Model Programs for Preserving Composite Invariants

Steve M. Shaner
Iowa State University
smshaner@cs.iastate.edu

Hridesh Rajan
Iowa State University
hridesh@cs.iastate.edu

Gary T. Leavens
University of Central Florida
leavens@eecs.ucf.edu

ABSTRACT

We describe a solution for the SAVCBS challenge problem: a technique for specifying and verifying invariants for objects designed using the Composite design pattern. The solution presents a greybox specification technique using JML's model program feature. We show that model program specifications function as exemplars for capturing helper method calls in a way that preserves modularity and encapsulation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement — Documentation; D.2.11 [**Software Engineering**]: Software Architectures — Information hiding, Languages, Patterns; D.3.3 [**Programming Languages**]: Language Constructs and Features — classes and objects, inheritance; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

General Terms

Verification

Keywords

Greybox specification, verification, model program, composite design pattern, JML language.

1. INTRODUCTION

Invariants in formal specification languages, such as the Java Modeling Language (JML) [5, 7, 8] describe relationships that hold in each visible object state. They both document intended relationships and impose proof obligations on code for each object. Our solution to the 2008 SAVCBS challenge problem describes a methodology for verifying invariants of objects designed using the Composite pattern [6, pp. 163]. We demonstrate how model program specifications can be used to enforce a simple invariant for Composite objects [9, Section 4.1].

A model program [16] is JML's realization of the greybox specification technique [4]. A model program is thus a hybrid between program code and specification, and can be thought of as an abstract algorithm [1, 2, 3, 11, 12]. The algorithm is abstract in the sense that it may suppress many implementation details, only specifying their effects using specification statements. This allows the specifier to hide some details, while showing others to the reader. In the refinement calculus, code satisfies an abstract algorithm specification if the code refines the specification [1, 12, 11]. However, JML currently limits refinement of model program by requiring all exposed code to match the implementation exactly [16].

A major benefit of JML's model programs is that they can be used to specify functional dependencies among objects in a straightforward way—by exposing code that maintains the dependencies. In this paper we show how this technique works within the Composite class of the Composite design pattern. For complex object structures like those seen in the Composite design pattern, a set of helper methods can be defined that exploit the object structure to establish the invariant. Writing model programs that show how all methods that may affect the invariant invoke these helper methods and then showing that these model programs individually preserve the invariant is our recommended methodology. We demonstrate this methodology in our solution to the challenge problem.

2. JML MODEL PROGRAMS

Model program specifications in JML, like their counterpart in Büchi and Weck's greybox specifications [4], are algorithmic abstractions of concrete functionality. They selectively expose only the desired parts of a method's concrete behavior. In particular they can specify calls of certain methods in specified states; we call such specified method calls "mandatory" calls [16].

Model program specifications have two major benefits:

1. A suitable model program specification allows more expressive invariants on the concrete behavior compared to a behavioral specification, and
2. Such invariants do not depend upon hidden implementation details, thus they improve information-hiding modularity [15] compared to exposing all of the implementation for the purpose of writing invariants.

Figure 1 shows an example JML specification for the class `ElementCollection`. In the special JML comments, the private field `inner` is declared to be public for specification purposes using **`spec_public`**. The model program specification for the addAll method for ElementCollection class is shown in Figure 1 on lines 4–13. This specification has two parts: a behavioral specification statement on lines 5–10 and a white-box specification on lines 11 and 12. Behavioral specification statements all begin with the **normal_behavior** keyword and this one contains a precondition (the **requires** clause), a frame axiom (**assignable** clause), and two postconditions (**ensures** clauses). In postconditions the operator **old** is used to refer to the previous state. The **normal_behavior** keyword that starts specification statements selects a total correctness specification that allows no exceptions to be thrown; i.e., if execution starts in a state that satisfies the precondition, then it must terminate normally in a state that satisfies both the frame axiom and the postconditions.

```
1 class ElementCollection extends Collection {
2   private /*@ spec_public @*/
3     Collection inner;
4   /*@ public model_program {
5     @  normal_behavior
6     @    requires inner != null;
7     @    assignable this.inner;
8     @    ensures c.size() == \old(c.size());
9     @    ensures this.inner.size() ==
10    @     \old(this.inner.size());
11    @  for (Element e : c)
12    @    this.add(e);
13    @ } @*/
14  public void addAll(ElementCollection c) {
15    /*@ refining normal_behavior
16      @  requires inner != null;
17      @  assignable this.inner;
18      @  ensures c.size() == \old(c.size());
19      @  ensures this.inner.size() ==
20      @    \old(this.inner.size()); @*/
21    { /* resize array if necessary */ }
22    for (Element e : c)
23     this.add(e);
24  } }
```

Figure 1: An example JML model program specification.

The behavioral specification describes invariants maintained by the parts of the concrete implementation that are not visible in the specification. This hides changeable implementation details from the client code. The white-box specification exposes part of the concrete implementation to allow clients to write more expressive invariants. A simple example of such increased expressiveness would be the guarantee that all invariants maintained by the method add for class ElementCollection (not shown) will also be preserved by the method addAll. Thus, for example if the method add maintains a count of all elements in the collection, such count would be accurate event if elements are added in bulk using the method addAll.

