Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs

Steve M. Shaner, Gary T. Leavens, and David A. Naumann

TR #07-04b
March 2007, revised April, July 2007

Keywords: Model program, verification, specification languages, grey-box approach, higher order method, mandatory call, Hoare logic, refinement calculus.

2006 CR Categories:
D.2.1 [*Software Engineering*] Requirements/Specifications — languages, methodologies; D.2.4 [*Software Engineering*] Software/Program Verification — correctness proofs, formal methods, programming by contract; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, classes and objects, control structures, frameworks, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, logics of programs, pre- and post-conditions, specification techniques.

To appear in *OOPSLA 2007*.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Modular Verification of Higher-Order Methods with Mandatory Calls Specified by Model Programs

Steve M. Shaner Gary T. Leavens
Iowa State University, Ames, IA 50011 USA
{smshaner, leavens}@cs.iastate.edu

David A. Naumann
Stevens Institute of Technology, Hoboken, NJ 07030 USA
naumann@cs.stevens.edu

Abstract

What we call a "higher-order method" (HOM) is a method that makes mandatory calls to other dynamically-dispatched methods. Examples include template methods as in the Template method design pattern and notify methods in the Observer pattern. HOMs are particularly difficult to reason about, because standard pre- and postcondition specifications cannot describe the mandatory calls. For reasoning about such methods, existing approaches use either higher-order logic or traces, but both are complex and verbose.

We describe a simple, concise, and modular approach to specifying HOMs We show how to verify calls to HOMs and their code using first-order verification conditions, in a sound and modular way.

Verification of client code that calls HOMs can take advantage of the client's knowledge about the mandatory calls to make strong conclusions. Our verification technique validates and explains traditional documentation practice for HOMs, which typically shows their code. However, specifications do not have to expose all of the code to clients, but only enough to determine how the HOM makes its mandatory calls.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications — languages, methodologies; D.2.4 [*Software Engineering*]: Software/Program Verification — correctness proofs, formal methods, programming by contract; D.3.3 [*Programming Languages*]: Language Constructs and Features — classes and objects, control structures, frameworks, procedures, functions, and subroutines; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — assertions, logics of programs, pre- and post-conditions, specification techniques.

General Terms Languages, Verification

Keywords Model program, verification, specification languages, grey-box approach, higher order method, mandatory call, Hoare logic, refinement calculus.

1. Introduction

No program exists in a vacuum. Instead, developers use components from libraries and frameworks. For example, a Java programmer may use Swing, Java's input/output framework, and Jakarta Commons. Such reuse improves productivity. It can also improve other attributes of software, such as its performance or maintainability.

The importance of reusable OO components is both a challenge and opportunity for software engineering. It is an opportunity because better documentation of such components can payoff in productivity and quality.

In this paper we focus on some of reuse's technical challenges, namely the specification language design and verification challenges posed by higher-order methods. For us, the term *higher-order method* (*HOM*) means a method whose requirements include one or more mandatory calls. A *mandatory call* is a method call that must occur under certain specified conditions. The HOM's specification describes how it sequences these mandatory calls, and in what states these calls are made. A HOM may also make calls that are not mandatory.

Reasoning about HOMs is a long-standing hard problem [25, 42]. Our contribution is a practical technique that builds on the grey-box approach [7, 8, 9]. We show its practicality for sequential Java programs by integrating it with JML [24, 29], and by showing how to do modular reasoning simply. For verification of client code, we show how the use of a copy rule [33] in conjunction with grey-box specifications allows one to draw strong conclusions. Our verification techniques are explained using a Hoare logic, and we give a new soundness proof. Remarkably, refinement style reasoning is not needed to use the grey-box approach, though refinement is used in our soundness proof.

In what follows we first give more details on the problem. Our solution approach is described in Section 3. Section 4 formalizes our approach and gives a soundness proof. After that, we discuss other issues, future work, and conclude.

2. The Problem

Several standard and important examples of HOMs are found in common design patterns [18]. These include:

- The `Notify` method of the Observer pattern, which makes mandatory calls to the `Update` method in each observer object.
- The kind of method described by the Template Method pattern, which makes mandatory calls to several abstract "primitive operation" methods in some particular order.
- The `HandleRequest` method of the Chain of Responsibility pattern, which, if it cannot directly handle a request, makes a mandatory call to the next such method. (This illustrates that such mandatory calls need not happen in every execution, despite the name.)

In addition, clients of methods found in behavioral design patterns are often HOMs that make mandatory calls to the pattern's methods. This includes callers of: the `Interpret` method in the Interpreter pattern, the `Execute` method in the Command pattern, the `Handle` method in the State pattern, the `Accept` method in the Visitor pattern, and the strategy method in a strategy object.

As can be seen from the above examples, typically a mandatory call is both dynamically-dispatched and calls a method with a weak specification. A method specification is *weak* if it does not completely describe the state transformation that the caller of the HOM cares about, but instead only states some limited property (such as that a mandatory dynamic call will terminate, or that it does not write to certain fields). For example, in a Java implementation of the Observer pattern, the `actionPerformed` method of a Listener, which corresponds to the `Update` method of an observer, has such a weak specification, which allows Listener objects to perform many different tasks. The code sequencing calls to such methods thus has very weak dependencies on their effects. Mandatory calls will typically be dynamically dispatched, because they will typically be calls to abstract methods. For example, the call to the `actionPerformed` method of a Listener object will be dynamically dispatched.

2.1 Client Reasoning

Because the mandatory calls of a HOM typically have weak specifications, the HOM's specification will typically not be sufficient for client-side reasoning. That is, if a client wants to know that a call to a HOM accomplishes some specific state transformation, then the HOM's weak specification will generally not be enough to prove what the client wants.

As an example of this problem and of the problem of integrating with an interface specification language such as JML, we show a very simple instance of the Observer pattern. First, consider the class `Counter`, shown in Figure 1, whose HOM `bump` is to be observed, and which holds a single listener to observe it. This class declares two private fields, `count` and `lstnr`. The JML annotations declare both fields to be **`spec_public`**, meaning that they can be used in public specifications [26]. The field `count` is the main state in counter objects. The field `lstnr` holds a possibly null `Listener` object.[1] `Counter`'s `register` method has a Hoare-style specification. The precondition is omitted, since it is just "true." Its **`assignable`** clause gives a frame axiom, which says that it can only assign to the field `lstnr`. Its postcondition is given in its **`ensures`** clause. The figure does not specify the HOM `bump`, as a major part of the problem is how to specify such methods.

```
public class Counter {
  private /*@ spec_public @*/ int count = 0;
  private /*@ spec_public nullable @*/
    Listener lstnr = null;

  /*@ assignable this.lstnr;
    @ ensures this.lstnr == lnr;   @*/
  public void register(Listener lnr) {
    this.lstnr = lnr;
  }

  public void bump() {
    this.count = this.count+1;
    if (this.lstnr != null) {
      this.lstnr.actionPerformed(this.count);
    }
  }
}
```

Figure 1. A Java class with JML specifications. JML specifications are written as annotation comments that start with an at-sign (`@`), and in which at-signs at the beginnings of lines are ignored. The specification for method `register` is written before its header.

The interface `Listener`, specified in Figure 2, contains a very weak specification of its `actionPerformed` method. `Counter`'s `bump` method notifies a listener by calling `actionPerformed`. Its specification is weak because it has no pre- and postconditions. The only thing constraint on its actions is given by the specification's assignable clause. This clause names **`this.`**`objectState`, which is a datagroup defined for class `Object`. A datagroup is a declared set of fields that can be added to in subtypes [29, 31].

The class `LastVal`, specified in Figure 3 is a subtype of `Listener`. Objects of this type track the last value

[1] In JML fields are automatically specified to be non-null by default [11, 29], so **`nullable`** must be used in such cases.

```
public interface Listener {

    //@ assignable this.objectState;
    void actionPerformed(int x);
}
```

Figure 2. Specification of the interface `Listener`.

passed to their `actionPerformed` method in the field `val`. This field is placed in the `objectState` datagroup by the **`in`** clause following the field's declaration. Making `val` a member of the `objectState` datagroup allows the `actionPerformed` method to update it [29, 31]. Objects of this class also have a method `getVal`, which allows Java code to access the field's value.

```
public class LastVal implements Listener {
  private /*@ spec_public @*/ int val = 0;
  //@ in objectState;

  /*@ also
    @   assignable this.objectState;
    @   ensures this.val == x;   @*/
  public void actionPerformed(int x) {
    this.val = x;
  }

  //@ ensures \result == this.val;
  public /*@ pure @*/ int getVal() {
    return this.val;
  }
}
```

Figure 3. The JML specification of `LastVal`.

With these pieces in place, we can now show a typical example of client reasoning with the observer pattern. Consider the code in Figure 4. This code creates a `LastVal` object `lv` and a `Counter` object `c`. It passes `lv` to `c` by calling `c`'s `register` method. Hence, as the second assertion states, the `lstnr` field of the `Counter` object `c` holds `lv`. This sets the stage for calling `c`'s HOM `bump`.

```
    LastVal lv = new LastVal();
    //@ assert lv != null && lv.val == 0;
    Counter c = new Counter();
    c.register(lv);
    //@ assert c.lstnr == lv && lv != null;
    //@ assert c.count == 0;
    c.bump();
    //@ assert lv.val == 1;
```

Figure 4. A Java example that draws a strong conclusion (the assertion in the last line) about a call to the HOM `bump`.

The call to `bump` increments `c`'s count field to 1, and then passes 1 to `lv`'s `actionPerformed` method. This causes `lv` to store 1 in its field `val`, which makes the last assertion in Figure 4 hold. The problem we address is how to write modular specifications that enable modular and static verification of such assertions.

For proving the last assertion in Figure 4, a normal Hoare-style specification for `Counter`'s method `bump`, such as the one shown in Figure 5, is not sufficient. The problem

```
  //@ assignable this.count, lstnr.objectState;
  //@ ensures this.count == \old(this.count+1);
  public void bump();
```

Figure 5. A standard JML specification for `bump`.

with using Figure 5 to prove assertions like the last one in Figure 4 is that Figure 5 does not say anything about the particular state change that may occur in the `lstnr` object. Furthermore, a first-order specification like this has no way to even say that the mandatory call is made.

