IOWA STATE UNIVERSITY
Digital Repository

Computer Science Technical Reports | Computer Science

3-2009

Modular verification of higher-order methods with mandatory calls specified by model programs

Steve M. Shaner
Iowa State University, smshaner@mac.com

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

 Part of the Programming Languages and Compilers Commons

Recommended Citation
Shaner, Steve M., "Modular verification of higher-order methods with mandatory calls specified by model programs" (2009). *Computer Science Technical Reports*. 312.
http://lib.dr.iastate.edu/cs_techreports/312

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Modular verification of higher-order methods with mandatory calls specified by model programs

Steve M. Shaner

TR #09-16
Initial Version: January 2009. Revised: March 2009.

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.

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

TABLE OF CONTENTS

LIST OF FIGURES iv

ABSTRACT v

CHAPTER 1. OVERVIEW 1

1.1 Introduction 1

1.2 The Problem 3

1.3 Our Solution 3

1.4 Contributions & Outline 6

CHAPTER 2. RELATED WORK 7

2.1 Solutions for Higher-order Methods 7

2.1.1 Higher-order Logic 7

2.1.2 Trace-based Semantics 8

2.1.3 Contracts in Scheme 8

2.2 Applications for Model Programs 9

2.2.1 Monitoring Runtime Behavior 9

2.2.2 Greybox Refinement 9

CHAPTER 3. SOLUTION APPROACH 10

3.1 Verifying Implementations 11

3.2 Client Reasoning 13

3.3 Extracting Implicit Model Programs from Code 14

3.4 Example Verifications 14

3.4.1 Template Methods: Following a Recipe 14

3.4.2 Chain of Responsibility: Testing Static Configurations 17

3.4.3 Technical Limitations 18

CHAPTER 4. EXTENDING JML WITH MODEL PROGRAMS 21

4.1 JML Background 21

4.2 Our Extension 22

4.2.1 The Model Program Specification Case 22

4.2.2 Implicit Model Programs via **`extract`** 22
4.2.3 **`refining`** Specification Statements 22
4.3 Design Implications 23
CHAPTER 5. FUTURE WORK & CONCLUSIONS 24
5.1 Future Work 24
5.2 Conclusions 25
BIBLIOGRAPHY 26

LIST OF FIGURES

Figure 1.1 One possible ecology of software genres. 1
Figure 1.2 A Java class with JML specifications. 4
Figure 1.3 Specification of the Listener interface. 5
Figure 1.4 Specification of the LastVal class. 5
Figure 1.5 Java code that draws a strong conclusion about HOM call `bump`. 5

Figure 2.1 Specification in the style of Ernst, *et al.* [9] for `bump`. 7
Figure 2.2 Specification in the style of Soundarajan and Fridella [22] for `bump`. 8
Figure 2.3 Greybox model programs (bottom) synthesize blackbox (left) and whitebox (right) specification styles. 9

Figure 3.1 Model program specifying the mandatory call to `actionPerformed`. . . . 10
Figure 3.2 Code matching the model program specification for Counter's mandatory call. 12
Figure 3.3 The result of substituting the model program's body for the call `c.bump()` from Figure 1.5. 13
Figure 3.4 Class CakeFactory with its template method `prepare`, and two hook methods. 15
Figure 3.5 `prepare`'s extracted specification. 15
Figure 3.6 Class StringyCake, a subclass of CakeFactory. 16
Figure 3.7 Client code that calls `prepare`. 16
Figure 3.8 Client code that calls `prepare`, after using the copy rule. 17
Figure 3.9 The Mailer interface. 18
Figure 3.10 An example mailing network connecting Alice to Bob. 18
Figure 3.11 Client code that makes an assertion of guaranteed message delivery. 19
Figure 3.12 Class Map implements a staple of functional programming in Java. 19
Figure 3.13 Client code that calls `map` while asserting its desired effect. 20
Figure 3.14 Code of Figure 3.13 after substituting a model program for `map`. 20

ABSTRACT

Formal specification languages improve the flexibility and reliability of software. They capture program properties that can be verified against implementations of the specified program. By increasing the expressiveness of specification languages, we can strengthen the argument for adopting formal specification into standard programming practice.

The higher-order method (HOM) is a kind of method whose behavior critically depends on one or more mandatory calls in its body. Programmers using HOMs would like to reason about the HOM's behavior, but revealing the entire code for such methods restricts writers of HOMs to a specific implementation.

This thesis presents a simple, intuitive extension of JML, a formal specification language for Java, that enables client reasoning about the behavior of HOMs in a sound and modular way. Furthermore, our particular technique is capable of fully automatic checking with lower specification overhead than previous solutions.

Supporting client reasoning about HOMs enables formal verification of some of the behavioral properties of HOM-using object-oriented design patterns, like Observer and Template Method. The technique also applies to specifying HOM behavior in any procedural language.

CHAPTER 1. OVERVIEW

This chapter introduces the reader to the ongoing project of formal software specification, exposes a current problem for client reasoning and develops an extension to specification languages that solves this problem. We close the chapter by identifying key contributions of this thesis and giving an outline of the content of subsequent chapters.

1.1 Introduction

All programs are written. As a collection of written artifacts, they form a body of literature for analysis. Classifying programs into genres of software is one way to study these writings. Depending on one's choice of perspective, many possible taxonomies might be used for classifying programs. We prefer an ecological perspective, since programs often interact with, consume and produce other programs. They share and compete for resources while constant development and user evaluation allow software to co- evolve over time. If one were to group programs according to their ecological roles, one might arrive at a system resembling Figure 1.1.

