. Thus, unless additional restrictions are in place, behavioral subtyping is not enough to guarantee safe subtyping or implementation exchange.

Unfortunately, since the ability to indirectly specify the semantics of a method by describing its relationship with another method is also responsible for their ability to describe object state, prohibiting such indirect specification would defeat their purpose. On the other hand, if such indirect specification were not restricted, modular verification would not be possible, because every class loaded into the system could carry an additional invariant relying on undocumented behavior of an otherwise unrelated class. Modular verification will therefore by necessity have to restrict the range of indirect specifications. Various kinds of restrictions are conceivable, but as this report deals primarily with other matters, we discuss only one set.

1. While proving a contract containing pure methods, the implementation of the contained methods may not be used

   For instance, no implementation of set() could be proven correct, as the effect it supposedly has on get can not be verified due to get()'s lack of a specification. This rule has the drawback of requiring method specifications even for trivial methods. Also, since it must be possible to prove a specification from the implementation, the latter must be at least as strong as the former[5] for this method to be applicable[6]. On the bright side, implementation exchanges are safe and, as long as behavioral subtyping is strictly abided by, overriding methods is, too. Unfortunately, strict compliance to inherited contracts conflicts with information hiding, as the former requires all contracts to be respected, but the latter permits hiding a contract from the subclass by declaring it private. In this report, we will ignore the visibility problem by assuming *all* specifications are respected by the subclass.

   That rule implies that no verified, implemented method can indirectly specify the semantics of a pure method appearing in its contract.

2. Even in specifications, pure methods may only be called if their receiver is provably allocated.

   Since the receiver is allocated, there is a concrete behavioral subtype in the system. We assume all loaded types have been verified. Therefore, this behavioral subtype has been verified. By the claim proven from rule 1, we conclude that the method in question can not indirectly specify pure methods. Therefore, if a pure method is called in a specification, it does not indirectly specify another pure method.

[5] as seen in section 4.1.1 this is non-trivial to achieve

[6] Circumventing that requirement by requiring correctness for allocated objects only still permits indirect specification for non-allocated objects

For instance:

```
class A {
    //@ pure
    boolean m();
}

abstract class B {
    /*@ public normal_behavior
      @     ensures (\forall A a; \allocated(a); !a.m());
      @ pure
      @*/
    abstract void n();
}
```

$B.n$ indirectly specifies $A.m$. However, if rule 2 is enforced and $B.n$ is called in a specification, it provably does not indirectly specify $A.m$.

This rule does not prevent indirect specification. However, it defines a *safe use* of specifications that prevents indirect specification.

2.4 Sound Verification

Now that we have modularity, we address soundness. In program verification, we want to show that the implementation of a class C is correct, i.e. the program will perform according to its specification, i.e. for every program run all method invocations[7] i are correct:

$$\forall i; correct(i)$$

To prove this, we use induction along execution time, i.e. we define

$$j < i \Leftrightarrow j \text{ completes before } i \text{ does}$$

$<$ is well-founded, since at any point in time, only finitely many invocations have completed and completion times are totally ordered.

$$\forall i; (\forall j; (j < i) \Rightarrow correct(j)) \Rightarrow correct(i)$$

which is implied by

$$\forall i; (\forall j; (j \text{ is called by } i) \Rightarrow correct(j)) \Rightarrow correct(i) \tag{1}$$

since

$$(j \text{ is called by } i) \Rightarrow (j < i)$$

Equation (1) is then proved by case split according to the method invoked, assuming the implementation of that method and the semantics of Java.

[7] *not* method names

Unfortunately, this proof obligation is not sufficient to prove a method specification that refers to a different pure method as the semantics of that method are not known. We therefore need to provide additional information to the proof obligation. We have seen in the previous subsection that providing the implementation is not advisable. Therefore, we strive to use the specification, that is: whenever a specification calls a pure method, we would like to permit assuming correctness of that invocation, i.e. that its contract is met. Note that this constitutes safe use of specifications, thus ruling out indirect specification. We know verification is sound if there is a well-founded relation $\sim$ such that invocation i may only rely on the correctness of invocation j if $j \sim i$. This was trivial for implementations because we could assume their termination, i.e. that the call-graph is well-founded. However, when a pure method is invoked in a specification, it depends not on the execution of the method, but on its specification, and while the execution is required to terminate, the specification may recurse without restriction. Therefore, the dependency graph for specifications need not be well-founded, and thus induction along it need not be permissible. Therefore, the verification system must prove this well-foundedness of depedency prior to assuming the correctness of specification expressions. If that step is skipped, the class may be incorrect even though all methods satisfy the proof obligation:

```
/*@ public normal_behavior
  @ ensures \result && !n();
  @*/
//@ pure
boolean m() {return true;}

/*@ public normal_behavior
  @ ensures \result && !\result && m();
  @*/
//@ pure
boolean n() {return true;}
```

The first conjunct of m's proof obligation can be inferred from Java's semantics, the second from correctness of n. The first conjunct of n's proof obligation is implied by Java's semantics, the other two by correctness of m.

2.5 Checking Well-foundedness of Dependency

We say an invocation i of a specification expression depends on invocation j if and only if i's contract evaluates j. Similarly, we say a method specification s depends on method specification t if and only if the contract of s contains a call to the method t specifies. Obviously, if the method dependencies are well-founded, so are invocation dependencies.

If that does not work, we can try inferring well-foundedness by providing a mapping to some ordered set. For example, we can use allocation order by showing:

$$\forall i, j; (i \text{ depends on } j) \Rightarrow receiver(j) <_{alloc} receiver(i)$$

which we can deduce in constructor specification invocations using that in the prestate

$$o \neq this \wedge allocated(o) \Rightarrow o <_{alloc} this$$

We will use this method in this report.

Or, if we have an ownership relation at our disposal, we can write

$$\forall i, j; (i \text{ depends on } j) \Rightarrow receiver(i).owns(receiver(j))$$

which again implies well-foundedness.

However, no single ordering is applicable to all programs, therefore, several well-founded relations need to coexist. Though we have not looked into this, our intuition is that recursive specifications will rarely be mutually dependent, therefore, interaction between different relations is rare (and of course detectable).

Also, in general, the discovery of such mappings will have to be helped along. [4] suggests to use a method's measured_by clause for that purpose, which works by mapping the parameter space of the method onto a well-founded set, for example the non-negative integers. However, its use can be a little impractical as we will witness in our case-study.

A main goal of our work was to explore if and how specifications can be rewritten to minimize the effort required to prove their soundness.

3 Case Study: JMLObjectSet

3.1 Introduction

3.1.1 JMLObjectSet

We consider the task of specifying JMLObjectSet, whose instances represent mathematical sets of references, where two elements a and b are considered equal if and only if the references are identical, i.e. a==b. JMLObjectSet therefore differs in behavior from java.util.Set, which considers elements a,b equal if and only if a.equals(b).

In order to keep matters simple, we focus on the core functionality of JMLObjectSet (creation, insertion, removal, comparison, cardinality and iteration) and omit convenience methods, since their semantics can easily be specified in terms of the sequence of operations they are a shortcut for, and such specifications are not difficult to handle.

To formally define the intended semantics, let A be the abstraction function, i.e. for the current[8] object store, $A[s]$ denotes the set represented by s. Then, we define[9]:

[8] or, at our leisure, any descendants of it, because instances are immutable

[9] to keep matters simple, we will ignore integer overflows in this report

$$
\begin{aligned}
A[\text{new JMLObjectSet}] &= \emptyset \\
A[s.insert(a)] &= A[s] \cup \{a\} \\
A[s.remove(a)] &= A[s] \setminus \{a\} \\
s.isSubset(s2) &= A[s] \subseteq A[s2] \\
s.equals(s2) &= A[s] = A[s2] \\
s.count() &= |A[s]|
\end{aligned}
$$

To permit accessing instances of JMLObjectSet in specifications, all members are **pure**; insert () and remove() therefore work by returning new instances.

In order to permit treating the instance like the value it represents, instances are immutable.

3.1.2 Goals of the Specification

Our goal is to find a specification for JMLObjectSet that is

- verifiably sound
- correct
- complete
- concise
- understandable (by programmers)

We call a specification verifiably sound if and only if its satisfiability can be proved automatically. We call it correct, if and only if the properties of the datatype the programmer has in mind imply the specification:

$$datatype \Rightarrow specification$$

which implies that every property provable from the specification is also provable for the datatype. We call it complete, if and only if the specification implies the logical description of the datatype:

$$datatype \Leftarrow specification$$

which implies that every property provable for the datatype is also provable from the specification.

3.2 Equational

A well-studied approach for specifying sets is to use an equational theory. The official specification [3] of JMLObjectSet contains an encoding of such a theory in JML. We use the slightly beautified version listed in the appendix as starting point.

Unfortunately, this specification has several shortcomings, causing it to be both unsound and incomplete[10], which we will describe and fix shortly. Before we do that, you are welcome to try spotting them.

3.2.1 Fix: Specification Describes Unallocated State

In JML, quantification over reference types includes unallocated non-null references in specification expressions. Consequently, the argument s2 to equational_theory () can not be assumed to be null or allocated, however, axioms 5 and 6 refer to the result of method invocations on s2.

Non-allocated objects have not yet been initialized by the program and thus, according to the semantics of JML, yield arbitrary values of the appropriate type. Thus, the implementation of instance methods actually depending on the state accessible through this is underspecified for these objects, and to enable verification of an implementation, so must the specification. Therefore, barring cases where the state refererred to by a reference passed to a method (in receiver or parameter) need not be used in the implementation, we may not specify semantics in case the reference does not point to a allocated object.

Note that references to unallocated objects do not cause that problem, only *dereferencing* them does. Moreover, references to unallocated objects can be useful, as we will witness shortly.

Here, isSubset() will clearly depend on the state referred to by this, and therefore, we refrain from overspecification by replacing the quantification's range predicate with allocated(s2). Note that since neither e1 nor e2 need to be dereferenced, omitting their allocation requirement is safe. Furthermore, we shall see omitting the check is beneficial.

It should be noted that these allocation requirements are already implied by the syntax convention from section 1.2 (which also establishes modularity rule 2), retroactively justifying that convention. However, for the sake of clarity, we state allocation requirements explicitly in this report.