Proving such an assertion requires that the specification talk about the mandatory call and that there is some way to use the specification from `LastVal` to reason about that call. Thus `bump` must be specified so that the caller can use a specification like the one in `LastVal`, even though modularity prohibits `Counter` from knowing anything about `LastVal`.

2.2 Related Work

Several solutions to this problem of how to modularly reason about HOMs have appeared previously in the literature, albeit not dealing with OO issues like behavioral subtyping.

Ernst, Navlakha, and Ogden [15] use higher-order logic to handle such problems. As shown in Figure 6, one would use **`pre`** and **`post`** to refer to the pre- and postconditions of called methods.[2] However, this technique makes `bump`'s specification more complex and involved than its

```
/*@ requires this.lstnr != null
  @      ==> this.lstnr.actionPerformed
  @             .pre(this.count);
  @ assignable this.count, this.lstnr.objectState;
  @ ensures this.lstnr != null
  @     ==> (this.count == \old(this.count+1)
  @          && this.lstnr.actionPerformed
  @                .post(\old(this.count),
  @                      this.count));      @*/
public void bump();
```

Figure 6. A specification in the style of Ernst, *et al.* [15] for `bump`.

code. Furthermore, verification takes place in a higher-order logic, since the specification of `bump` takes a description of `actionPerformed` as a parameter, and thus quantifies over function or predicate symbols. This makes automated verification difficult, as most theorem provers for higher-order logic are interactive. Besides these shortcomings, their work does not technically require the function object to be called by an implementation, only that the effect specified by the parameter be achieved.

[2] Similarly, Damm and Josko [12] allow use of Hoare triples as predicates on procedure parameters.

Findler and Felleisen also use higher-order logic to describe higher-order contract checking [16]. Their contract language is able to express contracts for HOMs, including examples such as the Observer pattern. However, they focus on techniques for producing helpful error messages (blame assignment) and run-time checking of contracts. Thus, in comparison to our work, they have a more complex language for HOM contracts and they do not investigate static verifiction (such as how do draw strong conclusions about calls to HOMs).

Soundarajan and Fridella [42] solve the problem of making sure that a higher order method actually makes the mandatory calls by writing specifications that track a trace of method calls. For example, `Counter`'s `bump` method could be specified in their style as shown in Figure 7. In this figure, the trace, τ has one element, which is a call to `lstnr`'s `actionPerformed` method (which would have to be declared as a "hook method", hence the notation "$.hm$" for retrieving the name of this method from the trace). Using Soundarajan and Fridella's "Enrichment Rule" (R2), one can prove assertions like the last one in Figure 4, by using knowledge of the value of the field `lstnr`, and the specification in `LastVal`. However, writing such trace-based

$$
\begin{array}{l}
\textit{epre}.\texttt{Counter.bump()} \equiv [\tau = \epsilon] \\
\textit{epost}.\texttt{Counter.bump()} \equiv \\
\quad [(\textbf{this}.\texttt{lstnr} \neq \texttt{null}) \Rightarrow \\
\quad\quad ((|\tau| = 1) \\
\quad\quad\ \wedge\ \ (\tau[1].hm \\
\quad\quad\quad\quad = \textbf{this}.\texttt{lstnr.actionPerformed}))] \\
\quad \wedge\, [(\textbf{this}.\texttt{lstnr = null}) \Rightarrow \tau = \epsilon]
\end{array}
$$

Figure 7. A specification in the style of Soundarajan and Fridella [42] for `bump`.

specifications is still not very intuitive for programmers, especially when they involve sequencing several calls. Also, reasoning about such specifications involves intricate proofs about traces. For example, Soundarajan and Fridella's paper spends about 8 pages to describe a case study of specifying and verifying a single HOM (for bank accounts) [42, pages 321–329].

Büchi and Weck's "grey-box" approach [7, 8, 9], is a simpler way to specify such HOMs. We build on and adapt their work in this paper, integrating it with JML. In their work, specifications of HOMs are written as abstract programs, which in JML are called *model programs*. A model program exposes information about the method's mandatory calls, while hiding some details. Details can be hidden by using specification statements in the model program to describe the effect of the hidden code. As we will show, the resulting sequence of hidden behaviors and exposed mandatory calls allows variation in implementations while permitting clients to draw strong conclusions. Büchi and Weck also did not explain a practical technique for verifying that an implementation of a HOM satisfies a model program specification, nor did they give a verification rule for client reasoning. Our paper's contribution is a solution to these technical problems, a new soundness proof, and a practical adaptation to JML.

Barnett and Schulte [5] support run-time verification of model programs in the .NET environment. Their model programs are similar to ours in spirit, although expressed in AsmL. Their work, like ours, addresses the specification and checking of HOMs with mandatory calls. Their contribution is a technique for checking conformance of a running implementation, even when the specification may involve nondeterminism. For simplicity, in this paper we only consider model programs with limited syntactic support for nondeterminism, although it is present, due to the ability to write specification statements. Finally, instead of run-time checking we seek to provide static guarantees.

3. Solution Approach

Our solution approach relies on grey-box, model program specifications [7, 8, 9] and uses a copy rule [33] to reason about calls to HOMs specified with model programs.

A model program specification for `Counter`'s HOM `bump` is shown in Figure 8. In this figure, the **`public`** modifier says that this specification is intended for client use [26]. The keyword **`model_program`** introduces the model program. Its body contains a statement sequence consisting of a specification statement followed by an if-statement. The *specification statement* starts with **`normal_behavior`** and includes the **`assignable`** and **`ensures`** clauses. Specification statements can also have a **`requires`** clause, which would give a precondition; in this example the precondition defaults to "true." A specification statement describes the effect of a piece of code that would be used at that place in an implementation. Such a piece of code can assume the precondition and must establish the postcondition, assigning only to the datagroups permitted by its assignable clause. Thus specification statements can hide implementation details and make the model program less specific. Although

```
/*@ public model_program {
  @
  @  normal_behavior
  @    assignable this.count;
  @    ensures this.count == \old(this.count+1);
  @
  @  if (this.lstnr != null) {
  @    this.lstnr.actionPerformed(this.count);
  @  }
  @ }
  @*/
public void bump();
```

Figure 8. Model program that specifies the mandatory call to the `actionPerformed` method.

the example uses a specification statement in a trivial way, they can be used to abstract arbitrary pieces of code, and have been used to do so in the refinement calculus [2, 34].

Our approach prescribes how to do two verification tasks:

- Verification that a HOM implementation satisfies a model program specification. Our approach imposes verification conditions on the code by first "matching" the code against the model program, which yields a set of verification conditions for parts of the code that implement the model program's specification statements.
- Verification of calls to HOMs specified with model programs. Our approach uses a verification rule that copies the model program to the call site, with appropriate substitutions. The caller can then draw strong conclusions using a combination of the copied specification and the caller's knowledge of the program's state at the call site. In particular, at the site of the mandatory calls made by the substituted model program, the client may know more specific types of such calls' receivers. These more specific receiver types may have stronger specifications, which client reasoning can exploit.

3.1 Verifying Implementations

Verification of implementation code takes place in two steps.

The first step is matching, which checks whether code has the form specified by the model program. The matching we use in verifying that code satisfies a model program is simple, requiring exact matches except where the model program contains a specification statement. A specification statement can only be matched by a **refining** statement, which must have the same specification as the specification statement.

In our example, `bump`'s code in Figure 9 matches the model program in Figure 8. This is because the **refining** statement in the code matches the specification statement in the model program, and the call to `actionPerformed` in the code matches the same (mandatory) call in the model program. Thus each piece of the code matches a corresponding piece of the model program. Note that `bump`'s code in Figure 1 does not match, since it has no refining statement.

```
public /*@ extract @*/ void bump() {
  /*@ refining normal_behavior
    @   assignable this.count;
    @   ensures this.count == \old(this.count+1);
    @*/
  this.count = this.count+1;

  if (this.lstnr != null) {
    this.lstnr.actionPerformed(this.count);
  }
}
```

Figure 9. Code matching the model program specification for `Counter`'s `actionPerformed` method. The **extract** syntax is explained in Section 3.3.

The second stage is a proof that each refining statement in the code implements its specification. That is, one must check that, assuming the specification statement's precondition, the body of the refining statement achieves the specification's postcondition and only assigns to the fields permitted by its frame. Since all other matches are exact, this is sufficient to show that the code must refine the model program. It also ensures that the mandatory calls occur in the implementation in the specified states.

In our example, the specification statement has no precondition, and so one simply has to prove that the code's assignment `this.count = this.count+1` meets the postcondition and only assigns to `this.count`. This proof is straightforward.

Despite its simplicity, our technique is practical. In particular, it allows programmers to trade the amount of effort they invest in specification and verification for flexibility in maintenance. Programmers writing abstract specifications that hide some details gain the ability to change code that implements those specifications. Conversely, programmers can choose to avoid most of the overhead of specification and verification and simply use the code for a HOM as a (white-box) specification, with the obvious loss of flexibility in maintenance. The only details that our technique forces programmers to reveal are the mandatory calls for which client-side reasoning is to be enabled and the control structures surrounding such calls. For all other details the choice is left to them and is not dictated by our technique.

3.2 Client Reasoning

Our technique for verification of calls to HOMs with model program specifications, *client reasoning*, can reach strong conclusions without the use of higher-order logic or traces in specifications. As mentioned above, it uses a *copy rule* [33], in which the body of the model program specification is substituted for the HOM call at the call site, with appropriate substitutions.[3]

For example, to reason about the call to `c.bump()` in Figure 4, one copies the body of the model program specification to the call site, substituting the actual receiver `c` for the specification's receiver, **this**. This produces the code shown in Figure 10.

From the code shown in Figure 10 it is easy to verify the final assertion, since the call to `actionPerformed` is present. Thus the client can continue reasoning by using the **assignable** clause of the specification statement to show that, just before the call to `actionPerformed`, `c.lstnr == lv`. This allows the client to use the specification of `actionPerformed` from `Lastval` to prove the final assertion.