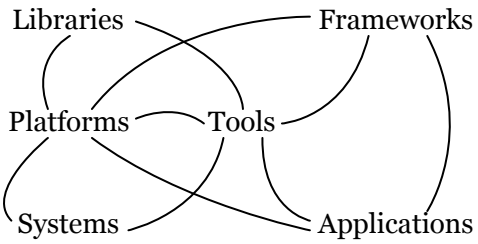

Figure 1.1 One possible ecology of software genres with interactions shown.

Applications, systems and platforms are the most visible software genres in such a taxonomy. Applications consume resources provided by platforms, while systems communicate with each other and are often composed of smaller sub-systems, platforms and tools. All three of these genres evolve by adopting or deploying frameworks and libraries. These latter genres function as a basic functional unit of the software ecology, whose size and complexity can range from a single function, script or object that performs a single task to near-turnkey solutions for a particular domain. The final program

category, the genre of tools, drives software development forward. Whether transforming between representations, editing source or interpreting bytecode, tools enable the construction and comprehension of modern software in every genre. These classifications are not meant to be authoritative, merely descriptive. Nor do we intend the boundaries between genres to be rigid and absolute. Many programs overlap multiple genres and can play ambiguous or shifting roles in the resulting ecology. The genres themselves have changed over time and will continue to change in the future. We provide this perspective to capture a snapshot of the present that addresses the variety of modern software.

Within the worldview of Figure 1.1, we consider the programmer whose job it is to straddle these genres. We would argue such a programmer represents the majority of today's software writers. For example, writers of libraries and frameworks must consider not only competing libraries and frameworks, but also the tools, applications and platforms with which their code may interact. Applications written using different tools behave differently, and smart programmers exploit these differences to improve the quality of their software. In every case where existing code is reused, both from within and outside of a development project, there must be an understanding of how the reused code works. Pragmatically, no code can be reused until programmers know how to call, link, compile or execute it. But behavioral descriptions go beyond this level of understanding. They allow programmers to reason about where, how and why the existing code will be reused. If this kind of reasoning is to be assisted by tools, then we need formal specifications to capture the relevant behavior.

As a genre, tools play a privileged role in our software ecology. A virtuous cycle exists in software evolution: improving support for formal specification in our tools increases the quality of reuse in the software created by those tools. Formal verification provides one way to observe this cycle in action. During analysis and design, specifications pose as models of the software to be created. Where tools are aware of them, these models can be checked for consistency with varying degrees of automation. During the development, testing and deployment of a program, specifications can act as pliable oracles for conformance. If programs fall short of the specified ideal, then either the specification or the program may be at fault and needing revision. In both cases, specification-aware tools enable programmers to improve their understanding of the software under inspection. Furthermore, after revisions are made, both specification and software have increased in value. Software performs according to the specification, and specifications describe software behavior for programmers seeking to reuse it.

Tools for writing and checking formal specifications have been developed for some time. Many effective specification conventions exist and current techniques to describe program behavior work well in most cases. As this thesis will show, however, some writers of software require more detail than current specification techniques provide. By providing an extension to the vocabulary of formal specification, we aim to bridge this gap. Software engineering advances insofar as the new specifications deliver more useful program properties to programmers at an acceptable cost. We aim to convince the reader that our work meets these criteria.

1.2 The Problem

As a supplement to conventional formal specification, we seek to specify the properties of mandatory calls made by higher-order methods. A *higher-order method* (or HOM) is any method whose behavior critically depends on one or more mandatory calls. A *mandatory call* is a method call that must occur within a particular calling context. In order to reason about the behavior of a particular HOM, we need to know both the identity of its mandatory calls as well as a sufficient description of the context in which the mandatory call will be made. Mandatory calls are useful because they enable structural patterns of code reuse and abstraction. However, in order to remain flexible, mandatory calls are often weakly-specified. We consider a method specification to be *weak* if it only states some limited property that does not completely describe the state transformation of interest to the clients of the HOM.

The calling structure of mandatory calls can be found in the actual implementation code, but current techniques for specifying functional behavior do not capture this structure sufficiently. Examples of such inadequacy can be found when considering the behavior of callbacks, supporting client reasoning for select object-oriented design patterns and also when testing an implementation for API or library conformance. Work on support for some object-oriented design patterns has been done by the author, with Leavens and Naumann in a paper appearing in OOPSLA 2007 [21] from which we adapt an example of client reasoning below.

Szyperski identified some specification problems with callbacks through a simple example using directories [23]. This is a specific design that invokes the Observer design pattern, where the `addEntry` method allows any number of directory observers to respond to the event after it occurred. Reasoning about calls to `addEntry` will require knowing both about how `addEntry` notifies those observers and what side effects will occur as the observers respond to notification.

Callbacks with this problematic behavior show up again and again in the context of other common object-oriented design patterns. Specifically, whenever a pattern delegates behavior inside of a method to some other call, that pattern calls for the creation of a higher-order method whose mandatory call will be weakly specified. Three such examples, one of which is introduced in the next section, will be explored in Chapter 3.

1.3 Our Solution

Generalizing from these examples, each involves a weakly-specified call whose occurrence must be verified inside some higher-order method. Current specification practice prefers to describe HOMs in terms of pre-/postcondition pairs, with possibly a frame axiom describing the set of transformed states. Preconditions capture what a method assumes to be true before it executes, and postconditions describe what is true after method execution. Frame axioms simply define what data might be changed

in the post-state. These concepts are not sufficient for our purposes, since clients often want to use their knowledge about the mandatory call to reason about the HOM's behavior. These issues are probably best explained using the following example from our OOPSLA 2007 paper [21].