3.2.2 Fix: Instance Invariant Needs to Hold Only If Instances Exist

In the above specification, the equational theory is enforced as an instance invariant of JMLObjectSet. This is too weak, since an instance invariant is only required to hold for every instance. If no instance has been allocated so far, the instance invariant is therefore trivially satisfied. The constructor's specification being part of that instance invariant, it, too, is trivially satisfied, regardless of

[10] the official specification compensates for the latter by additionally providing largely – but not completely – redundant method specifications.

what the first constructor invocation does, thus failing to specify its semantics. For example, the implementation

```
private static boolean first=true;

//@ pure
JMLObjectSet() {
    // proper intialization to empty set
    if ( first ) {
        // evil code
        first =false;
    }
}
```

is provably correct, regardless of what modifications of local state happen in //evil code. To get a better look at the problem, we desugar the instance invariant to:

```
/*@ public static invariant
  @     (\ forall JMLObjectSet s; \allocated(s); (
  @     (\ forall JMLObjectSet s2; \allocated(s2); (
  @     (\ forall Object e1, e2; ;
  @         equational_theory(s, s2, e1, e2) )));
  @*/
```

and notice that the range predicates are too restrictive at least for the axiom specifying the constructor. This is not a flaw of this particular equational theory or of its summary treatment of the axioms: It is easy to prove that an equational specification of constructor semantics can only be formulated in the prestate of the constructor's execution, where an instance invariant is trivially satisfied if no object has been allocated so far. It is therefore impossible to completely and equationally specify constructor semantics using an instance invariant.

We solve this problem by using a static invariant, where we can drop the global allocation requirement. We compensate by requiring allocation for those axioms that refer to the state pointed at by the quantified variable, i.e. those that mention a field or a method on the quantified variable.

3.2.3 Fix: Incorrect Specification of Remove

The following axiom is incorrect, i.e. permits derivation of properties that do not hold for a set:

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

The first case of the ternary operator is incorrect, as it implies that if a is an object reference and

$$\{a\} = (\text{new JMLObjectSet}().\text{insert}(a))$$

then

$$\{a\}.\text{insert}(a).\text{remove}(a).\text{equals}(\{a\})$$

which clearly contradicts the semantics of mathematical sets.

We fix this by splitting the axiom for readability and adding a precondition to the first case:

```
@          !s.has(e1) ==> s.insert(e1).remove(e1).equals(s)
@      ) && (
@          e1!=e2 ==>
@              s.insert(e1).remove(e2).equals(s.remove(e2).insert(e1))
```

3.2.4 Fix: Informal Specification of Immutability

Immutability is specified as a comment. We make this specification machine readable and checkable by translating it to the history constraint

```
//@ public constraint (\forall Object o; \old(has(o)) == has(o));
```

Note that although, from a mathematical point of view, it might be more intuitive (and general) to specify an object's immutability as its equality to any later version of itself, we can not express this is JML, as it would involve the evaluation of equals() while its parameters refer to two different object stores.

3.2.5 Fix: Incomplete Specification of Equals

The equational theory defines equals() using s2, which is an instance of JMLObjectSet . However, the parameter type of equals is Object. Therefore, the specification does not define the semantics of equals() for references of other classes. For instance, the return value of:

```
new JMLObjectSet().equals("strange");
```

is unspecfied. We fix this by quantifying over o and checking the parameter's dynamic type:

```
@          \allocated(s) ==> (\forall Object o; \allocated(o); (
@              s.equals(o) == (o instanceof JMLObjectSet)
@                          && s.isSubset((JMLObjectSet)o)
@                          && ((JMLObjectSet)o).isSubset(s)))
```

Note that requiring allocation for o does not cause incompleteness since the case $o = \text{null}$ is already defined by the specification inherited from Object [5].

3.2.6 Fix: equals() Does Not Get Special Treatment in JML

In mathematics, equality is not a relation like any other. By general convention, equal objects never behave differently and can therefore for all intents and purposes be treated as if they were identical. Specifically, we can assume

$$a = b \Rightarrow f(a) = f(b) \text{ for all functions f}$$

Though Java's equals() is modeled after that mathematical ideal, its contract only requires it to be an equivalence relation [5]. Thus, the equivalent to the above equation,

$$(a.equals(b)) \Rightarrow f(a).equals(f(b)) \text{ for all operations } f,$$

is *not* implied by the contract of equals. Short of using reflection to quantify over all methods (which works here since they are all pure), this would be impossible to express at that level of generality, also, such a restrictive definition would not be applicable to all implementations of equals() out there.

Translating mathematical models to JML therefore requires this implicit assumption to be stated explicitly, while failure to do so ruins completeness. For example, since our above specification assumes the special nature of equals without prescribing it, serveral axioms[11] are weaker than intended:

$$\begin{aligned} ax8 &\not\Rightarrow \neg\text{new JMLObjectSet().remove(a).remove(a).has(a);} \\ ax3 &\not\Rightarrow 0 = \text{new JMLObjectSet().remove(a).count();} \\ ax11 &\not\Rightarrow \text{new JMLObjectSet().remove(a).isEmpty();} \end{aligned}$$

Indeed, these expressions are undefined. We prove this by giving a correct implementation that does not compute the expected return value. For the first expression, the implementation given by the formulas

$$\begin{aligned} \text{new JMLObjectSet()} &= (\emptyset, \emptyset) \\ \text{(I,R).insert(a)} &= (I \cup \{a\}, R \setminus \{a\}) \\ \text{(I,R).remove(a)} &= (I \cup \{a\}, R \oplus \{a\}) \\ \text{(I,R).has(a)} &= a \in I \setminus R \end{aligned}$$

does not contradict any axiom, but

new JMLObjectSet().remove(a).remove(a).has(a);

contrary to the semantics of sets, yields true.

For the second expression, we take a correct implementation and add the line

size=−4;

at the end of remove(). The correct implementation satisfies the axiom system, and the modification does not change that, since no axiom contains both count() and a property whose modification by remove() is specified. Nevertheless, the code blatantly violates our notion of set cardinality.

For the third expression, since isEmpty() is specified in terms of count() only, we can use a variant of the preceding exploit.

[11] the numbering refers to the improved specification listed in section A.2

We solve this problem by prescribing the special nature of equality for the operations where it is not already implied by the specification in an additional axiom:

```
@  s.equals(s2) ==> (
@       s.count() == s2.count()
@  && s.remove(e1).equals(s2.remove(e1)))
```

3.2.7 The Fruit of Our Labor

The result of applying the above patches is listed in section A.2

We could have specified choose() equationally, but, mirroring the official specification we started from, did not.

3.2.8 Discussion

Our goal in this case study is to find a specification that is:

- verifiably sound
- correct
- complete
- concise
- understandable

Using mathematical logic, the specification's soundness is a direct corollary of its correctness and the existence of mathematical sets. However, the datatype the specification is supposed to describe is not available in machine-readable form, therefore, neither correctness nor the existence of such a datatype is available to the verification environment. From the point of view of the verification environment, it must be shown that there is a group of semantic functions satisfying the specification. The verification environment does not know that sets satisfy the specification and will thus be hard pressed to find a witness for satisfiability. Aiding it along by specifying induction order is feasible, but cumbersome due to its complexity. We conclude that it is not obvious how the specification's soundness could be verified automatically unless we burden the programmer with cumbersome specifications of induction order.

Correctness of the specification can be proved by verifying that an implementation constructed according to the datatype satisfies the specification. Completeness could probably be shown by proving that the specification implies the axioms specifying the data structure. The specification is hardly concise and definitely not easy to understand by programmers.

Intuitively, it can't be that hard to meet our goals when specifying something as simple as sets. Indeed, it turns out we can do much better – provided we choose another approach.

Our equational specification lacks an internal structure. Consequently, it is not clear how to locate an axiom pertaining to some method short of looking through the entire equational theory. Also, since axioms pertaining to the same method are separated, it is easier to overlook contradictory specifications. Also, the fewer method calls appear in specifications, the less dependent the method specifications are (recall that changing the implementation of a pure method can falsify a contract it appears in[12]; overriding a pure method therefore requires revalidation of all methods whose contracts refer to it).

3.3 Abstract Public Specification

It seems therefore more intuitive and more practical for the user of the specification to use method specifications to reduce the use of methods in specifications to a minimum. We call this approach method-centric.

We shall prove shortly that the following method-centric specification is equivalent to the equational one, that its soundness is easy to verify, and that the correctness of its non-abstract methods can be proved.

To showcase JML's usefulness for abstract specification, we specify the public interface without providing (and thus without relying on) an implementation.

The specification's listing can be found in section A.3. Some remarks:

- even though has() lacks a method specification, we shall see that it is sufficiently specified.
- instance immutability (i.e. that the set represented by an instance is immutable) is prescribed using the history constraint
- the precondition of equals() is necessary since we do not want to specify the semantics of equals() if and only if $s2$ points to a not-yet-allocated object, as such specification would refer to unallocated state, which, as we have seen in section 3.2.1, is not advisable. The semantics in case $s2 = \text{null}$ are already defined by the inherited specification and therefore not repeated.
- empty() is included to show that object initialization can be specified, too. The capability for object initialization is not provided by a constructor, because constructors can not be abstract, and no implemenation could be verified in the abstract setting.
- like the equational specification, this specification relies on the inclusion of not-yet-allocated objects in quantification range to specify that objects created after initialization of the set are not part of the set. Otherwise, the return value of

[12] for instance, correctness of the contract method might have been proven using a private invariant that, since it is not visible to the overriding subclass, is not respected by the overriding method

```
boolean demo() {
    JMLObjectSet s = new JMLObjectSet();
    Object o = new Object();
    return s.has(o);
}
```

would be unspecified, as the reference to the newly allocated object would not be in the quantification range of the constructor's contract. This matter is discussed in detail in section 4.1.

3.3.1 Soundness

To prove that the method-centric specification is sound, we observe that, because every method is specified as a well-defined function of has(), if has() is sound, so is the entire class. To show has() is sound, we have to show there is a semantic function satisfying the specification of the *class* for every object there could be. We do this using induction along the sequence of method invocations.