[3] The copy rule can be used repeatedly to verify recursive calls, as long as there is a way to limit the depth of recursive copying for each use. We do not provide a rule for determining such limits.

```
LastVal lv = new LastVal();
//@ assert lv != null && lv.val == 0;
Counter c = new Counter();
c.register(lv);
//@ assert c.lstnr == lv && lv != null;
//@ assert c.count == 0;
/*@ normal_behavior
  @   assignable c.count;
  @   ensures c.count == \old(c.count+1);
  @*/
if (c.lstnr != null) {
  c.lstnr.actionPerformed(c.count);
}
//@ assert lv.val == 1;
```

Figure 10. Result of substituting the model program's body for the call `c.bump()` from Figure 4.

The reason this approach works well for clients is that their reasoning does not have to rely on a weak, pre- and postcondition specification of the HOM or the very weak specification of its mandatory calls. Instead clients can use the model program and their knowledge of (stronger) specifications for the actual mandatory calls. Thus clients reasoning can use their knowledge of specific arguments to the HOM, or the states and types of objects, to draw strong conclusions.

3.3 Extraction of Model Programs from Code

Due to matching, model program specifications necessarily duplicate all of the implementation code that is not hidden by specification statements. This duplication introduces the possibility of errors and is a maintenance headache.

However, the ability to keep model program specifications separate from the code they specify is useful in two cases. The first is when there is no code, i.e., for an abstract method. The second is when the code cannot be changed at all, e.g., when the code is owned by a third party.

If the specification does not have to be kept separate from the code, we can avoid the problems of duplication by writing the code and the specification together. An example of how this would be done is shown in Figure 9. The method modifier **`extract`** says to extract the specification from the code. The extraction process forms a model program specification, in this case the one shown in Figure 8, by taking the specification of each **`refining`** statement as a specification statement in the model program (thus hiding its implementation part), and by taking all other statements as written in the code. This extracted model program automatically matches the code.

Figure 9's use of **`extract`** is syntactic sugar for writing the specification shown in Figure 8. The specification shown in Figure 8 would be what a specification browsing tool would show to readers, even if the specification was written in the code as in Figure 9.

3.4 Template Method Example

We have worked several nontrivial examples to validate our approach, and they worked beautifully. However, due to lack of space, we can only present one of these, an instance of the Template Method design pattern [18].

Template methods are HOMs that are used in frameworks, where they sequence calls to "hook methods" that are overridden (customized) by the framework's users. Typically the hook methods have weak specifications. The template method makes mandatory calls to these hook methods, which works very well with model program specification.

As an example, consider the HOM `prepare` in Figure 11. The model program specification extracted from the method `prepare` is shown in Figure 12. This model program has two mandatory calls to the weakly specified hook methods: one each to `mix` and `bake`.

```
import java.util.Stack;

public abstract class CakeFactory {
    public /*@ extract @*/ Object prepare() {
        Stack pan = null;

        /*@ refining normal_behavior
          @   assignable pan;
          @   ensures pan.isEmpty(); @*/
        pan = new Stack();

        this.mix(pan);
        this.bake(pan);
        return pan.pop();
    }

    //@ requires items.size() == 0;
    //@ assignable items.theCollection;
    //@ ensures items.size() == 1;
    public abstract void mix(Stack items);

    //@ requires items.size() == 1;
    //@ assignable items.theCollection;
    //@ ensures items.size() == 1;
    public abstract void bake(Stack items);
}
```

Figure 11. The class `CakeFactory`, with its template method `prepare`, and two hook methods: `mix` and `bake`.

```
    /*@ public model_program {
      @   Stack pan;
      @
      @   normal_behavior
      @     assignable pan;
      @     ensures pan.isEmpty();
      @
      @   this.mix(pan);
      @   this.bake(pan);
      @   return pan.pop();
      @ }                     @*/
    public Object prepare();
```

Figure 12. The extracted specification for `prepare`.

A specializer, like `StringyCake` in Figure 13, supplies code and stronger specifications for the hook methods.

```
import java.util.Stack;

public class StringyCake extends CakeFactory {

    /*@ also
      @   requires items.size() == 0;
      @   assignable items.theCollection;
      @   ensures items.size() == 1
      @        && items.peek().equals("batter");
      @*/
    public void mix(Stack items) {
        items.push("batter");
    }

    /*@ also
      @   requires items.size() == 1
      @        && items.peek().equals("batter");
      @   assignable items.theCollection;
      @   ensures items.size() == 1
      @        && items.peek().equals("CAKE");
      @*/
    public void bake(Stack items) {
        items.pop();
        items.push("CAKE");
    }
}
```

Figure 13. `StringyCake`, a subclass of `CakeFactory`. The keyword **`also`** indicates that the given specification is joined with the one it overrides [23, 27].

A client of `StringyCake` would be able to use the model program specification of `prepare` and the specifications of the hook methods to prove the assertion in Figure 14. This works because the client can substitute the model program specification for the call to `prepare`, which allows use of the extended specifications for the hook methods.

```
CakeFactory c;
Object r;
c = new StringyCake();
r = c.prepare();
//@ assert r.equals("CAKE");
```

Figure 14. Client code that uses `prepare`.

The result of substituting the actuals into the model program from Figure 12 for the call to the `prepare` method is shown in Figure 15. In this substitution, we have changed the return in the code into the assignment to the variable receiving the call's value, as usual [41]. From Figure 15, it is straightforward to prove the final assertion, taking advantage of the fact that `c` is an instance of `StringyCake` and the specifications of its hook methods.

4. Formalization of Reasoning with Model Programs

This section formalizes the two kinds of verification described above: verification of the correctness of methods against model program specifications and verifications of calls to such methods. To precisely investigate their soundness, we first give details of the subset of JML we study, and then formalize matching and the Hoare logic for this language.

```
CakeFactory c;
Object r;
c = new StringyCake();
{
  Stack pan = null;

  normal_behavior
    assignable pan;
    ensures pan.isEmpty();

  c.mix(pan);
  c.bake(pan);
  r = pan.pop();
}
//@ assert r.equals("CAKE");
```

Figure 15. Client code that uses `prepare`, after using the copy rule and substituting the actual receiver `c` for **`this`**.

4.1 Model Program Language

We study a subset of Java enriched with a subset of JML specification constructs. Except for model program specifications, this subset is essentially that of Core JML [28], which has classes and interfaces. Classes can declare fields and methods; we do not consider JML's model fields or invariants. Model fields could be simulated with JML's specification-only **`ghost`** fields, which Core JML handles, but we omit them from this paper's formal treatment, as they are orthogonal to our main concerns; likewise for interfaces. The remainder of this paper focuses on methods and their specifications.

Our subset of JML allows two kinds of method specifications. A method may have either a Hoare-style pre- and postcondition specification or a model program specification. For simplicity, we ignore frame axioms (JML's assignable clause) in the formalism (they can be encoded in postconditions), and we concentrate on partial correctness.

A grammar for the subset of JML model programs that we formalize is shown in Figure 16. Examples, such as the one in Figure 12 were shown with JML-style annotation comments and the full JML syntax, but our formalism ignores these lexical details as well JML's visibility modifiers for method specifications. We use the abbreviation $\mathbf{spec}(P, Q)$ for a JML specification statement with precondition P and postcondition Q. We also write $\mathbf{refining}\ P, Q\ \mathbf{as}\ S$ for a refining statement with precondition P and postcondition Q, and body S.

Statements $(S, \mathbb{S})$ in Figure 16 include both the statements that can appear in model programs and the code statements that can appear in method bodies. However, we do not allow refining statements to appear in model programs and we do

MD ::= "method declarations"
$\quad MS$ **void** $m\,(\overrightarrow{T\,x})\ S$

MS ::= "method specification"
$\quad$ **requires** P; **ensures** Q; "requires-ensures"
$\quad$ **model_program** $\mathbb{S}$ "model program"

$S, \mathbb{S}$ ::= "JML statement"
$\quad$; "skip"
$\quad | \ x = E;$ "assignment"
$\quad | \ x\,.\,m\,(\vec{x})\,;$ "method call"
$\quad | \ \{D\ S\}$ "block"
$\quad | \ S_1\ S_2$ "sequence"
$\quad |$ **if** $(x)\ S_1$ **else** S_2 "if"
$\quad | \ \mathbf{spec}(P, Q)$ "specification"
$\quad |$ **refining** P, Q **as** S "refining"

D ::= "declaration"
$\quad T\ x;$ "local of type T"
$\quad | \ D^*$ "simultaneous"

E, P, Q ::= "expression"
$\quad$ **this** $| \ x \ | \ x_1\,.\,x_2 \ | \ x_0\,.\,m\,(\vec{x}) \ |$ **new** $T\,(\vec{x})$
$\quad | \ x_1 == x_2 \ | \ x_1 < x_2 \ | \ x_1$ && $x_2 \ | \ x_1$ '||' x_2
$\quad | \ !\ x \ | \ x_1 + x_2 \ | \ x_1 \,\%\, x_2 \ | \ \ldots$
$\quad |$ **old**(E)

Figure 16. A core JML grammar for model programs. The nonterminal T stands class names and primitive types, and x stands for identifiers.

not allow specification statements to appear in method bodies. Refining statements are also not allowed in the bodies of refining statements.

We sometimes use $\mathbb{S}$ for statements in model programs, in contexts where S is used only for code. However, to avoid duplication of definitions, we do not technically define two different syntactic categories.

To avoid complications arising from specifications that may fail to terminate, method calls and new object constructions are not allowed in requires and ensures clauses.

4.2 Structural Similarity from Matching

Matching an implementation against a model program is straightforward. The *matches* predicate merely checks that the code could have had the model program extracted from it, as would be done if the **extract** keyword had been used. So *matches* is defined using an operator *extract*, which takes a statement and recursively replaces each **refining** statement with the specification statement it contains. This (very simple) algorithm is given in Figure 17.

As can be seen in Figure 17, statements match only against themselves, with the exception of statements that contain refining statements, which can only match specification statements. For example, the statement that forms the

$\mathit{matches} : S \times \mathbb{S} \to \mathit{Boolean}$
$\mathit{matches}(S, \mathbb{S}') = (\mathit{extract}(S) = \mathbb{S}')$

$\mathit{extract} : S \to \mathbb{S}$
$\mathit{extract}(\textbf{refining}\ P, Q\ \textbf{as}\ S) = \mathbf{spec}(P, Q)$
$\mathit{extract}(\{\,D\ S\,\}) = \{\,D\ \mathrm{extract}(S)\,\}$
$\mathit{extract}(S_1\ S_2) = \mathit{extract}(S_1)\ \mathit{extract}(S_2)$
$\mathit{extract}(\mathbf{if}\ (E_t)\ S_t\ \mathbf{else}\ S_f)$
$\quad = \mathbf{if}\ (E_t)\ \mathrm{extract}(S_t)\ \mathbf{else}\ \mathit{extract}(S_f)$
$\mathit{extract}(S) = S$ (if none of the above cases apply)

Figure 17. Definition of when a code statement matches the statement of a model program, which is built on a definition of how to extract a model program from code. The "=" notation used to compare statements means textual equality.

body of the `bump` method shown in Figure 9 matches the model program given in Figure 8.