Start by considering the class Counter, shown in Figure 1.2, 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 [14]. The field `count` is the main state in counter objects. The field `lstnr` holds a possibly null Listener instance.[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 1.2 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 Listener interface, specified in Figure 1.3, contains a very weak specification of its callback method, `actionPerformed`. Counter's `bump` method invokes this callback to notify the registered Listener object (if any). 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 [16, 17].

The LastVal class, specified in Figure 1.4 is a subtype of `Listener`. Objects of this type hold the last value passed to their `actionPerformed` method in the field `val`. This field is placed in

[1] In JML fields are automatically specified to be non-null by default [7, 16], so **`nullable`** must be used in such cases.

```
public interface Listener {

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

Figure 1.3 Specification of the Listener interface.

the `objectState` datagroup by the **`in`** clause following the field's declaration. Doing so allows the `actionPerformed` method to update it [16, 17]. Objects of this class also have a method `getVal` to allow other 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 1.4 Specification of the LastVal class.

```
    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 1.5 Java code that draws a strong conclusion about HOM call `bump`. The conclusion is the assertion in the last line.

With these pieces in place, we turn our attention to a typical example of client reasoning with the observer pattern in Figure 1.5. In the code, we set up a Counter object `c` with a registered observer `lv` and our client wants to be able to reason about the effect of calling the `bump()` method on `c`. The `bump()` method is informally known to invoke a method on `c`'s registered observer, but without formally revealing how that call is made, the strong conclusion of Figure 1.5 can't be verified. In this thesis, we argue that the best way to capture the missing information is found in the greybox approach.

Büchi and Weck define the greybox approach [3, 4, 5] as a technique for generating verification conditions that captures both the mandatory nature of these calls and the context in which they occur. Their basic contribution is the notion of a model program for revealing this information as a smaller trade-off in the level of abstraction of the specification. Model programs are considered to be greyboxes since they combine the blackbox (or obscured) nature of pre- and postconditions with the whitebox (or revealed) nature of exposing the code directly. The model program itself represents a sequential interleaving of these two paradigms that reads like an abstract description of the algorithm being specified. Where abstraction is preferred, one gives only a blackbox contract on the implementation. Where more detail is required (i.e. at the site of a mandatory call), one reveals the exact implementation as it must appear in the code. Model programs represent a combination of the finest level of detail that also grants some flexibility to implementors of the modeled method. The details of how model programs constrain HOM implementation can be found in Chapter 3.

Several solutions to this problem of how to modularly reason about HOMs have appeared previously in the literature, as well as some work on model programs in different contexts. Chapter 2 compares these attempts to our own.

1.4 Contributions & Outline

This thesis implements model programs for the Java Modeling Language (JML), a formal specification language for Java [13, 16]. To do so, we must provide what Büchi and Weck do not: their technique assumes that the structure of a model program is preserved by an implementation. This work gives a practical, though restrictive, algorithm for discharging that assumption among other claims.

In adapting the greybox approach to JML, this work makes the following contributions:

- a practical "pattern matching" algorithm for discharging the structure-preserving assumption of Büchi and Weck, and
- a design overview of the code that brings model program verification to JML.

This work proceeds as follows. Chapter 2 discusses related contributions, ending with Büchi and Weck's original formulation of greybox model programs. Chapter 3 goes into detail about our adaptation of the greybox approach with JML's model programs. Chapter 4 presents design details from the implementation of model programs in the JML Common Tools. Chapters 2 and 3 have been adapted from earlier material in our OOPSLA 2007 paper [21], while the material of Chapter 4 is original to this thesis. Chapter 5 presents paths for future work before drawing summary conclusions.

CHAPTER 2. RELATED WORK

This chapter examines the literature for existing solutions to the problem of higher-order methods as well as some applications for model programs. We wrap up this examination with a definition for greybox reasoning, which serves as a foundation for the solution proposed by this thesis.

2.1 Solutions for Higher-order Methods

Many other researchers have worked on the problem of higher-order methods using a variety of techniques. The first technique we will examine applies higher-order logic to parametrize specifications; the second reasons in terms of permitted traces of method calls.