We observe there are only four ways client code can obtain a reference to an instance of JMLObjectSet: the constructor and the methods empty(), insert() and remove(). For the induction step, we assume that has() is well-defined for every allocated object in the pre-state of the operation invoked. We proceed by case split:

- the operation is a constructor call:
  The constructor's postcondition is obviously sound.
- the operation is empty():
  the postcondition is obviously sound
- the operation is insert():
  by the induction hypothesis, we know has() is well-defined for every allocated object. The receiver is allocated, thus has() is well-defined for the receiver. The postcondition defines \result.has() as a well defined function of this.has(). Thus, has() is well-defined.
- the operation is remove():
  analogous to preceding case

Therefore, has() is well-defined for every newly passed out object. In addition, by the induction hypothesis, we know that for every previously passed out object, has() is well-defined in the prestate, i.e. there is a semantic function satisfying all axioms. The same semantic function trivially satisfies the history constraint in the poststate. Also, the postcondition of the operation executed in the meantime does not mention and thus does not care about previously passed out objects. Therefore, has() is well-defined in the poststate for all previously passed out objects as well.

We conclude that has() is well-defined for every allocated object there could be. Thus, the specification is sound. q.e.d

The preceding proof follows a general pattern, only the parts specific to the specification of JMLObjectSet would therefore have to be discovered by the verification environment, which is within its capabilities.

3.4 Implementation Specification

The code listed in section A.4 provides an implementation for JMLObjectSet in a subclass to it. We shall see shortly that the implementation is specified sufficiently for it to pass verification. Also, the specification's soundness can be verified. It represents sets internally using a singly linked list without duplicates.

- If $A[s]$ denotes the set represented by instance s,

$$A[s] = \emptyset \text{ if and only if } s.next = null$$

$$A[s] = \{s.value\} \cup A[s.next] \text{ otherwise}$$

- The invariant states that the list does not contain duplicates. It also implies acyclicity of the linked list.
- insert_nonexisting () and remove_existing() differ from insert () and remove () rsp. only by their stronger precondition. Their more restrictive nature can be used to assert membership properties in passing, as demonstrated by the client code in the next section.

3.4.1 Logical Structure

Unlike the abstract specification we have seen before, the implementation specification refers to has() on other instances. We could provide a measured_by clause featuring count() to state an ordering on instances that could be used to guarantee the absence of a cyclic dependency, but count() itself depends on the recursive specification, so we would have to inline its definition, resulting in a quite lengthy annotation. Fortunately, the class is immutable and no instance points to itself, resulting in a provably well-founded reference structure. This special nature can be exploited to prove soundness:

3.4.2 Soundness

Formally, we have to show that, for every *allocated*[13] object, there is a semantic function satisfying the description of has(). Since we may assume the receiver to be allocated, we know it has been initialized by the program, and can use induction along this program execution to prove soundness.

Induction hypothesis: the specification of has() is sound for all allocated objects. Base case: trivially true. Induction step: Every object is either publicly accessible in the prestate or newly allocated:

[13]since by convention from section 1.2, every specification has an implicit precondition stating that the receiver is allocated

- accessible in prestate:
  Only local axioms can introduce unsoundness. If no operation takes place, only the invariant and the history constraint are applicable. By the induction hypothesis, the specification was sound in the prestate, that is: there is a semantic function satisfying both history constraint and invariant in the prestate. The same function for the post state satisfies the history constraint and invariant in the poststate. Therefore, the specification is sound for this object in the poststate.

- newly allocated:
  The new object causes new axioms to require satisfaction. Those are: the invariant, the contract for has.

  The object has been created by a constructor. Its postcondition defines $\forall o; \neg has(o)$. The contract for has() agrees and implies: $next = null$. The inherited contracts agree and imply nothing new.

  The second constructor:

  ```
  @ requires \allocated(n) && !n.has(v);
  @ ensures value=v && next==n;
  ```

  We notice the invariant is satisfied if the constructor's postcondition holds. Then, the contract for has is satisfiable, provided $next.has()$ is. By the induction hypothesis, $has()$ is well-defined for all objects allocated in the prestate. By the requires clause, n is allocated in the prestate. Thus, $n.has()$ is satisfiable. In the preceeding case we have seen we may propagate soundness over time. Therefore, $next.has()$ is satisfiable in the poststate.

Note that the above proof only works because we can assume the receiver to be allocated. It seems like the parts specific to JMLObjectSet would be simple enough to be found automatically. Also, the framework might be applicable in greater generality. We therefore show the crucial ingredient from a more general point of view:

We permitted to assume the soundness of method invocation $m(p)$ when proving soundness of $m(q)$ if:

$$\backslash allocated(p) \wedge \neg \backslash allocated(q)$$

which is sound, since $\backslash allocated$ is a monotone predicate over execution time

$$t < t' \Rightarrow \backslash allocated(t) \Rightarrow \backslash allocated(t')$$

and the condition thus implies:

$$allocationtime(p_i) < allocationtime(q_i)$$

permitting the assumption of the induction hypothesis.

We used this assumption in the prestate of q's allocation, propagated the knowledge about $m(p)$ to the poststate and used it to establish satisfiability of $m(q)$. It is conceivable that this approach could work for structural induction on immutable datatypes in general.

3.4.3 Correctness

We have informally verified the more problematic methods in the implementation and are confident it is correct. However, due to a lack of time, we did not prove correctness formally.

3.5 Example: Using the Public Specification to Prove Correctness of Client Code

Assuming correctness of JMLObjectSetImpl, we can prove correctness of the following code:

```
public class JMLObjectSetClient {
    /*@ public normal_behavior
      @ ensures (\forall Object o; \result.has(o) == s1.has(o) && s2.has(o)
            );
      @*/
    public JMLObjectSet intersect(/*@ non_null @*/ JMLObjectSet s1, /*
        @ non_null @*/ JMLObjectSet s2) {
        JMLObjectSet s = s1;
        JMLObjectSet r = new JMLObjectSetImpl();
        while (!s.isEmpty()) {
            Object co = s.choose();
            if (s2.has(co)) {
                r=r.insert_nonexisting(co);
            }
            s=s.remove_existing(co);
        }
        return r;
    }
}
```

Please note that the use of the assertion variants insert_nonexisting() and remove_existing() only strengthens the proof. Ok, here we go:

JMLObjectSet s = s1;

$$\forall o; s.has(o) = s1.has(o)$$

JMLObjectSet r = new JMLObjectSetImpl();

$$r.isEmpty() \wedge \forall o; s.has(o) = s1.has(o)$$

$$\forall o; \neg r.has(o) \wedge s.has(o) = s1.has(o)$$

to keep notation short, we define:

$$loopinv := \forall o; \quad (s.has(o) \Rightarrow s1.has(o)) \\ \wedge \quad (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o))$$

then

$$loopinv$$

while (!s.isEmpty()) {

$$\neg s.isEmpty() \wedge loopinv$$

Object co = s.choose();

$$s.has(co) \wedge loopinv$$

$$\neg r.has(co) \wedge s1.has(co) \wedge loopinv$$

if (s2.has(co)) {

$$s.has(co) \wedge \neg r.has(co) \wedge s1.has(co) \wedge s2.has(co) \wedge r = R \wedge loopinv$$

r=r.insert_nonexisting(co);

$$\begin{array}{l} s.has(co) \wedge s1.has(co) \wedge s2.has(co) \\ \wedge r.has(co) \wedge \forall o; o \neq co \Rightarrow (R.has(o) \Leftrightarrow r.has(o)) \\ \wedge \forall o; (s.has(o) \Rightarrow s1.has(o)) \wedge (R.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o)) \\ )) \\ \Rightarrow \\ s.has(co) \wedge s1.has(co) \wedge s2.has(co) \wedge r.has(co) \wedge (\forall o; (\\ \quad (s.has(o) \Rightarrow s1.has(o)) \wedge \\ \quad (o \neq co \Rightarrow (s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o) \Leftrightarrow r.has(o)) \\ \Rightarrow \\ s.has(co) \wedge (r.has(co) \Leftrightarrow s1.has(co) \wedge s2.has(co)) \wedge (\forall o; (\\ \quad (s.has(o) \Rightarrow s1.has(o)) \wedge \\ \quad (o \neq co \Rightarrow (s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o) \Leftrightarrow r.has(o)) \\ )) \\ \Rightarrow \\ s.has(co) \wedge (\forall o; (\\ \quad (s.has(o) \Rightarrow s1.has(o)) \wedge \\ \quad (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge (\neg s.has(o) \vee o = co)) \\ )) \end{array}$$

} **else** {

$$s.has(co) \wedge \neg s2.has(co) \wedge loopinv$$

$$s.has(co) \wedge \neg s2.has(co) \wedge \forall o; (\\ s.has(o) \Rightarrow s1.has(o)) \wedge (\\ r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o))\\ )$$

$$s.has(co) \wedge (\forall o; (\\ (s.has(o) \Rightarrow s1.has(o)) \wedge \\ (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge (\neg s.has(o) \vee o = co))\\ )$$

}

$$s.has(co) \wedge (\forall o; (\\ (s.has(o) \Rightarrow s1.has(o)) \wedge \\ (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge (\neg s.has(o) \vee o = co))\\ )$$

s=s.remove_existing(co);

$$\neg s.has(co) \wedge \forall o; (o \neq co \Rightarrow s.has(o) = S.has(o)) \wedge (S.has(co) \wedge (\forall o; (\\ S.has(o) \Rightarrow s1.has(o)) \wedge (\\ r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge (\neg S.has(o) \vee o = co))\\ ))$$

$$\Rightarrow$$

$$\neg s.has(co) \wedge (\forall o; (\\ s.has(o) \Rightarrow S.has(o)) \wedge (\\ S.has(o) \Rightarrow s1.has(o)) \wedge (\\ r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o)\\ ))$$

$$\Rightarrow$$

$$loopinv$$

}

$$s.isEmpty() \wedge \forall o; (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o) \wedge \neg s.has(o))$$

$$\Rightarrow$$

$$\forall o; (r.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o))$$

return r;

$$\forall o; \backslash result.has(o) \Leftrightarrow s1.has(o) \wedge s2.has(o)$$

3.6 Proof Sketch: Equivalence of Method-centric and Equational Spec

3.6.1 Method-centric ⇒ Equational

We have to use JMLObjectSetImpl instead of JMLObjectSet since only the concrete implementation supplies a callable constructor. The proof exploits that knowledge only in order to use the contract of the constructor; the other specifications in JMLObjectSetImpl are not used.