This definition of *matches* allows specification statements to match themselves. However, because specification statements do not appear in normal code, but only in model program specifications, this does not matter for matching.

4.3 Verification and its Soundness

In this section we formalize our verification technique in the manner of Hoare logic. Section 4.3.1 gives proof rules, focusing on the rule for client reasoning and the rule for verifying method implementations with respect to model program specifications. In order to justify these rules with actual behavior of programs, we first define a semantics for ordinary statements (without specification statements) as state transformers (Section 4.3.2). Then we define a predicate transformer semantics for the language extended with specification statements (Section 4.3.3). Finally, Section 4.3.4 uses the semantics to define notions of satisfaction and Section 4.3.5 shows soundness of the proof rules for statements and for complete programs.

Notations and judgments used in the formalism are summarized in Figure 18.

4.3.1 Verification Logic

As in our Core JML formalism [22, 28], we let Γ range over typing assignments, which are maps from variable names to types. The judgment $\Gamma \vdash S$ says that S is well formed in the context Γ and the class table CT. The class table CT is implicit, and can be thought of as a compiled version of the program's class-level declarations. In particular, for a method in class T with parameters $\vec{x} : \vec{U}$, its body will be type checked in context **this** $: T, \vec{x} : \vec{U}$. (We only consider void-returning methods, for simplicity.)

Hoare triples are found in judgments of the following form $\Gamma \vdash P\ \{S\}\ Q$. Such a judgment means that if P holds, and if the statement S terminates normally, then Q is true in its post-state. Such a judgment is well-formed if

CT	class table (declarations)
Γ	type assignment
$\Gamma \vdash S$	S is well formed
$\Gamma \vdash E : T$	E has type T
$T' \leq T$	T' is subtype of T
$mtype(T, m)$	type of method m in class T
$\Gamma \vdash P\ \{S\}\ Q$	Hoare triple is derivable
ST	specification table
$re(P, Q)$	requires-ensures method spec.
$mp(\mathbb{S})$	model program method spec.
$\Gamma \vdash S$ **sat** $ST(T, m)$	S satisfies $T.m$'s spec.
$matches(S, \mathbb{S})$	S matches model program $\mathbb{S}$
$CT \vdash ST$	declarations provably correct
μ	method environment
$[\![\Gamma \vdash S]\!]$	state transformer semantics
$\{\!|\Gamma \vdash \mathbb{S}|\!\}$	predicate transformer semantics
wlp	weakest liberal precondition
$\sigma\!\downarrow\!\varphi$	$\{\sigma' \mid (\sigma, \sigma') \in \varphi\}$
$\mu \models P\ \{\mathbb{S}\}\ Q$	$P\ \{\mathbb{S}\}\ Q$ is valid in μ
$\mu \models ST$	ST is valid in μ
$f \sqsubseteq g$	f is refined by g

Figure 18. Summary of notations and judgment forms. The type assignment Γ is sometimes omitted if clear from context, e.g., $[\![S]\!]$ instead of $[\![\Gamma \vdash S]\!]$.

$\Gamma \vdash S$, and if the precondition P is also well formed in Γ, and if the postcondition Q is also well formed. Well-formedness of Q allows its "`\old`" expressions to refer to the statement's initial state. (For Q used as postcondition in a method specification, occurrences of **`this`** and of the parameters are implicitly treated as if inside "`\old`" so they have a sensible interpretation at invocation sites.) The formal rules apply only to well formed correctness statements, but for brevity we sometimes we omit the context Γ.

There is some variation among logics and verification systems about how the heap is modeled, and this affects the rules for field update and object construction (at least). For example, the Jive system [35, 40] uses an explicit global variable that stands for the heap, and the field assignment rule uses an update expression for the heap; ESC/Java [17] and Spec# [4] encode the heap as a collection of arrays, one per field, and treat field assignment as array update; de Boer and Pierik [13, 39] use another approach and Parkinson [38] yet another (separation logic). Our results do not depend on such particulars of the assertion language or proof rules, with the exception of the rules for method invocation and the rules for verifying method implementations. We refrain from stating a rule for field update and also omit the standard rules [20] for assignment, control structures, consequence, etc.

The key rules appear in Figure 19, which we explain below.

Specification statements do not occur in method implementations but they occur in model programs and therefore

(SPEC STMT)

$$\frac{}{\Gamma \vdash P\ \{\mathbf{spec}(P, Q)\}\ Q}$$

(REF)

$$\frac{\Gamma \vdash P\ \{S\}\ Q}{\Gamma \vdash P\ \{\mathbf{refining}\ P', Q'\ \mathbf{as}\ S\}\ Q}$$

(MCALL)

$$\frac{\begin{array}{c} ST(T', m) = re(P', Q') \\ \Gamma \vdash x : T \qquad T' \leq T \qquad mtype(T', m) = \vec{y} : \vec{T} \to \mathtt{void} \\ P = P'[x, \vec{x}/\mathtt{this}, \vec{y}] \qquad Q = Q'[x, \vec{x}/\mathtt{this}, \vec{y}] \end{array}}{\Gamma \vdash P\ \mathtt{\&\&}\ x\ \mathtt{instanceof}\ T'\ \{x\mathtt{.}m(\vec{x})\mathtt{;}\}\ Q}$$

(HOCALL)

$$\frac{\begin{array}{c} ST(T', m) = mp(\mathbb{S}') \\ mtype(T', m) = \vec{y} : \vec{T} \to \mathtt{void} \qquad \mathtt{this} : T', \vec{y} : \vec{T} \vdash \mathbb{S}' \\ T' \leq T \qquad \mathbb{S}' \text{ does not assign to } \vec{y} \\ \mathbb{S} = \mathbb{S}'[x, \vec{x}/\mathtt{this}, \vec{y}] \qquad \Gamma, x : T' \vdash P\ \{\mathbb{S}\}\ Q \end{array}}{\Gamma, x : T \vdash P\ \mathtt{\&\&}\ x\ \mathtt{instanceof}\ T'\ \{x\mathtt{.}m(\vec{x})\mathtt{;}\}\ Q}$$

(CLASS TABLE)

$$\frac{\begin{array}{l} \text{for all } (T, m) \text{ in } CT\text{:} \\ \quad \text{if class } T \text{ declares } m \text{ with body } S \\ \quad \text{and } mtype(T, m) = \vec{y} : \vec{T} \to \mathtt{void} \\ \quad \text{then } \mathtt{this} : T, \vec{y} : \vec{T} \vdash S\ \mathbf{sat}\ ST(T, m) \end{array}}{CT \vdash ST}$$

(SAT RE)

$$\frac{ST(T, m) = re(P, Q) \qquad \Gamma \vdash P\ \{S\}\ Q}{\Gamma \vdash S\ \mathbf{sat}\ ST(T, m)}$$

(SAT MP)

$$\frac{\begin{array}{c} ST(T, m) = mp(\mathbb{S}) \qquad matches(S, \mathbb{S}) \\ \text{for all } (\Gamma' \vdash \mathbf{refining}\ P, Q\ \mathbf{as}\ S') \text{ in } S\text{: } \Gamma' \vdash P\ \{S'\}\ Q \end{array}}{\Gamma \vdash S\ \mathbf{sat}\ ST(T, m)}$$

Figure 19. Selected proof rules. Subtyping is written $\leq$.

in the antecedent of rule (HOCALL) so our logic needs a rule for them. Rule (SPEC STMT) is straightforward [2, 34].

Rule (MCALL) is an ordinary rule for method call.[4] It is similar to the rule (SPEC STMT) in that the pre- and postcondition are obtained directly from the specification given by the program's specification table, ST.

The specification table, ST, is fixed for a given program. It maps a pair consisting of a class type and a method name to that method's specification. The specification ta-

[4] Some logics use more complicated "adaptation rules" for recursive procedure calls [21, 37], but adaptation is needed for auxiliary variables which we omit in favor of the "`\old`" notation. So, for reasoning about recursive calls, the substitutions in rule (MCALL) suffice.

ble contains two kinds of specifications: pre-/post-condition (requires/ensures) pairs, written $re(P, Q)$, and model program specifications, written $mp(\mathbb{S})$. The (MCALL) rule is only used for calls where the specification table contains a pre-/postcondition pair.

In JML, occurrences of parameters and **this** in a post-condition Q' are interpreted to refer to the initial state, which is why straightforward substitutions account for parameter passing in rule (MCALL). Note that the assumed type T' of the receiver must be a subtype of the static type of x, otherwise the precondition would be ill formed. When $T' = T$, the condition x **instanceof** T' is trivially true and can be omitted. (Note that in Java $mtype(T', m) = mtype(T, m)$.)

Of course, rule (MCALL) relies on behavioral subtyping [1, 32]: the method implementation dispatched according to the receiver's dynamic type must satisfy the specifications of its supertypes [14, 23, 27, 30].

ASSUMPTION 4.1. *The specification table, ST, has* behavioral subtyping*, i.e., for all classes T, T' with $T \leq T'$, and all methods m declared or inherited in T', $ST(T, m)$ refines $ST(T', m)$.*

In Section 4.3.4 we formalize the notion of refinement with which the assumption can be made precise, but this is not a central issue in this paper and we do not dwell on it.

In the (HOCALL) rule, the antecedent **this** $: T', \vec{y} : \vec{T} \vdash \mathbb{S}'$ says that the model program's body $\mathbb{S}'$ type checks in a type context appropriate for the method's type.[5] This ensures that the substitution $\mathbb{S}'[x, \vec{x}/\textbf{this}, \vec{y}]$ that produces $\mathbb{S}$ is well-typed. (Substitution renames locals to avoid capture.) Note that the antecedent $\Gamma, x : T' \vdash P \ \{\mathbb{S}\} \ Q$ uses the type context Γ with the receiver x's type changed to T'. This change of x's type is necessary because S' type checks with **this** $: T'$. Assignments to parameters are disallowed, without loss of generality, as usual in proof systems.