2.1.1 Higher-order Logic

Ernst, Navlakha and Ogden [9] verify the effect of calling a HOM by allowing its specification to be parametrized. Specifically, the authors support assertions that represent the pre- and postconditions of a mandatory call, parametrized to reflect the context in which the higher-order method invokes it. Superficially, the assertions involving mandatory calls' pre- and post-states make specification longer and in some cases more obfuscated than the code specified. One such example can be found in Figure 2.1. These specifications are checked using higher-order logic during verification, to quantify

```
/*@ 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 2.1 Specification in the style of Ernst, *et al.* [9] for `bump`, from previous work [21].

over all possible mandatory calls. Automating the verification task is complicated by the interactive nature of most theorem provers for higher-order logic. Furthermore, mandatory calls must occur as part

of the behavior of a higher-order method. This technique only verifies which effects have occurred in the post-state, leaving clients to guess about behavioral dependencies.

2.1.2 Trace-based Semantics

Soundarajan and Fridella [22] use a trace-based semantics to verify the set of the calls made during any execution. The trace set that is produced is checked against the set of traces specified for the higher-order method. Figure 2.2 provides a demonstration of what such a specification might look like for our HOM `bump`.

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

Figure 2.2 Specification in the style of Soundarajan and Fridella [22] for `bump`, from previous work [21].

This solution requires that the correct calls are made from the desired states, but verification is complicated with the way by which the set of permitted traces is computed. Describing sequences of mandatory calls quickly adds to the complexity of these specifications. Specifiers are required to reason in terms of a higher-order logic that quantifies over all possible implementations. The contribution of this thesis should simplify how higher-order method specifications are written, used and verified.

2.1.3 Contracts in Scheme

Casting further afield, Findler and Felleisen [10] use assertion-style contracts on the function argument of a higher-order procedure in Scheme. Relative to our work, which focuses on client reasoning for the higher-order method, the authors seek to report contract violations where a function argument is misused. Their system allows blame assignment when the contract for a function argument of a higher-order procedure can be checked at runtime. This work generalizes first-order contract systems for those languages supporting first-class procedures. The extended contract system would be able to enforce calling constraints on function arguments passed to higher-order procedures, but do not specify information about when, where or if those argument procedures are invoked in the body of the higher-order procedure.

2.2 Applications for Model Programs

We are not the first to attempt to apply model programs to program specification. Other researchers have used model programs to enforce run-time constraints on implementations.

2.2.1 Monitoring Runtime Behavior

Barnett and Schulte [2] use model program specifications to construct execution monitors for reactive systems in the .NET environment. The authors write model programs using $AsmL$ to flexibly express nondeterministic compositions of mandatory calls. An algorithm to translate such expressions into automata for runtime verification is given. These efforts solve a different problem from the work contained in this thesis. Barnett and Schulte provide a solution for checking runtime behavior against a model program whereas we give static structural constraints on the implementation of HOMs. When we discuss future work in Chapter 5, we will consider some novel ideas for manipulating abstract statements inspired by this approach.

2.2.2 Greybox Refinement

Recall Büchi and Weck's "greybox" approach from the previous chapter. This work forms the primary inspiration for our own. As we mentioned in Chapter 1, the basic intuition here is that of Figure 2.3. Greybox model programs can be viewed as a sequential interleaving of blackbox and whitebox specifications. What is missing from previous work is a specified means to practically express these specifications that is also capable of verifying that implementations share a structure similar to their model programs. This thesis explores the consequences of our choices in bridging that gap.

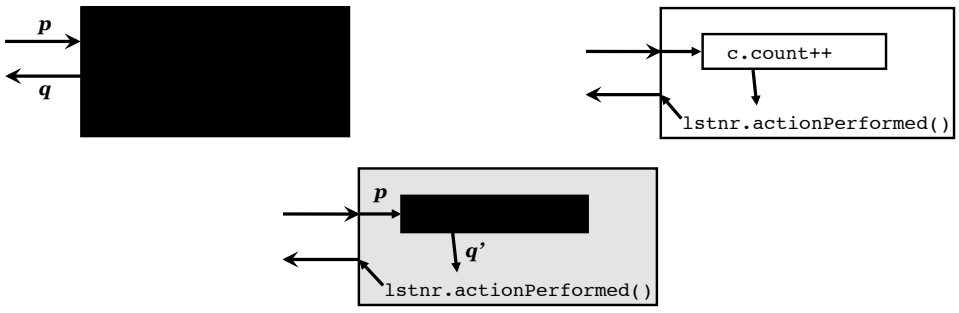

Figure 2.3 Greybox model programs (bottom) synthesize blackbox (left) and whitebox (right) specification styles. Irrelevant implementation details can be hidden while still identifying the conditions in which exposed code executes.

CHAPTER 3. SOLUTION APPROACH

Our solution for capturing mandatory calls inside of higher-order methods (HOMs) adapts grey-box, model program specifications [3, 4, 5] and uses a copy rule [18] to reason about calls to HOMs specified with model programs. An example model program specification for Counter's HOM `bump` is shown in Figure 3.1. In this figure, the **`public`** modifier says that this specification is intended

```
/*@ 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 3.1 Model program specifying the mandatory call to `actionPerformed`, from previous work [21].

for client use [14]. 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 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 [1, 19].

Our approach prescribes how to do two verification tasks:

- *Verification of a method implementation against its model program specification.* Our approach imposes verification conditions on the code by "matching" the code against the model program, which yields a set of verification conditions for the code fragments 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 (or client) 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.

We will look at the details required for each verification, then give a practical way to derive implicit model programs directly from annotated code. Examples that formalize common object-oriented design patterns are then discussed in detail. This chapter closes by identifying some limits to our current technique.

3.1 Verifying Implementations

Verifying a method implementation against its model program is itself a two-step procedure. The first step is matching, to check whether the method body has a similar structure to that of the model program. The matching we use to establish this property is simple. We require that implementations must match the model program exactly except where the model program contains a specification statement. Specification statements can only be matched by a **`refining`** statement in the implementation. To associate **`refining`** statements with the corresponding point in the model program, each **`refining`** statement must have a specification identical to the specification statement it implements.

To see an example of this, compare `bump`'s code in Figure 3.2 with the model program in Figure 3.1. This is a correct match, 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 call in the model program. The mandatory call exposed in this example is `actionPerformed`, inside of the HOM `bump`. Each piece of code matches a corresponding piece of the model program, so we are guaranteed that both model program and implementation share a similar structure.