Assume there is an implementation satisfying the method-centric spec. We have to prove this implementation also satisfies the equational spec.

Since said implementation satisfies the method spec, when can assume that the properties given in the specification hold in all publicly visible states. We note that all these properties contain the implicit[14] precondition that guarantees allocatedness of this. This precondition is automatically satisfied if we use a reference returned by a member of JMLObjectSetImpl, which is all what we will be doing.

1. $\neg(\text{new JMLObjectSetImpl}()).has(e1)$

   $\overset{constructor}{\Rightarrow}$

   $\text{new JMLObjectSetImpl}().isEmpty()$

   $\overset{isEmpty()}{\Rightarrow}$

   $\forall o; \neg(\text{new JMLObjectSetImpl}().has(o))$

   $\neg(\text{new JMLObjectSetImpl}().has(e1))$

2. $s.insert(e1).has(e2) \Leftrightarrow (e1 = e2 \vee s.has(e2))$

   case $e1 = e2$

   $\overset{insert}{\Rightarrow}$

   $s.insert(e1).has(e1)$

   $s.insert(e1).has(e2)$

   $e1 = e2 \vee s.has(e2)$

   $s.insert(e1).has(e2) \Leftrightarrow (e1 = e2 \vee s.has(e2))$

   case $e1 \neq e2$

   $\overset{insert}{\Rightarrow}$

   $s.insert(e1).has(e2) = s.has(e2)$

   $s.insert(e1).has(e2) = (e1 = e2 \vee s.has(e2))$

[14] by convention in section 1.2

3. new JMLObjectSetImpl().$count() = 0$

$\overset{constructor}{\Rightarrow}$

$$isEmpty()$$

$\overset{isEmpty()}{\Rightarrow}$

$$\forall o; \neg has(o)$$

$$0 \overset{\Downarrow}{=} (\backslash\text{sum } o; has(o); 1) \overset{count}{=} count()$$

4. $s.insert(e1).count() = (s.has(e1)?s.count() : s.count() + 1)$

case $s.has(e1)$

$\overset{insert}{\Rightarrow}$

$$\forall o; s.insert(e1).has(o) \Leftrightarrow s.has(o)$$

$\overset{count}{\Rightarrow}$

$$s.insert(e1).count() = s.count()$$

case $\neg s.has(e1)$

$\overset{insert}{\Rightarrow}$

$$s.insert(e1).has(e1)$$
$$\wedge \forall o; o \neq e1 \Rightarrow (s.insert(e1).has(o) \Leftrightarrow s.has(o))$$

let o by any object, then:

$$o \neq e1 \Rightarrow (s.insert(e1).has(o) \Leftrightarrow s.has(o))$$

$$\begin{aligned} o \neq e1 \wedge s.insert(e1).has(o) &\Leftrightarrow o \neq e1 \wedge s.has(o) \\ &\Leftrightarrow s.has(o) \end{aligned}$$

$$\begin{aligned} s.insert(e1).count() &\overset{count}{=} (\backslash\text{sum } o; s.insert(e1).has(o); 1) \\ &= \underbrace{[s.insert(e1).has(e1)]}_{1} + \underbrace{(\backslash\text{sum } o; o \neq e1 \wedge s.insert(e1).has(o); 1)}_{\underbrace{(\backslash\text{sum } o; s.has(o); 1)}_{s.count()}} \end{aligned}$$

5. $s.isSubset(s2) \Leftrightarrow (\forall o; (s.has(o) \Rightarrow s2.has(o))$

$\overset{isSubset}{\Rightarrow}$

$$s.isSubset(s2) \Leftrightarrow (\forall o; (s.has(o) \Rightarrow s2.has(o))$$

6. $s.equals(o) \Leftrightarrow (o\text{instanceof JMLObjectSet}) \wedge s.isSubset(o) \wedge o.isSubset(s)$

case $o$instanceof JMLObjectSet

$$\begin{aligned} & s.equals(s2) \\ \overset{equals}{\Leftrightarrow} \quad & (\forall o; s.has(o) \Leftrightarrow s2.has(o)) \\ \Leftrightarrow \quad & ((\forall o; s.has(o) \Rightarrow s2.has(o)) \wedge (\forall o; s2.has(o) \Rightarrow s.has(o))) \\ \overset{isSubset}{\Leftrightarrow} \quad & s.isSubset(s2) \wedge s2.isSubset(s) \end{aligned}$$

case $\neg$($o$instanceof JMLObjectSet)

$$\overset{equals}{\Rightarrow} \quad \neg s.equals(o)$$

$$\neg((o\text{instanceof JMLObjectSet}) \wedge s.isSubset(o) \wedge o.isSubset(s))$$

$$s.equals(o) \Leftrightarrow (o\text{instanceof JMLObjectSet}) \wedge s.isSubset(o) \wedge o.isSubset(s)$$

7. $s.equals(s2) \Rightarrow s.count() = s2.count() \wedge s.remove(e1).equals(s2.remove(e1))$
assume $s.equals(s2)$

$$\overset{equals}{\Rightarrow} \quad \forall o; s.has(o) \Leftrightarrow s2.has(o)$$

$$\sum_{\substack{o \\ s.has(o)}} 1 = \sum_{\substack{o \\ s2.has(o)}} 1$$

$$\overset{count}{\Rightarrow} \quad s.count() = s2.count()$$

Let o be an object. Then:

case $o = e1$

$$\overset{remove}{\Rightarrow} \quad \neg s.remove(e1).has(o)$$

$$\neg s2.remove(e1).has(o)$$

$$s.remove(e1).has(o) \Leftrightarrow s2.remove(e1).has(o)$$

case $o \neq e1$

$$\begin{array}{ccc} s.remove(e1).has(o) & \overset{remove}{\Leftrightarrow} & s.has(o) \\ & & \Updownarrow \\ s2.remove(e1).has(o) & \overset{remove}{\Leftrightarrow} & s2.has(o) \end{array}$$

therefore

$$\forall o; s.remove(e1).has(o) \Leftrightarrow s2.remove(e1).has(o)$$

$\overset{equals}{\Rightarrow}$

$$s.remove(e1).equals(s2.remove(e1))$$

8. (new JMLObjectSetImpl()).$remove(e1).equals$(new JMLObjectSetImpl())
   let $\emptyset$ = new JMLObjectSetImpl()
   let o be any object

   case $o = e1$
   $\overset{remove}{\Rightarrow}$

   $$\neg\emptyset.remove(e1).has(e1)$$

   $$\neg\emptyset.remove(e1).has(o)$$

   $\overset{constructor}{\Rightarrow}$

   $$\neg\emptyset.has(o)$$

   $$\emptyset.remove(e1).has(o) \Leftrightarrow \emptyset.has(o)$$

   case $o \neq e1$
   $\overset{remove}{\Rightarrow}$

   $$o \neq e1 \Rightarrow (\emptyset.remove(e1).has(o) \Leftrightarrow \emptyset.has(o))$$

   $$\emptyset.remove(e1).has(o) \Leftrightarrow \emptyset.has(o)$$

   Therefore

   $$\forall o; \emptyset.remove(e1).has(o) \Leftrightarrow \emptyset.has(o)$$

   $\overset{equals}{\Rightarrow}$

   $$\emptyset.remove(e1).equals(\emptyset)$$

9. $\neg s.has(e1) \Rightarrow s.insert(e1).remove(e1).equals(s)$
   assume $\neg s.has(e1)$
   let o be an object

   case $o = e1$

   $$\neg s.has(o)$$

   $\overset{remove}{\Rightarrow}$

   $$\neg s.insert(e1).remove(e1).has(o)$$

   $$s.insert(e1).remove(e1).has(o) \Leftrightarrow s.has(o)$$

   case $o \neq e1$

   $$s.insert(e1).remove(e1).has(o) \overset{remove}{\Leftrightarrow} s.insert(e1).has(o) \overset{insert}{\Leftrightarrow} s.has(o)$$

Therefore

$$\forall o; s.insert(e1).remove(e1).has(o) \Leftrightarrow s.has(o)$$

$$\overset{equals}{\Rightarrow}$$

$$s.insert(e1).remove(e1).equals(s)$$

10. $e1 \neq e2 \Rightarrow s.insert(e1).remove(e2).equals(s.remove(e2).insert(e1))$
assume $e1 \neq e2$
let o be any object

case $o = e1$

$$o \neq e2$$

$$\overset{insert}{\Rightarrow}$$

$$s.insert(e1).has(e1)$$

$$s.insert(e1).has(o)$$

$$\overset{remove}{\Rightarrow}$$

$$o \neq e2 \Rightarrow (s.insert(e1).remove(e2).has(o) \Leftrightarrow s.insert(e1).has(o))$$

$$s.insert(e1).remove(e2).has(o) \Leftrightarrow s.insert(e1).has(o)$$

$$s.insert(e1).remove(e2).has(o)$$

$$\overset{insert}{\Rightarrow}$$

$$s.remove(e2).insert(e1).has(e1)$$

$$s.remove(e2).insert(e1).has(o)$$

$$s.insert(e1).remove(e2).has(o) \Leftrightarrow (s.remove(e2).insert(e1).has(o))$$

case $o = e2$
symetric to previous case. proof can be obtained by consequently exchanging $insert/remove$, $e1/e2$ and putting $\neg$ in front of all intermediate results of the form $S.x$ for some S.

case $e1 \neq o \neq e2$

$$\overset{insert_{15}}{\Rightarrow}$$

$$\forall S; o \neq e1 \Rightarrow (S.insert(e1).has(o) \Leftrightarrow S.has(o))$$

$$\forall S; S.insert(e1).has(o) \Leftrightarrow S.has(o)$$

[15] technically, this is not true, since the contract can only be applied in a state the receiver is allocated in. However, for all instantions of this rule, this condition is met. More in the remark after the proof.