Rules (MCALL) and (HOCALL) are to be used for reasoning about both invocations by clients and recursive invocations within method declarations. By contrast, some Hoare logics include a separate rule for verifying the implementation of a recursive procedure, which is allowed to assume the correctness of the procedure for recursive invocations. Since rules (MCALL) and (HOCALL) take for granted that the invoked method satisfies its specification, we also formalize the obligation to verify every method implementation, using three more rules.

We use the judgment $CT \vdash ST$ to signify that every method implementation in every class satisfies its specification. We say that $P \ \{S\} \ Q$ *is derivable* iff it has a proof using the rules discussed earlier, and $P \ \{S\} \ Q$ *is provable* iff it is derivable and moreover $CT \vdash ST$ can be derived using rule (CLASS TABLE)

[5] This is best viewed as a well-formedness condition on ST, and the same for assignment to $\vec{y}$. We include these in the rule only for clarity.

The judgment "S **sat** $ST(T, m)$" says that S satisfies its specification. (Its semantics is given later, by Definition 4.7.) In the case of ordinary requires/ensures specifications, the rule (SAT RE) requires that statement S is verified in the usual way. Here S could be any statement, but the only use for rule (SAT RE) is for method bodies in the antecedent of rule (CLASS TABLE). Note that in the derivation of $P \ \{S\} \ Q$ one can use the method call rules.

In case $ST(T, m)$ is a model program specification of the form $mp(\mathbb{S})$, rule (SAT MP) says that an implementation S must match $\mathbb{S}$ and for each sub-statement in S of the form **refining** P, Q **as** S', the statement S' must satisfy specification P, Q. In the rule, each typing context Γ' is determined by the surrounding declarations.

Rule (REF) for the refining statement is a bit surprising. This rule ignores the refining statement's predicates, P' and Q', and instead only requires one to prove $P \ \{S\} \ Q$. The reason this rule can ignore P' and Q' is that execution of a refining statement just executes its body, S. However, if such a statement is used in a method with a model program specification, then a proof that S satisfies $P' \ \{S\} \ Q'$ will be required as part of the (SAT MP) rule.

4.3.2 State Transformer Semantics

To prove soundness of the Hoare logic described above, we need an independent semantics. We present a denotational semantics in this subsection. One reason for using such a semantics is that it is a good match for our Hoare logic, in which reasoning about method calls is based on their specifications. The denotational semantics is similarly compositional. For this purpose we adopt an existing denotational semantics of Java/JML [28], which is based on earlier work [3, 36].

In this section we let S range over method body (code) statements, excluding specification statements.

We write $State(\Gamma)$ for the set of program states appropriate for the typing context Γ. Each such state consists of a heap of objects together with a mapping of the variables declared in Γ to values. We consider only states that are well formed in the sense that every object reference that occurs as the value of a variable or in an object field is in the domain of the heap, and all field and variable values are type correct.

A statement S in context Γ denotes a *state transformer*, i.e., a (total) function from $State(\Gamma)$ to $State(\Gamma) \cup \{\perp\}$ where $\perp$ represents divergence or runtime error. Note that $\perp$ is not a state. Because S can invoke methods, its semantics is defined in terms of a *method environment* μ that gives the denotations of all methods on receivers of all classes. Ultimately we are interested in a particular method environment, namely the one, written $\widehat{\mu}$, denoted by the class table CT as described below.

The semantics of method call $x.m(\vec{x})$ first checks if the receiver x is null, in which case the result is $\perp$. Otherwise, the value of x is an object of some runtime type T. The state transformer denoted by the body of the method m declared or inherited in T is given by $\mu(T, m)$, and the semantics of the call simply applies this state transformer to a state where **this** is mapped to the value of x and parameters $\vec{y}$ are mapped to the values of $\vec{x}$.

The denotation of a statement S that type checks in context Γ, written $[\![\Gamma \vdash S]\!]$, is defined in terms of an arbitrary μ, so that $[\![\Gamma \vdash S]\!](\mu)$ is a function from $State(\Gamma)$ to $State(\Gamma) \cup \{\bot\}$. For given μ, the definition of $[\![\Gamma \vdash S]\!](\mu)$ goes by induction on the structure of S. It is entirely straightforward and most of the details are not relevant here. We define the semantics for "refining" statements as follows:

$$[\![\Gamma \vdash \textbf{refining } P, Q \textbf{ as } S]\!](\mu) = [\![\Gamma \vdash S]\!](\mu)$$

Recall that the specification part is only present to make the connection with a corresponding specification statement in a model program specification.

Specification statements only occur in model programs and their semantics is given in Section 4.3.3.

Some work is needed to define a method environment, $\widehat{\mu}$, that models the semantics of the class table. This is done by taking the least upper bound, in a straightforward ordering,[6] of a countable sequence of approximate method environments μ_i. For each i, the semantics $[\![\Gamma \vdash S]\!](\mu_i)$ accurately models an operational semantics for S in which the method call stack is bounded in depth by i, with outcome $\bot$ if the depth is exceeded. The limit, $\widehat{\mu}$, models an operational semantics with unbounded calling stack.

In detail, define μ_0 to interpret every method as the everywhere-$\bot$ function. For $i \geq 0$, construct μ_{i+1} as follows: for each class T and method named m with body S in T, the meaning of the method in μ_{i+1} is given by S in terms of μ_i, that is: $\mu_{i+1}(T, m) = [\![S]\!](\mu_i)$. In case m is inherited in T from some superclass T', $\mu(T, m)$ is defined to be $\mu(T', m)$. (Owing to this treatment of inheritance, the method call semantics can simply look up in the environment the method meaning associated with the dynamic type of the receiver.)

For any terminating computation, there is some finite maximum size of the calling stack, and this is reflected in the fact that for any S and any σ there is some i such that

$$[\![S]\!](\widehat{\mu})(\sigma) = [\![S]\!](\mu_j)(\sigma) \quad \text{for all } j \geq i \tag{1}$$

Hence if S is the body of m in T then $\widehat{\mu}(T, m) = [\![S]\!](\widehat{\mu})$.

We now define a semantics for Hoare triples. We write $Pred(\Gamma)$ for the powerset of $State(\Gamma)$, noting that $\bot$ is not a state and therefore not an element of any $\varphi \in Pred(\Gamma)$. For P well formed in Γ, written $\Gamma \vdash P$, we write $[\![\Gamma \vdash P]\!]$ for its denotation, which is an element of $Pred(\Gamma)$. This is consistent with the semantics $[\![\Gamma \vdash E]\!]$ of expressions; a boolean expression maps states to true or false, which can be seen as the characteristic function of a set. We assume that pre- and post-conditions never evaluate to $\bot$. The semantics of boolean expressions is straightforward, and our results do not depend on the particular syntax of assertions.

A postcondition Q can refer to both initial and final state, and we overload notation to write $[\![Q]\!]$ for the subset of $State(\Gamma) \times State(\Gamma)$ denoted by Q. We omit details, but recall in the case of method specifications, some desugaring is needed so that mention of **this** and method parameters in Q refer to the initial state, so that rule (MCALL) is sound.

DEFINITION 4.2 (valid triple). *Let μ be a method environment. Then $P\ \{S\}\ Q$* is valid for μ *iff*

$$\begin{array}{l}\textit{for all } \sigma, \sigma' \in State(\Gamma),\\ \textit{if } \sigma \in [\![P]\!] \textit{ and } \sigma' = [\![S]\!](\mu)(\sigma)\textit{, then } (\sigma, \sigma') \in [\![Q]\!].\end{array} \tag{2}$$

Note that this is partial correctness, because it imposes no constraint in case $[\![S]\!](\mu)(\sigma) = \bot$ since $\bot$ is not a state and σ, σ' range over states.

This definition is our touchstone, since it is the standard meaning of partial correctness specifications and it is based on a semantics that is essentially operational, in the sense that it describes directly how each construct manipulates the program states. However, this semantics is written in a denotational style.

4.3.3 Predicate Transformer Semantics

Definition 4.2 has one limitation: for client reasoning, formalized by the rule (HOCALL), we need Hoare triples for model programs. Since model programs may contain specification statements, a deterministic state transformer semantics would be insufficient. It is convenient to use predicate transformers which are a standard model for refinement calculi [2]. Ours is a weakest liberal precondition (wlp) semantics, for partial correctness.

Since the meaning of a specification statement is only needed during client reasoning, predicate transformer semantics is only needed at the level of statements, not at the level of method environments. Hence mutual recursion among methods does not need to be directly addressed in the predicate transformer semantics.

The predicate transformer semantics of $\mathbb{S}$ with respect to method environment μ is written $\{\![\Gamma \vdash \mathbb{S}]\!\}(\mu)$. We continue to interpret methods as state transformers, hence the use of the same kind of method environment as defined earlier.

For function $f : State(\Gamma) \rightarrow State(\Gamma) \cup \{\bot\}$, define the semantic weakest liberal condition function, $wlp(f) : Pred(\Gamma) \rightarrow Pred(\Gamma)$, by

$$\sigma \in wlp(f)(\varphi) \text{ iff } f(\sigma) = \bot \text{ or } f(\sigma) \in \varphi.$$

for all $\sigma \in State(\Gamma)$ and $\varphi \in Pred(\Gamma)$.

For all primitive statements S, other than specification statements, we define $\{\![\Gamma \vdash S]\!\}(\mu)$ by

$$\{\![\Gamma \vdash S]\!\}(\mu) = wlp([\![\Gamma \vdash S]\!](\mu)) \tag{3}$$

[6] State transformers are ordered, as usual, by the pointwise order on functions, with respect to the order with states incomparable and $\bot$ less than every state. That is, $f \leq g$ iff forall σ, if $f(\sigma) \neq \bot$ then $f(\sigma) = g(\sigma)$.