The second stage of this task is proving that every refining statement in the code correctly implements its specification. Let us demonstrate this with a proof using weakest-precondition semantics. That is, assuming the specification statement's postcondition, we must show that the end of the body of

```
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 3.2 Code matching the model program specification for Counter's mandatory call. The **`extract`** syntax is explained in Section 3.3.

the refining statement is reachable from the specification's precondition and only assigns to the fields permitted by its frame. In Figure 3.2, the only value allowed to change in the refining code is an instance's `count` field, which is incremented by one. The body of the **`refining`** statement is the statement

$$\texttt{this.count = this.count+1;}$$

so we must show

$$\{true\}\ \texttt{this.count = this.count+1;}\ \{\texttt{this.count == \old(this.count+1)}\}$$

where $true$ is the assumed precondition of our **`normal_behavior`** specification statement. By the standard proof rules for assignment [25], we can derive

$$\texttt{\old(this.count+1) == \old(this.count+1)},$$

or $true$, so this code is a permissible refinement of its model program counterpart. Since all other code (the **`if`**-statement containing a mandatory call) matches exactly, this is sufficient to show that the method implementation refines its model program. It also ensures that mandatory calls occur in the HOM implementation only in the specified states.

Despite its simplicity, our technique is practical. It allows programmers to trade the amount of effort they invest in specification and verification for flexibility in maintenance. Programmers can write abstract specification statements that hide details in order to allow multiple possible implementations to satisfy their intentions. Conversely, programmers may 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 this technique.

3.2 Client Reasoning

To verify calls of HOMs with model program specifications, we have developed a technique that supports strong conclusions without requiring the use of higher-order logic or trace semantics in specifications. Instead, we use a *copy rule* [18], in which the body of the model program specification is substituted for the HOM call at the call site, with appropriate substitutions.[1] For example, to reason about the call to `c.bump()` in Figure 1.5, one copies the body of the model program specification to the call site, substituting the actual receiver `c` for the specification's receiver, **`this`**. We show such a substitution in Figure 3.3.

```
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 3.3 The result of substituting the model program's body for the call `c.bump()` from Figure 1.5.

This code exposes a call to `actionPerformed` by `c`'s `lstnr` field, which makes it easy to verify the final assertion. Clients can infer from the assertions before the **`normal_behavior`** specification statement that just before the mandatory call is made, `c.lstnr` is equal to `lv`. For all matching implementations, any code refining the specification statement preserves this property, satisfying the **`assignable`** clause of the **`normal_behavior`**. To prove the final assertion is true, verifiers can apply the specification of `actionPerformed` from the LastVal class.

Our approach works well for clients, because their understanding of the code no longer relies on a less-than-helpful blackbox specification of the HOM or the very weak specification of its mandatory calls. Instead clients reason with the substituted body of a model program and their knowledge of often stronger specifications on the actual mandatory calls made at the call site. Thus clients can apply their specific knowledge about particular HOM calls to draw strong conclusions.

[1] The copy rule can be used repeatedly to verify recursive HOM calls, as long as there is a way to limit the depth of recursive copying for each case. Providing additional information to derive a maximum recursive depth, perhaps by defining a progress metric or declaring an explicit limit, is one way to enable reasoning about recursive specifications. For this presentation, however, we do not assume any such rule.

3.3 Extracting Implicit Model Programs from Code

Due to the simplicity of our matching, model program specifications necessarily contain redundant copies of all implementation code not hidden behind **`normal_behavior`** specification statements. This duplication introduces the possibility of errors and is a maintenance headache.

When 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 at the same time. We used this functionality earlier in Figure 3.2. When a method has the **`extract`** modifier, we extract an implicit specification from the code. This extraction process derives a model program, in this case resembling Figure 3.1, 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. The resulting model program automatically matches the code without creating another explicit copy. The specification shown in Figure 3.1 could be what a specification browsing tool would show to readers, even if the specification was written in the code as in Figure 3.2. Offering this shortcut makes model programs more practical for specifiers to adopt in many cases.

The ability to keep model program specifications separate from the code they specify remains useful in the two following 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. In both cases, explicit model programs are valuable specification artifacts with no direct copy to maintain.

3.4 Example Verifications

We have already shown how to specify the `bump` method for the Counter class, an example of the Observer design pattern [11]. Here we discuss the verification of other design patterns as well as a more general application for model programs. Specifically, we will show how model programs enhance verification of the Template Method and Chain of Responsibility design patterns [11]. These patterns make good examples because each uses our technique in a different way to improve on verifying object-oriented designs. The last example shows a non-OO application that demonstrates some technical shortfalls to our approach.

3.4.1 Template Methods: Following a Recipe

Template methods are HOMs that are used in frameworks, where they sequence calls to "hook methods" that are overridden to be customized by the framework's users. Typically hook methods have weak specifications in order to allow a wide variety of possible behavior in subclasses. A template method makes mandatory calls to these hook methods, which works very well with model program specification.

Consider the HOM `prepare()` in Figure 3.4. The model program specification extracted from the method `prepare` is shown in Figure 3.5. This model program has two mandatory calls to the weakly specified hook methods, `mix` and `bake`. Class StringyCake in Figure 3.6 is a specializer supplying code and stronger specifications for overridden methods. A client using StringyCake instances would be able to use the model program specification of `prepare` plus the specifications of the hook methods to prove the assertion in Figure 3.7. This works because the client can substitute the model program specification wherever they call `prepare`, which exposes the strongly specified hook method calls.

```
import java.util.Stack;

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

        /*@ refining normal_behavior
          @   assignable pan;
          @   ensures pan != null && 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 3.4 The class CakeFactory, with its template method `prepare`, and two hook methods: `mix` and `bake`.

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

Figure 3.5 `prepare`'s extracted specification.

```
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 3.6 Class StringyCake, a subclass of CakeFactory. The keyword **also** indicates that the given specification is joined with the one it overrides [12, 15].

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

Figure 3.7 Client code that calls `prepare`.

Figure 3.8 shows the result of substituting the actuals into the model program from Figure 3.5 for the call to the `prepare` method. In this substitution, we have changed the return in the code into the assignment to the variable receiving the call's value, as usual [25]. Since Figure 3.8 exposes hook methods where we can identify the more specialized type of their receiver, we can now prove the final assertion.