$\overset{remove}{\Rightarrow}$

$$\forall S; o \neq e2 \Rightarrow (S.remove(e2).has(o) \Leftrightarrow S.has(o))$$

$$\forall S; S.remove(e2).has(o) \Leftrightarrow S.has(o)$$

We now instantiate these rules for $S \in \{s, s1, s2\}$:

$$\begin{array}{lll} \underbrace{s.insert(e1)}_{\equiv s1}.remove(e2).has(o) & \overset{S1}{\Leftrightarrow} & s.insert(e1).has(o) \\ & & \Updownarrow s \\ & & s.has(o) \\ & & \Updownarrow s \\ \underbrace{s.remove(e2)}_{\equiv s2}.insert(e1).has(o) & \overset{S2}{\Leftrightarrow} & s.remove(e2).has(o) \end{array}$$

Therefore

$$\forall o; s.insert(e1).remove(e2).has(o) \Leftrightarrow (s.remove(e2).insert(e1).has(o))$$

$\overset{equals}{\Rightarrow}$

$$s.insert(e1).remove(e2).equals(s.remove(e2).insert(e1))$$

11. $isEmpty() \Leftrightarrow (count() = 0)$

$\overset{count}{\Rightarrow}$

$$count() = \sum_o [has(o)]$$

$$count() = 0 \Leftrightarrow \sum_o [has(o)] = 0 \Leftrightarrow (\forall o; \neg has(o)) \overset{isEmpty}{\Leftrightarrow} isEmpty()$$

3.6.2 Equational ⇒ Method-centric

We begin with a few lemmas:

1. $s.equals(s2) \Leftrightarrow (\forall o; s.has(o) \Leftrightarrow s2.has(o))$

$$\begin{array}{lll} s.equals(s2) & \overset{a6}{\Leftrightarrow} & s.isSubset(s2) \wedge s2.isSubset(s) \\ & \overset{a5}{\Leftrightarrow} & (\forall o; (s.has(o) \Rightarrow s2.has(o)) \wedge (s.has(o) \Leftarrow s2.has(o))) \\ & \Leftrightarrow & (\forall o; s.has(o) \Leftrightarrow s2.has(o)) \end{array}$$

2. $s.insert(e1).insert(e2).equals(s.insert(e2).insert(e1))$
(*insert* operations commute with respect to equals)
Proof:
Let o be any object

$$s.insert(e1).insert(e2).has(o) \overset{a2}{\Leftrightarrow} (o = e2 \vee s.insert(e1).has(o)) \overset{a2}{\Leftrightarrow} (o = e2 \vee o = e1 \vee s.has(o))$$
$$\Updownarrow$$
$$s.insert(e2).insert(e1).has(o) \overset{a2}{\Leftrightarrow} (o = e1 \vee s.insert(e2).has(o)) \overset{a2}{\Leftrightarrow} (o = e1 \vee o = e2 \vee s.has(o))$$

Therefore

$$\forall o; s.insert(e1).insert(e2).has(o) \Leftrightarrow s.insert(e2).insert(e1).has(o)$$

$\overset{L1}{\Rightarrow}$

$$s.insert(e1).insert(e2).equals(s.insert(e2).insert(e1))$$

3. $(s.insert(e1).insert(e1).equals(s.insert(e1)))$
(idempotency of *insert* with identical parameters)
Proof:
Let o be any object

$$\begin{gathered} s.insert(e1).insert(e1).has(o) \\ a2 \Updownarrow \\ (o = e1 \vee s.insert(e1).has(o)) \\ a2 \Updownarrow \\ (o = e1 \vee o = e1 \vee s.has(o)) \\ \Updownarrow \\ (o = e1 \vee s.insert(e2).has(o)) \\ a2 \Updownarrow \\ s.insert(e1).has(o) \end{gathered}$$

Therefore

$$\forall o; s.insert(e1).insert(e1).has(o) \Leftrightarrow s.insert(e1).has(o)$$

$\overset{L1}{\Rightarrow}$

$$(s.insert(e1).insert(e1).equals(s.insert(e1)))$$

4. $s.equals(s2) \Rightarrow s.u().equals(s2.u())$ where $u() \in \{insert(e1), remove(e1)\}$
(updates maintain equality)
Assume $s.equals(s2)$

case $u = insert(e1)$

$\overset{L1}{\Rightarrow}$

$$\forall o; s.has(o) \Leftrightarrow s2.has(o)$$

let o be any object

$$s.has(o) \Leftrightarrow s2.has(o)$$

$$\begin{array}{rll} s.insert(e1).has(o) & \overset{a2}{\Leftrightarrow} & (e1 = o \vee s.has(e2)) \\ & & \Updownarrow \\ s2.insert(e1).has(o) & \overset{a2}{\Leftrightarrow} & (e1 = o \vee s.has(e2)) \end{array}$$

$\overset{L1}{\Rightarrow}$

$$s.insert(e1).equals(s2.insert(e1))$$

$$s.u().equals(s2.u())$$

case $u = remove(e1)$

$\overset{a7}{\Rightarrow}$

$$s.remove(e1).equals(s2.remove(e1))$$

$$s.u().equals(s2.u())$$

The method specification only needs to hold for allocated receivers. We know that every instance of JMLObjectSet has been returned by the constructor, or by calling insert () or remove() on another instance. We can therefore characterize every instance by the sequence of operations that created it.

5. there is a mapping f on operation sequences with the properties
   (a) $f(seq).equals(seq)$
   (b) $f(seq)$ contains only *insert* operations. Moreover, every parameter is unique within the sequence.

   Proof sketch:
   We proceed by giving a provably terminating algorithm computing f. Let seq be the argument supplied to the algorithm. It has the invariant:

   $$\theta.equals(seq)$$

   The algorithms starts by setting θ to seq. By its contract in class Object, $equals$ is reflexive. We therefore have:

   $$seq.equals(seq)$$

   $$\theta.equals(seq)$$

i.e. the invariant holds.

We call an update of θ by θ' legal if and only if $\theta'.equals(\theta)$. Since $equals$ is transitive, legality implies:

$$\theta'.equals(seq)$$

i.e. the invariant is maintained by all updates to θ. We will prove all updates done by the algorithm are legal and thus maintain the invariant.

By axiom 10, insert and remove operations with differing arguments commute at the end of the sequence with respect to equals. By Lemma 4, this holds everywhere within the sequence. Therefore, we can rotate the first remove operation to the front until the remove operation hits a corresponding insert operation or the constructor call.

case remove hits constructor

We remove the call to *remove* from the sequence (legal due to axiom 8 and lemma 4)

The *remove* has been removed from the sequence while maintaining the invariant.

case remove hits corresponding insert

Because the current remove operation was the first and has only moved to the head of the sequence, it is still first. Consequently, there are only the constructor call and insert operations before it. Also, we know there is an insert operation with matching argument before the remove operation.

We rotate every *insert* operation with a matching argument towards the *remove* until there are no non-matching inserts between it and remove (obviously possible, legal due to lemma 2).

It then removes all but one matching inserts (legal due to lemma 3).

Now, the object represented by presequence of the matching insert does not contain remove's argument.

We then remove the matching insert along with its remove (legal due to axiom 9, whose precondition we have just established, and lemma 4)

The *remove* has been removed from the sequence while maintaining equals.

We now repeat the above procedure until there are no more removes left (which will happen, since the number of removes is a non-negative integer and decreases with each iteration. Also, each iteration produces a well-defined result) (this is legal since every step in every iteration is).

We then reorder the insert operations such that the sequence of arguments is monotone according to some order (legal due to lemma 2) and remove duplicate inserts (legal due to lemma 3).

We then return the current sequence.

Claim: The returned sequence satisfies the properties.
Proof: Since every step is legal and the invariant is initially satisfied, property a) holds. Property b) trivially holds.

We are now ready to prove the contracts:

- *isSubset*: $\backslash result \Leftrightarrow (\forall o; has(o) \Rightarrow s2.has(o))$
  $\overset{a5}{\Rightarrow}$
  $$this.isSubset(s2) \Leftrightarrow (\forall o; (this.has(o) \Rightarrow s2.has(o))$$
  $$\backslash result \Leftrightarrow (\forall o; (has(o) \Rightarrow s2.has(o))$$

- *equals*: $\backslash allocated(s2) \Rightarrow (\backslash result \Leftrightarrow (s2 \text{instanceof } \text{JMLObjectSet}) \wedge (\forall o; has(o) \Leftrightarrow ((JMLObjectSet)s2).has(o))$
  Assume $\backslash allocated(s2)$

  case $s2 \text{instanceof } \text{JMLObjectSet}$
  $\overset{L1}{\Rightarrow}$
  $$this.equals(s2) \Leftrightarrow (\forall o; this.has(o) \Leftrightarrow s2.has(o))$$
  $$\backslash result \Leftrightarrow (\forall o; has(o) \Leftrightarrow s2.has(o))$$
  $$\backslash result \Leftrightarrow (s2 \text{instanceof } \text{JMLObjectSet}) \wedge (\forall o; has(o) \Leftrightarrow s2.has(o))$$

  case $\neg(s2 \text{instanceof } \text{JMLObjectSet})$
  $\overset{a6}{\Rightarrow}$
  $$\neg\backslash result$$
  $$\neg((s2 \text{instanceof } \text{JMLObjectSet}) \wedge (\forall o; has(o) \Leftrightarrow s2.has(o)))$$
  $$\backslash allocated(s2) \Rightarrow$$
  $$(\backslash result \Leftrightarrow (s2 \text{instanceof } \text{JMLObjectSet}) \wedge (\forall o; has(o) \Leftrightarrow s2.has(o))$$

- *insert*: $\backslash result.has(o) \wedge (\forall o2; o2 \neq o \Rightarrow (\backslash result.has(o2) \Leftrightarrow has(o2)))$
  $\overset{a2}{\Rightarrow}$
  $$\forall o2; this.insert(o).has(o2) \Leftrightarrow (o = o2 \vee this.has(o2))$$
  $$\forall o2; \backslash result.has(o2) \Leftrightarrow (o = o2 \vee has(o2))$$

  let $o2 = o$

  $$(o = o2 \vee has(o2))$$
  $$\backslash result.has(o2) \Leftrightarrow (o = o2 \vee has(o2))$$
  $$\backslash result.has(o2)$$

  therefore
  $$\backslash result.has(o)$$

assume $o2 \neq o$

$$(o = o2 \vee has(o2)) \Leftrightarrow has(o2)$$

$$\backslash result.has(o2) \Leftrightarrow has(o2)$$

therefore

$$\forall o2; o2 \neq o \Rightarrow (\backslash result.has(o2) \Leftrightarrow has(o2))$$

$$\backslash result.has(o) \wedge (\forall o2; o2 \neq o \Rightarrow (\backslash result.has(o2) \Leftrightarrow has(o2)))$$