In particular, (3) applies in case S is a method invocation. This is why the predicate transformer semantics can be defined in terms of ordinary method environments, rather than storing predicate transformers in the method environment (as in [10]). Equation (3) turns out to hold for all S without specification statements, because we adopt the usual semantic definitions for control structures:

- $\{\!|\Gamma \vdash S_1; S_2|\!\}(\mu)(\varphi) = \{\!|\Gamma \vdash S_1|\!\}(\mu)(\{\!|\Gamma \vdash S_2|\!\}(\mu)(\varphi))$
- $\{\!|\texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2|\!\}(\mu)(\varphi)$
$= ([\![E]\!] \cap \{\!|S_1|\!\}(\mu)(\varphi)) \cup ([\![!E]\!] \cap \{\!|S_2|\!\}(\mu)(\varphi))$

As in the state transformer semantics, the meaning of a refining statement is given by its body:

$$\{\!|\Gamma \vdash \textbf{refining}\ P, Q\ \textbf{as}\ S|\!\}(\mu) = \{\!|\Gamma \vdash S|\!\}(\mu)$$

It remains to define the semantics for specification statements. Define, for all typing contexts Γ, for all μ, for all φ in $Pred(\Gamma)$, and for all σ and σ' in $State(\Gamma)$:

$$\begin{array}{rl} & \sigma \in \{\!|\Gamma \vdash \textbf{spec}(P, Q)|\!\}(\mu)(\varphi) \\ \text{iff} & \sigma \in [\![P]\!] \wedge (\forall \sigma' : (\sigma, \sigma') \in [\![Q]\!] \Rightarrow \sigma' \in \varphi) \end{array} \tag{4}$$

To define the meaning of Hoare triples we need one more technical ingredient. For each Γ, two-state predicate $\varphi \subseteq State(\Gamma) \times State(\Gamma)$, and $\sigma \in State(\Gamma)$, define $\sigma{\downarrow}\varphi$ by $\sigma{\downarrow}\varphi = \{\sigma' \mid (\sigma, \sigma') \in \varphi\}$.

DEFINITION 4.3. *For all well formed* $\Gamma \vdash P\ \{\mathbb{S}\}\ Q$*, where* $\mathbb{S}$ *may include specification statements, and every method environment* μ*, define* $\mu \models P\ \{\mathbb{S}\}\ Q$ *iff*

$$\forall \sigma \in State(\Gamma) : \sigma \in [\![P]\!] \Rightarrow \sigma \in \{\!|\mathbb{S}|\!\}(\mu)(\sigma{\downarrow}[\![Q]\!]).$$

In case Q does not depend on the initial state, so that we can consider $[\![Q]\!]$ to be a set of states rather than pairs, note that $\mu \models P\ \{\mathbb{S}\}\ Q$ is equivalent to $[\![P]\!] \subseteq \{\!|\mathbb{S}|\!\}(\mu)([\![Q]\!])$ —our rendering of the usual $P \Rightarrow \mathrm{wlp}(\mathbb{S}, Q)$.

LEMMA 4.4. *Suppose* S *contains no specification statements. Then* $\mu \models P\ \{S\}\ Q$ *if and only if* $P\ \{S\}\ Q$ *is valid for* μ*.*

This is a straightforward consequence of the relation (3) which can be shown to hold for all S.

4.3.4 Satisfaction for methods and method environments

The method call rules are only sound if the invoked method implementations satisfy their specifications. This subsection defines satisfaction for an individual method body —i.e., the semantics of the notation "S **sat** $ST(T, m)$" used in rules (SAT RE) and (SAT MP). This is closely related to satisfaction by all methods in the class table —i.e., the semantics of the notation $CT \vdash ST$ used in rule (CLASS TABLE).

The first step is to define the *refinement order* $\sqsubseteq$. For predicate transformers f, g over Γ we define

$$f \sqsubseteq g \iff \forall \varphi \in Pred(\Gamma) : f(\varphi) \subseteq g(\varphi).$$

The semantics of **spec** is justified by the following.

LEMMA 4.5. *For all* μ, P, S, Q*:*

$$\{\!|\textbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq \{\!|S|\!\}(\mu)\ \text{ iff }\ \mu \models P\ \{S\}\ Q.$$

The proof is straightforward.

The next notion expresses that each method in environment μ satisfies its specification.

DEFINITION 4.6. $\mu \models ST$ *iff for all* T *and all methods* m *declared or inherited in* T *with parameters* $\vec{y} : \vec{T}$*, and for* $\Gamma = \texttt{this} : T, \vec{y} : \vec{T}$*:*

- *if* $ST(T, m) = re(P, Q)$ *then*

$$\{\!|\Gamma \vdash \textbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq wlp(\mu(T, m));$$

- *if* $ST(T, m) = mp(\mathbb{S})$ *then*

$$\{\!|\Gamma \vdash \mathbb{S}|\!\}(\mu) \sqsubseteq wlp(\mu(T, m)).$$

The definition applies to any μ but consider what it means in case μ is the actual semantics $\widehat{\mu}$ of the program. That is, suppose $\widehat{\mu}(T, m)$ is the state transformer denoted by method body S. Then $wlp(\widehat{\mu}(T, m))$ is the same thing as $\{\!|S|\!\}(\widehat{\mu})$ —recall (1). So in the case that $ST(T, m)$ is a requires/ensures form $re(P, Q)$, the definition amounts to saying $\widehat{\mu} \models P\ \{S\}\ Q$ owing to Lemma 4.5.

A consequence of Assumption 4.1 is that $\mu \models ST$ implies

$$T' \leq T \Rightarrow wlp(\mu(T, m)) \sqsubseteq wlp(\mu(T', m) \tag{5}$$

Definition 4.6 is for the consequent of rule (CLASS TABLE) and the following for its antecedent.

DEFINITION 4.7. *Define* $\mu \models S$ **sat** $ST(T, m)$ *by cases:*

- *if* $ST(T, m)$ *is* $re(P, Q)$ *then* $\mu \models P\ \{S\}\ Q$
- *if* $ST(T, m)$ *is* $mp(\mathbb{S})$ *then* $\{\!|\mathbb{S}|\!\}(\mu) \sqsubseteq \{\!|S|\!\}(\mu)$

4.3.5 Soundness

This section proves the main results. The first, Theorem 4.12, says that if $CT \vdash ST$ is derivable using the rules, then $\widehat{\mu} \models ST$, where $\widehat{\mu}$ is the semantics of CT. The second, Corollary 4.13, says that the Hoare rules for statements are sound with respect to $\widehat{\mu}$ and the state transformer semantics.

The first steps are somewhat technical; the reader may skip to Theorem 4.9 on first reading.

It is easy to show that $wlp(f)$ is *monotonic*, in the sense that $\varphi \subseteq \psi$ implies $wlp(f)(\varphi) \subseteq wlp(f)(\psi)$, for any state transformer f. Moreover, the semantics for other constructs also yields monotonic predicate transformers, i.e., $\{\!|S|\!\}(\mu)$ is monotonic for all S and all μ.

A standard result in refinement calculus is that forms that combine statements (such as sequencing and **if**) are monotonic with respect to refinement [2]. For example, if $\{\!|S_1|\!\}(\mu) \sqsubseteq \{\!|S_1'|\!\}(\mu)$, then $\{\!|S_1 S_2|\!\}(\mu) \sqsubseteq \{\!|S_1' S_2|\!\}(\mu)$.

This case of $\sqsubseteq$-monotonicity depends on $\{\!|S_2|\!\}(\mu)$ being a monotonic predicate transformer.

To deal with the substitutions, we define semantic substitution for predicate transformers, in particular for those denoted by methods of some $mtype(T, m) = \vec{y}:\vec{T}\to\texttt{void}$. Note that for expressiveness in the programming language we do not require that arguments in method calls are distinct, which means we must consider non-injective substitutions, so we cannot just invert the substitution and apply that to the final state.

Let Γ be $\texttt{this}:T, \vec{y}:\vec{T}$ and let Γ' be $x:T, \vec{x}:\vec{T}$. Suppose f is a predicate transformer on $Pred(\Gamma)$ that is independent from the final values of $\texttt{this}, \vec{y}$, i.e., $f(\varphi) = f(\exists\texttt{this}, \vec{y}\cdot\varphi)$. Define $f[x, \vec{x}/\texttt{this}, \vec{y}]$ on $Pred(\Gamma')$ by $\sigma \in f[x, \vec{x}/\texttt{this}, \vec{y}](\varphi)$ iff

$$\sigma[x, \vec{x}/\texttt{this}, \vec{y}] \in f(\exists\texttt{this}, \vec{y}\cdot\varphi - x, \vec{x})$$

where we use an obvious notation for state substitutions and we write $\exists\texttt{this}, \vec{y}\cdot\varphi - x, \vec{x}$ for the Γ'-predicate obtained by dropping $x, \vec{x}$ from states and then adding all possible values for $\texttt{this}, \vec{y}$.

It is straightforward to show that this semantic substitution is monotonic and to prove the following.

LEMMA 4.8 (substitution). *Consider Γ, Γ' as just above. Suppose $\Gamma \vdash S$ and suppose S does not assign the variables in Γ (but may assign locals and the heap). Then, for any μ*

$$\begin{aligned}&\{\!|\Gamma' \vdash S[x, \vec{x}/\textbf{\textit{this}}, \vec{y}]|\!\}(\mu)\\ &= (\{\!|\Gamma \vdash S|\!\}(\mu))[x, \vec{x}/\textbf{\textit{this}}, \vec{y}]\end{aligned}$$

We omit the similar details for the semantic substitution operation on state transformers that do not assign $\vec{y}$, but note the connection for state transformer f on $State(\Gamma)$:

$$(wlp(f))[x, \vec{x}/\texttt{this}, \vec{y}] = wlp(f[x, \vec{x}/\texttt{this}, \vec{y}]) \qquad (6)$$

Now we can proceed to soundness of the statement rules. The modularity of reasoning in terms of specifications is reflected in the fact that they are sound with respect to every method environment that satisfies the specification table.

THEOREM 4.9 (Soundness of statement rules). *For any μ, if $\mu \models ST$ and $P\ \{S\}\ Q$ is derivable then $\mu \models P\ \{S\}\ Q$.*

Proof: by induction on derivation of $P\ \{S\}\ Q$. We omit proofs of the standard rules for assignment, etc.