At this call site, the critical knowledge clients hold is that `c` is a StringyCake instance. The definitions of its overridden hook methods have stronger specifications than CakeFactory objects do in general. For this proof, we start by assuming an empty initial state and applying the effects of each line from Figure 3.8. Initially, declare the variables `c` and `r`, then bind `c` to a new instance of type StringyCake. Inside the block representing our substituted model program, declare the variable `pan` before "executing" an arbitrary statement whose effect is described by the **`normal_behavior`** spec-

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

    normal_behavior
        assignable pan;
        ensures pan != null && pan.isEmpty();

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

Figure 3.8 Client code that calls `prepare`, after using the copy rule and substituting the actual receiver `c` for **`this`**.

ification. At this point, before calling either hook method on `c`, we know that `pan` is no longer null and its `isEmpty` method returns true. Since `isEmpty` is true, the precondition of `c`'s `mix` method has been met. The effect of that call is to add the string "batter" to the top of the `pan` stack. After returning from this call, the precondition of `c`'s `bake` method has been satisfied, so the top of the `pan` stack is now the string "CAKE". At this point, we know enough to establish that the value given to `r` by this code (i.e., the value returned by calling `pan.pop()`) is, in fact, the string "CAKE". This final state supports the final assertion and concludes our proof.

This proof works because it applies a formal understanding of how the StringyCake class implements the `mix` and `bake` hook methods without overriding its template, the `prepare` method. Client reasoning with model programs exposes this feature of a template method design: the interaction of overridden hook methods with a standard template describing their order of invocation.

3.4.2 Chain of Responsibility: Testing Static Configurations

Chain of Responsibility is another object-oriented pattern whose use can be formalized by calls to the pattern's characteristic methods [11]. Every receiver along the chain has up to two responsibilities: to implement the shared method and/or to pass unhandled cases farther along the chain. The method that chains receivers together must be a weakly-specified mandatory call, for the value in applying this pattern relies on the diversity of classes belonging to the chain.

One implementation of this pattern might be a mail system, some network of relays that are responsible collectively for transmitting a message (in our case, a letter) from one endpoint to another. The chain of responsibility is shared by every member of the network implementing the Mailer interface, shown in Figure 3.9. Suppose further that this network resembles Figure 3.10. For Alice to send a letter to Bob, she sends the letter l to the office she is nearest, Office `a`. As a member of the chain of

responsibility, Office `a` either must pass the letter off to Bob directly (which it can't) or pass the letter along the chain. This passing is handled by the `send` method, with Person, Office and Sorter instances all implementing the Mailer interface. Note that it would not be helpful to write a model program for the Mailer interface, because information about the receiver of the mandatory call will differ for each implementing class. Instead, model programs should be written for each specific implementation of `send`, but preferably with an eye to minimizing the total number of model programs.

```
public interface Mailer {
    public void send(Letter l);
}
```

Figure 3.9 The Mailer interface identifies a single method `send` for all objects that transmit messages in our mailing network.

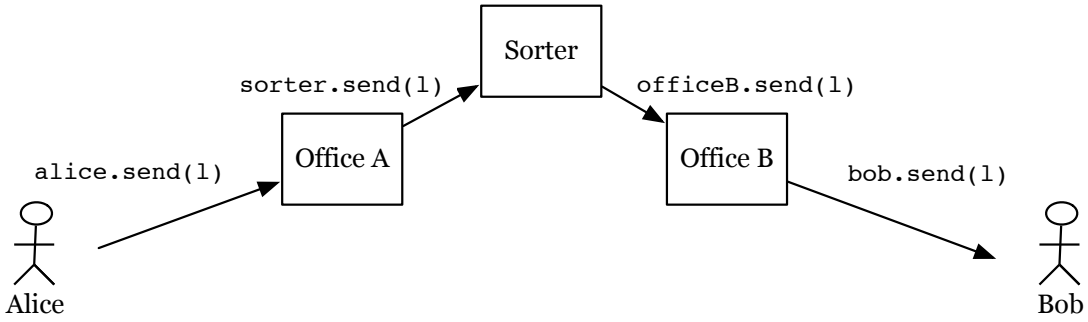

Figure 3.10 An example mailing network connecting Alice to Bob.

One concern for implementors of this network might be guaranteeing the delivery of a given message along a known static configuration. For our mailing network, this problem can be phrased as the question "Does Bob receive the letter Alice sent?" The assertion of Figure 3.11 is a formalization of this question. To reason about that result, we invoke the copy rule on `alice.send(l)`, whose model program exposes a call to `sorter.send(l)`. Invoking the copy rule twice more should reveal that `alice.send(l)` does indeed result in Bob receiving the message, if sufficiently-detailed model programs for those classes are given. In this case, our technique enables strong conclusions for systems with a static configuration of the responsibility chain.

3.4.3 Technical Limitations

Model programs give specifiers a finer degree of abstraction for HOMs, particularly by allowing structural or behavioral details of object-oriented designs to be formally captured. HOMs do not occur

```
Letter l = new Letter(alice, bob);
alice.send(l);
Mailer[] holder = new Mailer[1];
l.getHolder(holder);
//@ assert(holder[0] == bob);
```

Figure 3.11 Client code that makes an assertion of guaranteed message delivery.

solely inside of object-oriented code though. Functional programming has its share of HOMs to which we can apply our technique.