- *remove*: $\neg\backslash result.has(o) \wedge (\forall o2; o2 \neq o \Rightarrow (\backslash result.has(o2) \Leftrightarrow has(o2)))$
  Let f be the mapping constructed in lemma 5
  Let rr be $f(\backslash result)$

  $\overset{L5}{\Rightarrow}$

  $$rr.equals(\backslash result)$$

  $\overset{L1}{\Rightarrow}$

  $$rr.has(o) \Leftrightarrow (\backslash result.has(o))$$

  By construction, there are only insert operations in rr. Also, there is no insert operation with an argument matching o. By axioms 1 and 2, we conclude:

  $$\neg rr.has(o)$$

  $$\neg(\backslash result.has(o))$$

  For the second part of the proof, let rt be $f(this)$. Let $o2$ be any object such that $o2 \neq o$. We observe that by construction, every insert operation with argument $o2$ is present in rt exactly if it is present in rr.

  $$this.has(o2) \overset{L5,L1}{\Leftrightarrow} rt.has(o2) \overset{a2}{\Leftrightarrow} rr.has(o2) \overset{L5,L1}{\Leftrightarrow} \backslash result.has(o2)$$

- *count*: $\backslash result = \sum_o [has(o)]$
  let rt be $f(this)$
  $\overset{L5}{\Rightarrow}$

  $$rt.equals(this)$$

  rt only contains insert operations. Moreover, every parameter appears at most once. We now prove

  $$\sum_o [rt.has(o)] = rt.count()$$

  by induction along the sequence rt.

– Base case:
$\overset{a1}{\Rightarrow}$

$$\forall o; \neg(\text{new JMLObjectSetImpl}().has(o))$$

$$\sum_o [\text{new JMLObjectSetImpl}().has(o)] = 0$$

$\overset{a3}{\Rightarrow}$

$$\text{new JMLObjectSetImpl}().count() = 0$$

$$\sum_o [\text{new JMLObjectSetImpl}().has(o)] = \text{new JMLObjectSetImpl}().count()$$

– Induction step:
Let rtp denote the previous presequence. Since rt contains no insert operation, the next presequence can be written as:

$$rtp.insert(e)$$

for some parameter e. Since parameters are unique in rt, rtp does not contain e. By axiom 2, we conclude:

$$\neg rtp.has(e)$$

$\overset{insert}{\Rightarrow}$

$$rtp.insert(e).has(e) \quad \wedge \quad (\forall o; o \neq e \Rightarrow (rtp.insert(e).has(o) \Leftrightarrow has(o)))$$

$$\begin{array}{rrcl}
 & \sum_o [rtp.insert(e).has(o)] & & \\
= & \sum_o [e = o \wedge rtp.insert(e).has(o)] & + & \sum_o [e \neq o \wedge rtp.insert(e).has(o)] \\
= & 1 & + & \sum_o [e \neq o \wedge rtp.has(o)] \\
= & 0 & + & \sum_o [e \neq o \wedge rtp.has(o)] + 1 \\
= & \sum_o [e = o \wedge rtp.has(o)] & + & \sum_o [e \neq o \wedge rtp.has(o)] + 1 \\
= & \sum_o [rtp.has(o)] + 1 & &
\end{array}$$

$\overset{a4}{\Rightarrow}$

$$rtp.insert(e).count() = rtp.count() + 1$$

We assume the induction hypothesis

$$\sum_o [rtp.has(o)] = rtp.count()$$

$$\sum_o [rtp.has(o)] + 1 = rtp.count() + 1$$

and conclude the induction step:

$$\sum_o [rtp.insert(e).has(o)] = rtp.insert(e).count()$$

We have proven:

$$\sum_o [rt.has(o)] = rt.count()$$

$\overset{L1}{\Rightarrow}$

$$\forall o; rt.has(o) \Leftrightarrow has(o))$$

$\overset{a7}{\Rightarrow}$

$$rt.count() = \backslash result$$

$$\sum_o [has(o)] = \backslash result$$

- $isEmpty()$: $\backslash result \Leftrightarrow (\forall o; \neg has(o))$
  $\overset{a11}{\Rightarrow}$

$$isEmpty() \Leftrightarrow (count() = 0)$$

  $\overset{count}{\Rightarrow}$

$$count() = \sum_o [has(o)]$$

$$count() = 0 \Leftrightarrow \sum_o [has(o)] = 0 \Leftrightarrow (\forall o; \neg has(o))$$

$$\backslash result \Leftrightarrow (\forall o; \neg has(o))$$

3.6.3 Conclusion

We have proven that the method-centric specification is equivalent to the equational one. This is surprising, as the method-centric specification is by far more restricted in the induction order its use requires than the equational theory. Also, we have seen that the non-trivial induction order required by the equational theory makes proving even relatively simple properties difficult and cumbersome.

3.7 Conclusion

We started with the equational specification extracted from the official specification of JMLObjectSet, which turned out to be both incorrect and incomplete, due to several bugs which seem to have been caused by a poor understanding of the pitfalls of encoding an equational theory in JML. In particular, we identified and fixed an incorrect axiom, unintended specification of unallocated state, a too weak specification due to the application condition inherent to instance invariants, inadvertent loss of a required basic mathematical property during translation. We also enhanced the original specification by a formal specification implying instance immutability. The resulting equational specification seems sound and complete, but the fixes cluttered the specification to an extent

where its usability was impaired. Even in its original, flawed, version, the specification was neither concise nor easy to understand. We think the flexibility of specification placement granted by using an invariant instead of method specifications contributed to that problem, as unnecessary flexibility in the absence of coding guidelines often does.

Consequently, we switched to a more orderly method-centric approach. We provided an abstract specification followed by an implementation in a subclass, which we verified based on the private specifications given. Checking the soundness of the private specification shed some light on automatic sanity-checking of recursive specifications. To demonstrate the expressiveness and ease of use of the specification, we gave a formal verification of a client method computing the intersection of sets. Also, we formally proved that our method centric specification is at least as strong as the equational one and provided a proof sketch for the reverse direction.

It was the goal of this case study to develop a verifiably sound, correct, complete, concise and understandable specification for JMLObjectSet. Our specification is provably sound, correct, and complete, and substantially more concise and understandable than previous approaches.

4 Peculiarities of JML

4.1 Range of Quantification

The proper definition of quantified expression poses a design decision: Should quantification over a reference type include references that point to objects not allocated yet? While this may seem a curious notion a first glance, [4] states this. To study the respective advantages of these approaches, we consider the problem of specifying the return value of

```
JMLObjectSet s = new JMLObjectSet();
Object o = new Object();
return s.has(o);
```

The relevant parts[16] of the method-oriented specification are:

```
//@ public constraint (\forall Object o; \old(has(o)) == has(o));

//@ pure
abstract boolean has(Object o);

/*@ public normal_behavior
  @ ensures (\forall Object o; ! has(o));
  @*/
//@ pure
JMLObjectSet() {...}
```

[16] definition of isEmpty() inlined to keep matters concise

4.1.1 Extended Quantification Range

If quantification range includes the non-allocated non-null references, the above specification permits us to prove that the return value is false.

Unfortunately, if the quantification range is extended and we are permitted[17] to use the quantified variables as receiver or actual parameter of methods, the semantic function described by the specification of has() must be defined for these references. However, a basic assumption in program verification is that an implementation is at least as strong as the specification, i.e. if the specification prescribes something, the implementation computes that. However, this would imply that, since the specification's semantic function must accept not-yet-allocated objects, so must the implementation's.

We can choose to forego that property or to establish it by slightly extending the semantics of Java to permit not-yet-allocated references at the discretion of the specification. In practise, this would mean to drop the assumption that parameters are either null or \allocated from Java's semantics, but add an additional proof obligation that no reference to not-yet-allocated state is dereferenced. Of course, a method's specification could require that a parameter is \allocated; only in the absence of such a requirement would the program's semantic function be generalized.

In essence, a reference can be used to represent an identity or as pointer to an object. Programs that only care about identity will not dereference and therefore satisfy the proof obligation. JMLObjectSet is such a class. If we do not require the parameter of has() to be allocated, but require that the parameter of insert() is allocated or null, we can easily prove

$$\forall o; has(o) \Rightarrow \backslash allocated(o) \vee o = null$$

while providing semantics for has() even for not-yet-allocated objects, and thus specifying the membership status of newly allocated objects, without requiring any program modification. However the notation of the allocated clauses also takes up space and is not conducive to readability.

It must be noted that while this extension of Java's semantics may seem unintuitive, sacrificing the property that the implementation is stronger than the specification is really ugly. For instance:

```
/*@ public normal_behavior
  @ ensures \result && (o!=null && !\allocated(o) ==> !safe ==> safe
      )
  @*/
//@ pure
boolean unsound(Object o) {return true;}

boolean safe;
```

[17] if we are not, the extended quantification range is useless at least in our example and where we must resort to the ugly approach described below

```
    /*@ public normal_behavior
      @ ensures safe && (\forall Object o;;unsound(o));
      @*/
    void main () {
        safe=false;
    }
}
```

The specification of unsound() is obviously unsound, but as we may assume $\backslash allocated(o) \vee o = null$ for the verification of unsound, we can prove $\neg safe$. Since unsound()'s specification does not call methods, dependency is well-founded, and main() may assume the specification of unsound(). However, since there is a reference o such that $o \neq null \wedge \neg\backslash allocated(o)$, permitting deduction $\neg safe \Rightarrow safe$ and thus $safe$. Also, this shatters our result that a correctly implemented method referring to a pure method in its specification can not constrain that method's semantics:

```
    //@ pure
    abstract boolean m();

    /*@ public normal_behavior
      @ ensures o!=null && !\allocated(o) ==> m();
      @*/
    //@ pure
    void n (Object o) {}
```

which falsifies our results about safe implementation exchange or overriding of pure methods. Extending the semantics of Java seems to be the better choice.