For (MCALL) we must show

$$\mu \models P \;\texttt{\&\&}\; x \;\texttt{instanceof}\; T'\ \{x\,.\,m(\vec{x});\}\ Q$$

assuming the antecedents of the rule. As per Definition 4.3, consider any state σ in $[\![P \;\texttt{\&\&}\; x \;\texttt{instanceof}\; T']\!]$. By type soundness, the dynamic type of the receiver is some subclass T'' of T'. By an antecedent of the rule we have $ST(T', m) = re(P', Q')$. By hypothesis of the Theorem we have $\mu \models ST$. So by Definition 4.6 we have $\{\!|\mathbf{spec}(P', Q')|\!\}(\mu) \sqsubseteq wlp(\mu(T', m))$. So $\{\!|\mathbf{spec}(P', Q')|\!\}(\mu) \sqsubseteq wlp(\mu(T'', m))$ by behavioral subtyping, (5). To complete the argument that σ is in $\{\!|S|\!\}(\mu)(\sigma{\downarrow}[\![Q]\!])$, one unfolds the semantics of $x\,.\,m(\vec{x})$, which passes the argument values to $\mu(T'', m))$ in a way that matches the substitutions. We omit further details, which rely on substitution properties above, since the rule is not novel.

For rule (HOCALL) we must show

$$\mu \models P \;\texttt{\&\&}\; x \;\texttt{instanceof}\; T'\ \{x\,.\,m(\vec{x});\}\ Q$$

Because method call is a primitive and does not contain specification statements, we can prove the Hoare triple in the simpler form (2) as per Lemma 4.4. That is, for any σ we must show

$$\begin{aligned}&\text{if } \sigma \in [\![P]\!] \text{ and } [\![x\,.\,m(\vec{x})]\!](\mu)(\sigma)) \neq \bot\\ &\text{then } (\sigma,\ [\![x\,.\,m(\vec{x})]\!](\mu)(\sigma)) \in [\![Q]\!]\end{aligned} \qquad (7)$$

assuming the antecedents of rule (HOCALL). So suppose $ST(T', m) = mp(\mathbb{S}')$ and let $\mathbb{S} = \mathbb{S}'[x, \vec{x}/\texttt{this}, \vec{y}]$. From the key antecedent $P\ \{\mathbb{S}\}\ Q$, by induction on its derivation, we have $\mu \models P\ \{\mathbb{S}\}\ Q$. By Lemma 4.5 we get $\{\!|\mathbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq \{\!|\mathbb{S}|\!\}(\mu)$. For any T'' with $T'' \leq T'$ we have $\{\!|\mathbb{S}'|\!\}(\mu) \sqsubseteq wlp(\mu(T'', m))$ by behavioral subtyping and hypothesis $\mu \models ST$ of the Theorem. So by monotonicity of substitution we have

$$\begin{aligned}&\{\!|\mathbb{S}'|\!\}(\mu)[x, \vec{x}/\texttt{this}, \vec{y}]\\ &\sqsubseteq wlp(\mu(T'', m))[x, \vec{x}/\texttt{this}, \vec{y}].\end{aligned} \qquad (8)$$

(By the antecedent that $\mathbb{S}'$ does not assign to $\texttt{this}, \vec{y}$, its semantics $\{\!|\mathbb{S}'|\!\}(\mu)$ is independent from variables in the postcondition, so the semantic substitution is defined.) Now (8) is equivalent to

$$\{\!|\mathbb{S}|\!\}(\mu) \sqsubseteq wlp(\mu(T'', m))[x, \vec{x}/\texttt{this}, \vec{y}]$$

by Lemma 4.8 and definition of $\mathbb{S}$. Since we already proved $\{\!|\mathbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq \{\!|\mathbb{S}|\!\}(\mu)$ we get, for every $T'' \leq T'$, that $\{\!|\mathbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq wlp(\mu(T'', m))[x, \vec{x}/\texttt{this}, \vec{y}]$. Thus for each T'', unfolding the definition of $\sqsubseteq$, for all φ,

$$\begin{aligned}&\{\!|\mathbf{spec}(P, Q)|\!\}(\mu)(\varphi)\\ &\subseteq wlp(\mu(T'', m))[x, \vec{x}/\texttt{this}, \vec{y}](\varphi).\end{aligned} \qquad (9)$$

Finally let us show (7). Suppose $\sigma \in [\![P]\!]$. If $\sigma(x)$ is null, then by semantics $[\![x\,.\,m(\vec{x})]\!](\mu)(\sigma)$ is $\bot$ and we are done. Otherwise, $[\![x\,.\,m(\vec{x})]\!](\mu)(\sigma)$ is defined to be $\mu(T'', m)[x, \vec{x}/\texttt{this}, \vec{y}](\sigma)$ where T'' is the dynamic type of x in σ. We show that the consequent in (7) holds, provided that $\mu(T'', m)[x, \vec{x}/\texttt{this}, \vec{y}](\sigma) \neq \bot$.

Consider some $\sigma \in [\![P]\!]$ and instantiate (9) with $\varphi := (\sigma{\downarrow}[\![Q]\!])$. Using the semantics of $\mathbf{spec}(P, Q)$ and definition of $\sigma{\downarrow}[\![Q]\!]$, the left side of the inclusion contains σ, so

$$\sigma \in wlp(\mu(T'', m))[x, \vec{x}/\texttt{this}, \vec{y}](\sigma{\downarrow}[\![Q]\!]).$$

Rewriting by definition of *wlp* and using (6) we get

$$(\mu(T'', m)[x, \vec{x}/\texttt{this}, \vec{y}])(\sigma) \in (\sigma{\downarrow}[\![Q]\!]),$$

whence by definition of $\sigma{\downarrow}[\![Q]\!]$

$$(\sigma,\ (\mu(T'', m)[x, \vec{x}/\texttt{this}, \vec{y}])(\sigma)) \in [\![Q]\!].$$

LEMMA 4.10 (Soundness of (SAT RE)). *Let T be a class and m a method. Suppose $\mu \models ST$ and $ST(T, m) = re(P, Q)$. If the antecedents of rule* (SAT RE) *hold then $\mu \models S$ **sat** $ST(T, m)$.*

Proof: By premise of the rule we have $P\ \{S\}\ Q$ and so by Theorem 4.9 we get $\mu \models P\ \{S\}\ Q$, whence $\mu \models S$ **sat** $ST(T, m)$ by Definition 4.7 and Lemma 4.5.

THEOREM 4.11 (Soundness of (SAT MP)). *Let class T and method m be given. Let ST, and μ be such that $ST(T, m) = mp(\mathbb{S})$ and $\mu \models ST$. If the antecedents of rule* (SAT MP) *hold then $\mu \models S$ **sat** $ST(T, m)$.*

Proof: The premises of (SAT MP) are $match(S, \mathbb{S})$ and also $P\ \{S'\}\ Q$ for each "**refining** P, Q **as** S'" in S. So for each such S' we get, by Theorem 4.9, that $\mu \models P\ \{S'\}\ Q$. Thus by Lemma 4.5 we have $\{\!|\mathbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq \{\!|S'|\!\}(\mu)$. By definition of *matches*$(S, \mathbb{S})$, the only difference between S and $\mathbb{S}$ is that $\mathbb{S}$ may contain some specification statements $\mathbf{spec}(P, Q)$ that correspond, in S, to sub-statements of the form **refining** P, Q **as** S'. Recall that the semantics of **refining** P, Q **as** S' is just the semantics of S'. Hence, by using $\{\!|\mathbf{spec}(P, Q)|\!\}(\mu) \sqsubseteq \{\!|S'|\!\}(\mu)$ and $\sqsubseteq$-monotonicity of all program constructors we get $\{\!|\mathbb{S}|\!\}(\mu) \sqsubseteq \{\!|S|\!\}(\mu)$, whence $\mu \models S$ **sat** $ST(T, m)$ by Definition 4.7.

The preceding results give soundness of the rules for verifying statements and method bodies, under the assumption that a method environment μ is given such that $\mu \models ST$. The next result says that if every method body is verified, the assumption can be discharged by using the actual program semantics. Since methods can make recursive and mutually recursive calls, the proof resembles proofs of soundness for the recursive procedure rule in simple imperative languages.

THEOREM 4.12 (Soundness of rule (CLASS TABLE)). *Let $\widehat{\mu}$ be $[\![CT]\!]$. If $CT \vdash ST$ then $\widehat{\mu} \models ST$.*

Proof: We prove, by induction on i, that $\mu_i \models ST$ for every μ_i in the approximation chain that defines $\widehat{\mu}$. Hence by (1) the least upper bound also satisfies ST, i.e., $\widehat{\mu} \models ST$.

The base case $i = 0$ is trivial since each $\mu_0(T, m)$ is $\lambda\sigma.\bot$ which satisfies every specification.

For the induction step, suppose $\mu_i \models ST$. Now μ_{i+1} is defined using $[\![S]\!](\mu_i)$ for each method body S. We need to show that $[\![S]\!](\mu_i)$ satisfies $ST(T, m)$. By $CT \vdash ST$ according to rule (CLASS TABLE) we have S **sat** $ST(T, m)$, from either rule (SAT RE) or (SAT MP).

For the case of (SAT RE), we instantiate Lemma 4.10 to get $\mu_i \models S$ **sat** $ST(T, m)$ which is equivalent to $\{\!|spec(P, Q)|\!\}(\mu_i) \sqsubseteq \{\!|S|\!\}(\mu_i)$ and since the body S has no specification statements this is equivalent to

$$\{\!|spec(P, Q)|\!\}(\mu_i) \sqsubseteq wlp([\![S]\!](\mu_i)$$

which was to be proved.

For the case of (SAT MP), we instantiate Theorem 4.11 and the rest of the argument is similar.

COROLLARY 4.13. *If S has no specification statements and if $P\ \{S\}\ Q$ is provable, then it is valid in the state-transformer semantics.*

Proof: Recall that "provable" means $P\ \{S\}\ Q$ is derivable and in addition $CT \vdash ST$ is derivable. By the latter and Theorem 4.12 we have $\widehat{\mu} \models ST$. Using $P\ \{S\}\ Q$ and Theorem 4.9 we get $\widehat{\mu} \models P\ \{S\}\ Q$. By Lemma 4.4 we get satisfaction in terms of state transformers, i.e., (2) for $\widehat{\mu}$.

5. Discussion

5.1 Generality of our Approach

We formalized our approach as a Hoare logic, but it can be adapted to other ways of doing verification, for example, using verification conditions based on weakest preconditions, or using a refinement calculus.