For example, the common map operator could be implemented in Java with something like Figure 3.12. In this implementation, `map` is the HOM and the IntFun method `f` is our mandatory call. Here we use **`extract`** to derive an implicit model program directly from the code that implements the `map` operation over an array of integers. The derived model program hides none of the implementation, however, since the only abstraction we currently provide is the **`normal_behavior`** specification statement.

```
public class Map {
  public /*@ extract @*/ void map(IntFun s, int[] a)
  {
     for (int i = 0; i < a.length; i++) {
       s.f(a, i);
     }
  }
}
```

Figure 3.12 Class Map implements a staple of functional programming in Java.

This reveals a pair of related weaknesses for our current technique: the lack of abstract control-flow constructs and the relative strictness in how model programs match against implementations. If an abstract loop statement existed, then the **`for`**-loop outside of the mandatory call could remain hidden. Similarly, with a more flexible matching procedure, **`extract`** could generate multiple model programs (e.g., one that exposes the call to `f` on the IntFun argument and another that abstractly iterates over all elements of the array) to allow implementors to reason about the HOM differently depending on the salient features needed at different call sites. Chapter 5 discusses our plan to address these concerns.

We do not mean to imply that our technique cannot benefit such a HOM. Even without hiding any implementation details, our model programs still enable strong conclusions about mandatory calls. To see this is the case, look at the code of Figure 3.13. After substitution of our whitebox model program, the effect of a call to `map` is plain to see. If we assume that the Scale class is a subclass of IntFun whose `f` method scales integer arguments by a factor of two, then Figure 3.14 is sufficient to achieve the strong conclusion that `map` performs as expected.

```
int[] ai = new int[] {1,3};
Map m = new Map();
Scale by2 = new Scale(2);
m.map(by2,ai);
//@ assert ai[0] == 2 && ai[1] == 6;
```

Figure 3.13 Client code that calls `map` while asserting its desired effect.

```
int[] ai = new int[] {1,3};
Map m = new Map();
Scale by2 = new Scale(2);
for (int i = 0; i < ai.length; i++) {
  by2.f(ai, i);
}
//@ assert ai[0] == 2 && ai[1] == 6;
```

Figure 3.14 Code of Figure 3.13 after substituting a model program for `map`.

CHAPTER 4. EXTENDING JML WITH MODEL PROGRAMS

This chapter summarizes the state of the effort to implement model programs as an extension to the JML static checker `jmlc`. As described in Chapter 3, our greybox model programs add three new features to JML: the model program itself, the **`refining`** statement for matching specification statements in the model program to the implementation code that refines them and the syntactic sugar **`extract`** for creating implicit model programs directly from an existing implementation. We describe relevant design features of JML, define how model programs extend that design and then provide an informal analysis of that extension.

4.1 JML Background

To understand how the design of these features integrates with an existing tool for JML, we must first understand the design of the tool being extended. The static checker for JML included in the Common JML tools, named `jmlc`, is built on top of the MultiJava compiler, whose architecture has been documented by Clifton [8]. This tool builds on the MultiJava architecture to support JML's specification syntax and semantics. For clarity of the present discussion, we will highlight only those portions of the design of `jmlc` that impact our own extension. The three features being implemented for model programs belong to two categories of specification syntax: method annotations and specification statements.

JML adds specification annotations on method declarations in two primary ways: as specification cases that may come either before or after the method signature and as modifiers on the method or its arguments. Specification cases are the primary kind of specification annotation for Java methods. They describe the behavior of the method in terms of pre-/postcondition pairs, frame axioms and other blackbox detail. Model programs will become another kind of specification case. Some examples of method modifiers are **`pure`**, for describing a method without side effects, and **`non_null`**, which says a method's argument will never be null. Both of these modifiers act as syntactic sugars for common implicit specification cases. The **`extract`** modifier is a sugar, signaling for an implicit model program to be extracted from the method body.

JML also provides a number of statements for verifying specifications by annotating the code directly. These include annotated loops as well as statements for the creation and manipulation of ghost

variables. Heavyweight specification cases (i.e., the many shades of **`behavior`** cases) can also be used as specification statements, but will only be valid on their own inside of a model program or as part of a **`refining`** statement in the implementation. In this early implementation, only **`normal_behavior`** statements are explicitly supported. The **`refining`** statement is another specification statement, the role of which will be to tie model program statements to the implementation's code.

4.2 Our Extension

Having introduced where the new features fit into JML syntactically, we now disclose details of each feature's design. This chapter will conclude with a look at the direct implications of these choices.

4.2.1 The Model Program Specification Case

At the time of implementation, the `jmlc` codebase already contains nascent support for parsing model programs, the JmlModelProgram class. The responsibilities of this class include containing the AST representing the model program's body as well as defining the typechecking rules for model programs. In our implementation, model programs consist of a visibility modifier, a block of (possibly abstract) JML-permissible statements and a flag `isExtract`, identifying whether the model program was extracted. The visibility modifier has implications for the fields and methods that may be referenced in the model program's body, while `isExtract` is helpful when checking an implicitly-generated specification.

4.2.2 Implicit Model Programs via `extract`

For methods marked **`extract`**, instances of the class `JmlExtractModelProgramVisitor` generate implicit model programs based on the method's body. Such a visitor transforms the code into a model program as described in Section 3.3. These objects are not called directly by the checker, but instead by JmlModelProgram, with the class method `extractInstance`. In turn, this method is invoked by the class method `makeInstance` of the JmlMethodDeclaration class to add the implicit model program to the represented method's specification set.

4.2.3 `refining` Specification Statements

Operationally, the **`refining`** statement has no effect beyond associating a behavioral contract with the code that refines it. Maintaining this association is key to our technique, as we saw in Section 3.1. Checking that these **`refining`** statements occur as expected is the responsibility of the visitor described by the JmlRefineModelProgramVisitor class. This check is straightforward for the current technique: to check equality of AST nodes down to the level of the **`refining`** statements.