4.1.2 Limited Quantification Range

On the other hand, if quantification range were defined as the set of values permitted by the semantics of Java (specifically, null and references to objects, provided they are allocated), the specification proposed above would be invalid, since the history constraint would potentially call has() with an unallocated object. We can fix this by replacing the offending constraint with:

```
    //@ public constraint (\forall Object o;
    //@   has(o) == \old(\allocated(o)) ? \old(has(o)) : false );
```

While this is strong enough, and the implementation can be proven, this approach explicitly mentions object allocation, is therefore a nonmodular, dynamic description, very close to the operation of the program, but far from the view of the caller – at least, I don't think about a set by considering the effect object allocation has on membership ...

Also, unlike the preceding one, this approach does not generalize well to mutable sets: A history constraint is the only way to describe the effect of unrelated object allocation. If there was no mutation, a history constraint has to define has() to yield false for newly allocated objects. If a mutation

happened, has() must yield either true or false, depending on the mutation. A mutation changing has() to true must therefore leave a trace visible to the specification. We can use a modification counter in a model field to accumulate such traces:

```
// define an encapsulated instance model field
// containing a mathematical integer
// that is incremented by every insert

//@ public constraint \old(modcount) == modcount ==>
//@   (\forall Object o; ;
//@       has(o) == \old(\allocated(o)) ? \old(has(o)) : false);
```

but this is hardly concise ...

4.1.3 Conclusion

We conclude that while it is possible to specify the intended semantics with both approaches, the extended quantifier range permits a substantially smaller, more intuitive and more general specification than is possible without. On the other hand, proper handling of the problems introduced by extending the range also introduces some complexity and the need for additional notation.

4.2 Observability of Side Effects

The following specification observes side effects of a pure method. Assuming that these side effects can be ignored leads to unsound verification.

```
class A {
    /*@ public normal_behavior
      @ ensures !valid();
      @*/
    //@ pure
    A() {}

    /*@ public normal_behavior
      @ ensures !\result;
      @*/
    //@ pure
    boolean valid() {return false;}
}

class B {
    /*@ public normal_behavior
      @ ensures (\result instanceof A) ==> !((A)\result).valid();
      @*/
    //@ pure
    static /*@ non_null @*/ Object p() {
```

```
        return new A();
    }
}

class C {
    /*@ public normal_behavior
      @ requires \forall A a; \allocated(a) ==> a.valid();
      @*/
    void m() {
            Object o = B.p();
            if (o instanceof A) {
                A a = (A)o;
                assert a.valid();
            }
        }
    }
}
```

Claim: The above code (including the assertion) verifies correct.
Proof: $A.valid()$ is correct and so is $A.A()$. Since $A.A()$ is pure, so is $B.p()$, satisfying its contract. It remains to verify $C.m()$. At the beginning of method execution, we may assume the precondition. By the assumption and $B.p()$'s purity, we may assume the property holds after evaluating $B.p()$. Since the property does not refer to o, assignment to o does not affect the property, thus, it still holds before the if-statement. The condition in **if** is side-effect free, thus the property still holds in the true-branch. Casting is side-effect free, and updating a can not affect the property, it therefore still holds at the assertion, where we know $\backslash allocated(a) \wedge typeof(a) = A$ and can thus derive $a.valid()$. However, we can also prove $\neg a.valid()$, permitting to prove arbitrary properties about the program.

We conclude that execution of a pure method can violate a specification expression anywhere in the call chain. This has far-reaching consequences as the absence of such interference was used to prove that changing evaluation order of specification expressions does not affect the return value. Indeed, for instance, && is not commutative, since in a state where no object of class A was allocated so far, the following expressions differ in their return value:

```
(\forall A a; \allocated(a) ==> a.valid()) && !B.p().valid(); // true
!B.p().valid() && (\forall A a; \allocated(a) ==> a.valid()); // false
```

There seems to be no trivial change fixing this. It seems required to restrict the range of observation of quantification expressions, but no simple restriction seems to do the job, since the range of oberservation would have to depend on the current scope, and these scopes do not seem to have an immediately useful mathematical structure such as a subset-relation. Also, a reference entering scope (like in method $C.m()$) would entail additional proof obligations in order to safely include it in quantification range.

It is worth noting that this problem is not affected by the presence of not-yet-allocated objects in quantification range, since these are excluded by the quantification's range predicate in the preceding example.

4.3 Relative Immutability of Abstract State

We noted in the introduction that abstract state can be represented in JML by prescribing a relationship with other state, forming a recursion ending in concrete state. Alternatively, it can be represented using a family of abstract pure methods describing the state. We have seen functional properties can be expressed in either case. However, to use those functional properties in practise, and to modularly prove history constraints, we need to be able to establish that state is not manipulated from outside the instance (or at least the class). We call this property relative immutability, because only local methods can mutate the state. It is worth noting that relative immutability is weaker than encapsulation, because it is only concerned with write access to internal state.

In JML, we can limit the range of write effects using assignable clauses. Since an assignable clause can only mention visible locations, and concrete state can be declared private, we can enforce relative immutability for concrete state. Given a mapping from concrete to abstract state, we can propagate relative immutability to abstract state. However, if abstract state is represented using a family of pure methods, there is no syntactic construct that would permit inference of relative immutability.

For instance, in the specification of JMLObjectSet, we can not express that the return value of has() is relatively immutable, i.e. can be affected only using methods of the class. Therefore, the history constraint we used to prescribe immutability of has() is not modularly checkable.

If we want to avoid additional notation, we could use ownership types to restrict dependencies by enforcing that a pure method may only be affected by locations its receiver owns. Since we are only interested in relative immutability and not full encapsulation, the owner-as-modifier property [2] is sufficient, permitting the application of the relatively flexible universe type system [6].

However, this approach does not permit to keep state in the interface class, which is too restrictive in our opinion. Therefore we suggest to permit inference of relative immutability by defining an affected_by clause that, if present, exhaustively lists the method invocations[18] that can affect the method's return value. We can then specify relative immutability of has() by writing:

```
//@ affected_by \nothing
//@ pure
abstract boolean has(Object o);
```

Even relative immutability of a mutable variant of JMLObjectSet could then be conveniently specified:

```
//@ affected_by empty(),insert(o),remove(o)
```

[18] i.e. including receiver and parameters

```
//@ pure
abstract boolean has(Object o);
```

meaning that the return value of the expression $has(o)$ depends only on invocation of empty() with matching receiver, and on invocations of insert/remove with matching receiver and matching parameter. For safe subtyping, the subclass may not define additional methods that affect has, nor broaden the affected_by clause.

The reason we list method invocations, and not concrete fields or data groups is that concrete fields are too inflexible (no implementation exchange), and data groups may be extended by subclasses regardless of the relative immutability of the added elements. Specifically, a subclass may put public fields declared by it in a datagroup specified by its superclass.

5 Conclusion

In this report we studied the benefits and problems caused by permitting calls to pure methods in specifications. We have seen that pure methods extend JML's range of expressiveness to abstract state, but that their introduction jeopardizes modularity and soundness of verification. We suggested rules to prevent indirect specification from correctly implemented methods and thus partially reestablished modularity. We presented techniques to guarantee soundness of verification that work in our example, but whose general applicability has yet to be determined.

We developed two specifications for JMLObjectSet, where one is substantially more concise and user-friendly than the official version and proved their equivalence, correctness and completeness.

We showed that the public interface of JMLObjectSet can be specified with acyclic method specification dependencies. Specifying its implemenation, we showed that structural induction is best described using a recursive specification. We have not suceeded in finding an example where mutual recursion in specifications is required, giving rise to the hope that such cases are rare.

We showed that while including references to not-yet-allocated objects in quantification range eliminates the need to specify initial return values for these references, proper treatment of such references is not trivial and that the annotation overhead caused is significant. We also showed that it is not enough to prevent side-effects observable to the program in order to guarantee the absence of observable side effects to specification expressions and proved that contrary to [4], order of evaluation does matter for specification expressions. Finally, we showed that the usefulness of pure methods in specifications is limited because effects on the state they represent can not be expressed, and suggested a syntax extension to fix that.

References

[1] Á. Darvas and P. Müller. Reasoning About Method Calls in JML Specifications. In *Formal Techniques for Java-like Programs*, 2005.

[2] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.

[3] G. Leavens, A. Baker, C. Ruby, et al. JML-Specification of org.jmlspecs.models.JMLObjectSet. from www.jmlspecs.org. revision 1.72.

[4] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06x, Iowa State University, Department of Computer Science, 2004. See www.jmlspecs.org.

[5] Gary T. Leavens and Specifications from Compaq SRC's ESC/Java. JML-Specification of java.lang.Object. from www.jmlspecs.org. revision 1.42.