It was a pleasant surprise that, although refinement underlies soundness of our technique, the grey-box approach can be deployed without need for explicit reasoning in the style of refinement calculi. Nor does our technique require features of JML other than specification statements and refining statements.

Adaptation to total correctness should be easy, beyond the inherent complication of measure functions for mutually-recursive, dynamically-dispatched methods.

An important extension is to take specification visibility into account. Rules (MCALL) and especially (HOCALL) could allow use of any specifications visible at the call site, e.g., those with protected visibility in the caller's and therefore not necessarily visible in the callee's class. This is easily added to our formalism and specification visibility is already well integrated in JML [26].

5.2 Specification Inheritance

To fit in with the rest of JML, our technique must not cause major disruptions in JML's semantics. In particular, model program specifications must fit in with JML's ability to give multiple specifications (what it calls specification cases) for each method. This feature is used as a way to explain specification inheritance for methods in JML [14, 23, 27].

However, viewing model program specifications as just another kind of specification case in JML seems to work well. Reasoning happens as follows. To verify an implementation, the programmer must show that the code satisfies each specification case that applies to that method, including all specification cases inherited from supertypes. Similarly, clients reasoning about a call can pick one of the available specification cases to use in reasoning about the call. That is, a simple interpretation of the meaning of specification inheritance (and joins) for model program specifications is that implementers must satisfy all of the given model program specifications and that clients can choose a particular model program specification when reasoning about a HOM call.

This idea could be formalized as follows. First, generalize the specification table ST so that $ST(T, m)$ returns a set of specifications. The set $ST(T, m)$ is the set of all specifications declared in T or some supertype of T for m; this gives meaning to specification inheritance for both model programs and requires/ensures specifications. Second, generalize the (HOCALL) and (MCALL) rules so that clients can use any $sc \in ST(T, m)$. Third, for verification of implementations, verify that the code satisfies each $sc \in ST(T, m)$. This would guarantee that all types are behavioral subtypes of each of their supertypes, and that supertype abstraction is valid [14, 23, 27].

If m has two model program specifications $\mathbb{S}_1$ and $\mathbb{S}_2$ and if there is no body S that matches both $\mathbb{S}_1$ and $\mathbb{S}_2$, then m cannot be correctly implemented. That is, specification inheritance may strengthen a specification so much that it becomes unsatisfiable. (This also happens with standard requires/ensures specifications.)

Previous work formalized the join of requires/ensures specification cases [14, 23, 27] using state predicates. The ability to use joined specification cases adds power to the proof system. However, it is not clear how to succinctly express the join of model program specification cases, unless they all have code that is identical except for having possibly different specification statements in the same places.

5.3 Verification of Implementations

Our approach uses simple syntactic matching for verification of implementations. Its simplicity allows us to focus on the big picture, making explanations of the ideas and soundness proof clear.

However, a disadvantage of our technique's use of simple syntactic matching is that it only allows the specification of mandatory calls when the surrounding control structures are also exposed. An example of this exposure is shown in Figure 20. The control structure around the call to `f` is simple enough that its model program must reveal the order of iteration or cannot identify the mandatory call at all.

While a more complex notion of matching could be used, it would have to rely on semantical (e.g., proof-based) techniques, such as those used in the refinement calculus [2, 34] or in program transformation.

To obtain more flexibility, one could generalize matching in various ways. One way would be to generalize the "patterns," that is, the model program specifications. For example, one could allow more constructs from the refinement calculus, such as nondeterministic **if** statements. Although

```
public class HashTable {  /* ... */

  private Object[] values = new Object[17];

  public /*@ extract @*/
  void applyToValues(IntFun s)
  {
     for (int i = 0; i < values.length; i++) {
       if (values[i] != null) {
         s.f(values[i]);
       }
     }
  }
}
```

Figure 20. A hash table with an `applyToValues` HOM.

specification statements can simulate many refinement calculus features, they may not be a good basis for partially specifying control structures.

Another way to obtain more flexibility would be to allow the refining statement to match other statements. Since the soundness of our technique relies on the semantic notion of refinement, any complementary notion of matching and refinement could work. However, then verification would require a real refinement calculus, whereas a key contribution of our approach is that use of model programs does not force the reasoner beyond Hoare logic.

5.4 More about Refining Statements

Refining statements themselves are interesting. Their design arose through discussions on the JML interest mailing list, although they are only partially implemented in the Common JML tools. Several tool builders want something like a refining statement to give frame axioms (JML's assignable clause) for arbitrary statements, particularly for loop statements [6, 19]. Although the refining statement in our formalism does not include JML's assignable clause, it is easy to add such clauses to a refining statement, as it is essentially a way to attach a specification case to a statement.

The idea of attaching a specification to a statement is remarkably flexible. For example, one can use a refining statement to replace loop invariant declarations. Suppose S is the body of a loop for which P is to be declared as its invariant. This can be done by writing

```
refining
  normal_behavior
    requires P; ensures P;
  S
```

in place of the body S. However, if this is done, it requires a slightly more complex notion of pattern matching, since some **`refining`** statements will appear in model programs.

5.5 Runtime Assertion Checking

In our approach, **`refining`** statements would be the focus of runtime assertion checking for higher order methods.

That is, to dynamically check that code implements a model program specification, it suffices to statically match the code against the specification, check the assignable clause, and then to dynamically check that: the precondition given in the refining statement's requires clause holds just before executing its body, and its ensures clause holds just after executing its body.

6. Future Work

Future work includes extending the theoretical and practical treatment of grey-box specifications in several directions. Frame axioms seem to work fine using the datagroup concept (see Section 2.1), so a formal treatment may be easy. Other extensions to our approach could include termination, or treating exceptions and concurrency.

7. Conclusions

Documentation of HOMs is an important problem [25, 42]. It occurs in connection with most behavioral design patterns, and is quite important for frameworks using such patterns. Besides the grey-box approach [7, 8, 9], we know of two other approaches to documenting such methods: writing specifications in higher-order logic [12, 15] and writing trace-based specifications [42]. Both of these techniques are difficult to use, especially informally. We are unaware of any use of the higher-order logic approach for OO programs.

On the other hand, the grey-box approach corresponds well to standard documentation practice, which presents the method's code, perhaps omitting some details. Our formalization of this technique explains its basic soundness and gives insight into how to use it more effectively. First, we have precisely explained how one should use this technique in client reasoning, by using a copy rule [33] to explain calls to methods with such specifications. (See rule (HOCALL) in Section 4.3.1.) In essence one copies the model program specification's code to the call site, and replaces formals with actuals.

Second, we have shown the soundness of the grey-box technique for suppressing details by writing specification statements. Programmers can use this idea informally, by using specification statements in their documentation to stand for pieces of hidden code. In doing so, programmers provide a specification for the hidden piece of code.

The examples presented show that by using a **`refining`** statement to mark pieces of code as implementing a specification statement, one can automatically extract grey-box specifications from code. This helps make specifying HOMs more practical.

Acknowledgments

Thanks to Hridesh Rajan, Neeraj Khanolkar, and the OOPSLA program committee for comments on earlier drafts. Thanks to the members of IFIP working group 2.3, with whom Leavens discussed these ideas, and who suggested the idea of extracting the model program specification from the code. Thanks to Robby, John Hatcliff, Erik Poll, and Wojciech Mostowski for additional discussions about this work. The work of Shaner and Leavens was supported in part by NSF grant CCF-0429567. The work of Naumann was supported in part by NSF grant CCF-0429894.

References

[1] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J. ACM*, 52(6):894–960, Nov. 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[5] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, Mar. 2003.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[7] M. Büchi. Safe language mechanisms for modularization and concurrency. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.

[8] M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zürich, September 1997, 1997. `http://tinyurl.com/2833tr`.

[9] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, Aug. 1999. `http://tinyurl.com/ywmuzy`.

[10] A. L. C. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language of refinement. *IEEE Transactions on Software Engineering*, 26(8):713–728, Aug. 2000.

[11] P. Chalin and F. Rioux. Non-null references by default in the Java Modeling Language. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05)*, volume 31(2) of *ACM Software Engineering Notes*. ACM, 2005.

[12] W. Damm and B. Josko. A sound and relatively complete Hoare-logic for a language with higher type procedures. *Acta Informatica*, 20(1):59–101, Oct. 1983.

[13] F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[14] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is ISU CS TR #95-20c, `http://tinyurl.com/s2krg`.

[15] G. W. Ernst, J. K. Navlakha, and W. F. Ogden. Verification of programs with procedure-type parameters. *Acta Informatica*, 18(2):149–169, Nov. 1982.

[16] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Oct. 2002.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37(5) of *SIGPLAN*, pages 234–245, New York, NY, June 2002. ACM.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[19] E. C. R. Hehner. Specified blocks. Verified Software: Theories, Tools, Experiments (VSTTE), Oct. 2005. `http://tinyurl.com/2a7kf2`.

[20] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, Oct. 1969.

[21] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium on Semantics of Algorithmic Languages*. Springer-Verlag, 1971.

[22] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 132–146, N. Y., Nov. 1999. ACM.

[23] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, Nov. 2006. Springer-Verlag.

[24] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[25] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, June 2007.

[26] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, May 2007.

[27] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Sept. 2006.

[28] G. T. Leavens, D. A. Naumann, and S. Rosenberg. Preliminary definition of Core JML. CS Report 2006-07, Stevens Institute of Technology, Sept. 2006.

[29] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Feb. 2007.

[30] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

[31] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, Oct. 1998.

[32] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[33] C. Morgan. Procedures, parameters and abstraction: separate concerns. *Science of Computer Programming*, 11(1), Oct. 1988. Reprinted in the book *On the Refinement Calculus*.

[34] C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[35] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[36] D. A. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLS*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226, 2005.

[37] E. Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Comput. Sci.*, 24:337–347, 1983.

[38] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. The author's Ph.D. dissertation.

[39] C. Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Universiteit Utrecht, 2006.

[40] A. Poetzsch-Heffter, P. Müller, and J. Schäfer. The Jive tool. `http://tinyurl.com/3cke34`, Apr. 2006. Checked August 2, 2006.

[41] P. V. Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass.,

2004.

[42] N. Soundarajan and S. Fridella. Incremental reasoning for object oriented systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 302–333. Springer-Verlag, 2004.