This has been implemented by providing a unique visit method in the visitor for every leaf of the JML statement grammar. This choice was partly forced by the intricacies of the JML2 AST objects, but also allows modular modifications when considering future work. For example, a new form of specification statement should only require one new method per visitor and each method's implementation would depend only on the details of the new statement.

4.3 Design Implications

These descriptions provide a snapshot of an early JML2 implementation that supports our described technique. As attention has been given to how and why this works the way it does, so should we consider where and how such an implementation may go from here. The JML Common Tools also provide a runtime assertion checker, `jmlrac`. Modifying this tool to enforce the contracts associated by **`refining`** statements should be trivial. Tool support for the client reasoning prescribed in Chapter 3 follows by simply decoding **`refining`** statements as an assume/assert pair. In the course of extending `jmlc`, it became clear that some re-engineering of how assignability information is gathered will be necessary in the near future. This will be re-examined in Section 5.1. Finally, as the principles governing model program extraction and refinement are themselves adapted in future work, the two-visitor design presented here should prove effective in isolating these adaptations.

CHAPTER 5. FUTURE WORK & CONCLUSIONS

In this chapter we look ahead to further development and other applications for greybox reasoning with model programs. After listing some of those possibilities, we revisit the promises of previous chapters to make concluding remarks.

5.1 Future Work

The work described by Chapters 3 and 4 represents a working draft of specification language features that define how JML can support HOM documentation. The tools developed to solve this problem could assist other open research questions. For example, we use **`refining`** statements to associate executable Java code with its relevant specification statement in the model program. This functionality supports granular statement-level annotation of code with specification constructs. We particularly want to explore how this construct compares with temporal logic [20, 24]. Model programs themselves can be used for more than just supporting client reasoning as we have demonstrated here. A complementary form of model program has been developed by Veanes, et al. [27, 26] with an early application found in the work of Barnett and Schulte [2]. The Spec# paradigm uses model programs to specify interface automata, complete with its own notion of refinement as well as an exploration of how model programs compose together to derive more complete models of complex program behavior. One promising direction for JML would be to explore the transformation of a model program into an abstract model of program behavior. Such a behavioral model could foreseeably have applications in model checking, unit testing or as a rapid prototype for design feedback.

As we saw near the end of Chapter 3, our solution does not come without limitations. There is a demonstrable need for more and more-varied abstract constructs for capturing control flow as well as a more flexible matching procedure. Nondeterministic choice is capable of modeling both a choice in implementations as well as an abstract, permutable if-then-else specification statement. Also, there may be multiple ways to specify loops or recursions that invoke mandatory calls. Where matching falls apart lay primarily in its strictness. If the model program does not contain a specification statement at a particular program point, we say the implementation must match exactly. While this simplifies reasoning about concrete statements in the model code, there should be some room for negotiation, particularly for security purposes [6]. Another concern that emerges from the discussion of Chapter 3

is a clear need for a notion of refinement that allows model programs to refine each other. Solutions to this problem that are modular may well support model program composition for cases where multiple model program definitions are given for a single implementation. Currently, the implementation issues a warning in the presence of multiple model programs and only attempts to match the structure of the closest syntactic definition.

Chapter 4 mentions an intention to modify how `jmlc` handles its **`assignable`** clauses, which we will expand upon here. Where these clauses are traditionally encountered, at the method level, has a standard semantics that covers the entire method implementation. With the introduction of model programs, however, these clauses are brought down to the statement level, for example, as a clause within a **`normal_behavior`** specification statement. To properly mesh these new clauses with the established system, however, these **`assignable`** clauses need precise analysis. Previous work has explored the kind of delicacy required for the general case [28], but this may need revisiting in a model program context. A trivial implementation could simply union all the **`assignable`** clause information inside a given model program, but it remains to be seen if this is the correct intuition. The implementation work done for this thesis does not provide any special handling for assignability information inside of a model program.

5.2 Conclusions

This thesis aimed to convince the reader of the utility of a novel specification technique, greybox reasoning with model programs. We need such reasoning to enable clients to draw strong conclusions in the presence of higher-order methods that make mandatory calls. Object-oriented design patterns that provide structural and behavioral benefits are one domain where strong conclusions are needed to perform rigorous formal verification, though by no means are they unique. We have added a working implementation of model programs to the `jmlc` compiler in the JML Common Tools. Where possible, we prefer simple, practical techniques that minimize the cognitive overhead of the new constructs while maximizing the specification benefit of their use. As we saw in Section 5.1, multiple paths of progress stand before us. Model programs have a number of applications; both their present and future potential looks bright.

BIBLIOGRAPHY

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

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

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

[4] 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`.

[5] 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`.

[6] M. J. Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects of Computing*, 14(1):2–34, 2002.

[7] 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.

[8] C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. The author's masters thesis.

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Available from `http://www.jmlspecs.org`, Dec. 2008.

[17] 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, New York, NY, Oct. 1998. ACM.

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

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

[20] A. Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 1–38. Springer-Verlag, New York, NY, 1993.

[21] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367. ACM, Oct. 2007.

[22] 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.

[23] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edition, 2002.

[24] K. Trentelman and M. Huisman. Extending jml specifications with temporal logic. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 334–348, New York, NY, 2002. Springer-Verlag.

[25] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.

[26] M. Veanes, C. Campbell, and W. Schulte. Composition of model programs. In *FORTE '07: Proceedings of the 27th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, pages 128–142, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In M. Wermelinger and H. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 273–282. ACM, 2005.

[28] C. Ye. Improving JML's assignable clause analysis. Technical Report 06-19, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, July 2006.