[6] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

A Listings

A.1 Original Equational Specification

```
//-@ immutable
public /*@ pure @*/ class JMLObjectSet
{
    //********************** equational theory **********************

    /*@ public invariant (\forall JMLObjectSet s2; s2 != null;
      @                    (\forall Object e1, e2; ;
      @                          equational_theory(this, s2, e1, e2) ));
      @*/

    /** An equational specification of the properties of sets.
     */
    /*@ public normal_behavior
      @ {|
      @     // The following are defined by using has and induction.
      @
      @ ensures \result <==> (
      @          !(new JMLObjectSet()).has(e1)
      @     ) && (
      @          s.insert(e1).has(e2) == (e1 == e2 || s.has(e2))
      @     ) && (
```

```
@            (new JMLObjectSet()).count() == 0
@     ) && (
@            s.insert(e1).count()
@                == (s.has(e1) ? s.count() : s.count() + 1)
@     ) && (
@            s.isSubset(s2)
@                == (\forall Object o; ; s.has(o) ==> s2.has(o))
@     ) && (
@            s.equals(s2) == (s.isSubset(s2) && s2.isSubset(s))
@     ) && (
@            (new JMLObjectSet()).remove(e1).equals(new
    JMLObjectSet())
@     ) && (
@             s.insert(e1).remove(e2).equals(
@                       e1 == e2 ? s : s.remove(e2).insert(e1)
@             )
@     );
@
@     // The following are all defined as abbreviations.
@
@ also ensures_redundantly \result <==>
@     (
@          s.isEmpty() == (s.count() == 0)
@     ) && (
@          (new JMLObjectSet(e1)).equals(new JMLObjectSet().insert(
    e1))
@     ) && (
@          s.isProperSubset(s2)
@                == ( s.isSubset(s2) && !s.equals(s2))
@     ) && (
@          s.isSuperset(s2) == s2.isSubset(s)
@     ) && (
@          s.isProperSuperset(s2) == s2.isProperSubset(s)
@     );
@ |}
@
@ implies_that     // other ways to specify some operations
@
@      ensures \result <==> (
@          (new JMLObjectSet()).isEmpty()
@     ) && (
@          !s.insert(e1).isEmpty()
@     ) && (
@          (new JMLObjectSet(e1)).has(e2) == (e1 == e2)
@     );
static public pure model boolean equational_theory(JMLObjectSet s,
```

```
                                                JMLObjectSet s2,
                                                Object e1,
                                                Object e2);
      @*/

    //@ pure
    abstract boolean has(Object o);

    //@ pure
    abstract boolean isEmpty(Object o);

    //@ pure
    abstract boolean isSubset(/*@ non_null @*/ JMLObjectSet s2);

    //@ pure
    // overrides Object.equals()
    abstract boolean equals(Object s2);

    //@ pure
    JMLObjectSet() {}

    //@ pure
    abstract /*@ non_null @*/ JMLObjectSet insert(Object o);

    //@ pure
    abstract /*@ non_null @*/ JMLObjectSet remove(Object o);

    //@ pure
    int count() {
        return isEmpty() ? 0 : 1+remove(choose()).count();
    }

    /** a minimalistic iterator */
    /*@ public normal_behavior
      @ requires !isEmpty();
      @ ensures has(\result);
      @*/
    //@ pure
    abstract Object choose();
}
```

A.2 Improved Equational Specification

```
/*@ pure @*/ abstract class JMLObjectSet {

    //@ public constraint (\forall Object o; \old(has(o)) == has(o));
```

```
/*@ public static invariant (\forall JMLObjectSet s,s2; ;(
  @                    (\ forall  Object e1, e2; ;
  @                    equational_theory(s, s2, e1, e2))));
  @*/

/** An equational specification of the properties of sets.
 */
/*@ public normal_behavior
  @ {|
  @ ensures \result <==> (
  @          !(new JMLObjectSet()).has(e1)
  @     ) && (
  @          \allocated(s) ==>
  @               s.insert(e1).has(e2) == (e1 == e2 || s.has(e2))
  @     ) && (
  @               (new JMLObjectSet()).count() == 0
  @     ) && (
  @               \allocated(s) ==>
  @                 s.insert(e1).count() == (s.has(e1) ? s.count() : s.count
      () + 1)
  @     ) && (
  @               \allocated(s) && \allocated(s2) ==>
  @                 s.isSubset(s2) == (\forall Object o; ;(s.has(o) ==> s2.
      has(o)))
  @     ) && (
  @          \allocated(s) ==> (\forall Object o; \allocated(o); (
  @               s.equals(o) == (o instanceof JMLObjectSet)
  @                           && s.isSubset((JMLObjectSet)o)
  @                           && ((JMLObjectSet)o).isSubset(s)))
  @     ) && (
  @               \allocated(s) && \allocated(s2) ==>
  @                 s.equals(s2) ==>
  @                      s.count() == s2.count() && s.remove(e1).equals(s2.
      remove(e1))
  @     ) && (
  @               (new JMLObjectSet()).remove(e1).equals(new
      JMLObjectSet())
  @     ) && (
  @          \allocated(s) ==>
  @               !s.has(e1) ==> s.insert(e1).remove(e1).equals(s)
  @     ) && (
  @          \allocated(s) ==>
  @               e1!=e2 ==>
  @                    s.insert(e1).remove(e2).equals(s.remove(e2).insert(
      e1))
```

```
  @      ) && (
  @          \allocated(s) ==>
  @              s.isEmpty() == (s.count() == 0)
  @      );
  @ |}
  @
  @ implies_that     // other ways to specify some operations
  @
  @      ensures \result <==> (
  @          (new JMLObjectSet()).isEmpty()
  @      ) && (
  @          \allocated(s) ==>
  @              !s.insert(e1).isEmpty()
  @      );
  static public pure model boolean equational_theory(JMLObjectSet s,
                                                     JMLObjectSet s2,
                                                     Object e1,
                                                     Object e2);
  @*/

//@ pure
abstract boolean has(Object o);

//@ pure
abstract boolean isEmpty(Object o);

//@ pure
abstract boolean isSubset(/*@ non_null @*/ JMLObjectSet s2);

//@ pure
// overrides Object.equals()
abstract boolean equals(Object s2);

//@ pure
JMLObjectSet() {}

//@ pure
abstract /*@ non_null @*/ JMLObjectSet insert(Object o);

//@ pure
abstract /*@ non_null @*/ JMLObjectSet remove(Object o);

//@ pure
int count() {
    return isEmpty() ? 0 : 1+remove(choose()).count();
}
```

```
    /** a minimalistic iterator */
    /*@ public normal_behavior
      @ requires !isEmpty();
      @ ensures has(\result);
      @*/
    //@ pure
    abstract Object choose();
}
```

A.3 Method-Centric Specification

```
/*@ pure @*/ abstract class JMLObjectSet {
    // to do: express that has() depends on encapsulated state only.

    //@ public constraint (\forall Object o; \old(has(o)) == has(o));

    //@ pure
    abstract boolean has(Object o);

    /*@ public normal_behavior
      @ ensures \result == (\forall Object o; ! has(o));
      @*/
    //@ pure
    abstract boolean isEmpty();

    /*@ public normal_behavior
      @ ensures \result == (\forall Object o; has(o) ==> s2.has(o));
      @*/
    //@ pure
    abstract boolean isSubset(/*@ non_null @*/ JMLObjectSet s2);

    /*@ also public normal_behavior
      @ requires \allocated(s2);
      @ ensures \result == (s2 instanceof JMLObjectSet) && (\forall
            Object o;; has(o) <==> ((JMLObjectSet)s2).has(o));
      @*/
    //@ pure
    // overrides Object.equals()
    abstract boolean equals(Object s2);

    //@ pure
    JMLObjectSet() {}

    /*@ public normal_behavior
      @ ensures \result.isEmpty();
```

```
    @*/
  //@ pure
  abstract /*@ non_null @*/ JMLObjectSet empty();

  /*@ public normal_behavior
    @ ensures \result.has(o) && (\forall Object o2; o2!=o; \result.has(o2)
          == has(o2));
    @*/
  //@ pure
  abstract /*@ non_null @*/ JMLObjectSet insert(Object o);

  /*@ public normal_behavior
    @ ensures !\result.has(o) && (\forall Object o2; o2!=o; \result.has(o2
        ) == has(o2));
    @*/
  //@ pure
  abstract /*@ non_null @*/ JMLObjectSet remove(Object o);

  /** a minimalistic iterator */
  /*@ public normal_behavior
    @ requires !isEmpty();
    @ ensures has(\result);
    @*/
  //@ pure
  abstract Object choose();

  /*@ public normal_behavior
    @ ensures \result == (\sum Object o; has(o); 1);
    @*/
  //@ pure
  int count() {
      return isEmpty() ? 0 : 1+remove(choose()).count();
  }
}
```

A.4 Implemenation Specification

```
/*@ pure @*/ class JMLObjectSetImpl extends JMLObjectSet {
  private Object value;
  private JMLObjectSetImpl next;

  /*@ private invariant
    @ next==null || !next.has(value);
    @*/

  /*@ private normal_behavior
```

```
  @ ensures \result == (next!=null) && ((o==value) || next.has(o));
  @*/
//@ pure
boolean has(Object o) {
    return (next!=null) && ((o==value) || next.has(o));
}

boolean isEmpty() {
    return next==null;
}

boolean isSubset(/*@ non_null @*/ JMLObjectSet s2) {
    // this implementation could also go into the superclass' code
    JMLObjectSet s = this;
    while (!s.isEmpty()) {
        Object o = choose();
        if (!s2.has(o)) {return false;}
        s=s.remove(o);
    }
    return true;
}

boolean equals(Object s2) {
    return s2!=null
        && (s2 instanceof JMLObjectSet)
        && this.isSubset((JMLObjectSet) s2)
        && ((JMLObjectSet)s2).isSubset(this) ;
}

/*@ public normal_behavior
  @ ensures isEmpty();
  @*/
//@pure
JMLObjectSetImpl() {
    next=null;
}

/*@ private normal_behavior
  @ requires \allocated(n) && !n.has(v);
  @ ensures value=v && next==n;
  @*/
//@ pure
JMLObjectSetImpl(Object v, JMLObjectSetImpl n) {
    value=v;
    next=n;
}
```

```
/*@ non_null @*/ JMLObjectSet empty() {
    return new JMLObjectSetImpl();
}

/*@ non_null @*/ JMLObjectSet insert(Object o) {
    if (has(o)) {
        return this;
    } else {
        return insert_nonexisting(o);
    }
}

/*@ public normal_behavior
  @ requires !has(o);
  @ ensures \result.has(o) && (\forall Object o2; o2!=o; \result.has(o2)
        == has(o2));
  @*/
//@ pure
/*@ non_null @*/ JMLObjectSet insert_nonexisting(Object o) {
    return new JMLObjectSet(o,this);
}

/*@ non_null @*/ JMLObjectSet remove(Object o) {
    if (has(o)) {
        return remove_existing(o);
    } else {
        return this;
    }
}

/*@ public normal_behavior
  @ requires has(o);
  @ ensures !\result.has(o) && (\forall Object o2; o2!=o; \result.has(o2
      ) == has(o2));
  @*/
//@ pure
/*@ non_null @*/ JMLObjectSet remove_existing(Object o) {
    if (value==o) {
        return next;
    } else {
        return next.remove_existing(o).insert(value);
    }
}

public Object choose() {
```

```
            return value;
        }
    }
```