Such model program specifications are only valid for use in reasoning if the concrete implementations refines them [1, 11, 12]. For JML model program specifications instead of adopting a general notion of refinement, a more pragmatic approach based on structural refinement is adopted. A concrete implementation refines a model program specification if it is structurally the same as the specification, up to the code implementing the black-box specification behaviors. In Figure 1 the concrete implementation declares that the elided code on line 21 refines the black-box specification on lines 15–20 using the **refining** keyword. Each **refining** statement must have the same specification part and a body (between the braces and outside the special JML comments) that satisfies that specification. This makes verifying refinements easier. The rest of the method implementation (lines 22 and 23) is identical to its counterpart in the specification (lines 11 and 12). The details of the refinement technique are described in detail by Shaner, Leavens and Naumann [16].

In addition to the semantics described above [16], in this paper we also require that a correct implementation of a specification statement must not call any methods that are explicitly called in the whitebox (executable) code portion of the model program. We will see why this is needed below.

Since each **refining** statement contains a specification, writing model program specifications separate from the concrete implementation is often verbose and redundant. To reduce annotation burden and the cost of keeping model programs consistent with respect to the concrete implementation, JML also provides an additional syntactic sugar **extract** for extracting such specifications. An example of this feature is shown in Figure 2.

```
1 class ElementCollection extends Collection {
2   private /*@ spec_public @*/
3     Collection inner;
4   public /*@ extract @*/
5    void addAll(ElementCollection c) {
6     /*@ refining normal_behavior
7       @  requires inner != null;
8       @  assignable this.inner;
9       @  ensures c.size() == \old(c.size());
10      @  ensures this.inner.size() ==
11      @    \old(this.inner.size()); @*/
12    { /* resize array if necessary */ }
13    for (Element e : c)
14     this.add(e);
15  } }
```

Figure 2: Extracting Model Program Specifications.

In this version of the method addAll the **extract** keyword is used on line 4. This results in an automatic generation of the model program specification during verification. The automatic extraction of model program specification proceeds by suppressing the bodies of **refining** statements, replacing them with the specifications they contain as specification statements. The extracted model program specification for our example is the **normal_behavior** statement found on lines 6-11 in Figure 1, followed by the **for**-loop exactly as it appears in the code.

3. COMPOSITE'S SPECIFICATION

Our solution, given in Figure 3 and Figure 4, contains a combination of model programs, helper methods and pure methods.

The class Component is specified in Figure 3. As in previous example, the two protected fields parent and total are declared to be public for specification purposes using **spec_public**. The invariant in this figure is not the one we are mainly concerned with.

The subclass Composite is specified in Figure 4. It has 2 fields: an array components and an integer count. Lines 8–11 give the invariant we are mainly concerned with for the challenge problem; it states that total is one more than the sum of the values of the total fields of each object in the components array.

The two methods in Figure 4 have model program specifica-

```
class Component {
  protected /*@ spec_public nullable @*/
    Composite parent;
  protected /*@ spec_public @*/ int total = 1;
  //@ protected invariant 1 <= total;
}
```

Figure 3: Specification of Component.

tions: `addComponent` and `addToTotal`. The model program for method `addComponent` is given explicitly on lines 13–24 (preceeding that method's header), while the model program for `addToTotal` is implicit in the body of the method `addToTotal`, as indicated by the keyword **`extract`** [16]. The automatically extracted model program specification, for the method `addToTotal` is shown in Figure 5.

3.1 Specification

We now describe how the model programs of Figure 4 specify that `Composite` instances preserve the invariant on lines 8–11.

Consider method `addComponent`. The primary responsibility of this method is to modify the representation array `components` and appropriately update the `total` field. The invariant adds a subtle complexity to this update by requiring that the value of each subcomponents' `total` field is included in the value of its parent's `total` field. Thus a correct implementation of `addComponent` must capture the structural relationship between the composite and its subcomponents and use this information when updating the `total` fields.

In our example, this structural relationship is captured by the definition of method `addToTotal`. It both modifies this instance's `total` field and asks that the parent (if one exists) be modified as well. This has the useful effect of re-establishing the invariant for all instances for which the invariant might have been violated, provided `addToTotal` is called only once, and with the appropriate argument.

For this problem, we have written a model program for `addComponent` that exposes its call to `addToTotal`. Recall that, due to the restricted notion of refinement in our technique, correct implementations of `addComponent` must call `addToTotal` after changing both parent and child, as described by the model program. It is this notion of "structural similarity" that makes the call to `addToTotal` "mandatory" [16]. In proving that a model program for `addComponent` is refined by its implementation, we show structural similarity between the model program and the implementations of `addComponent` in all subclasses of `Composite`. Thus, if the model program preserves the invariant for all `Composite` objects, then the invariant will be preserved by all subclasses, since they must also refine the model program in the same sense.

As noted above, we require that specifcation statements must not call any methods that are explicitly called in model program. For the specifications in Figure 4, this means that the bodies of the **`refining`** statements inside the implementation of `addComponent` are prohibited from calling `addToTotal`.

```
1 class Composite extends Component {
2   private /*@ spec_public @*/
3     Component[] components = new Component[5];
4       //@ in objectState;
5       //@ maps components[*] \into objectState;
6   private /*@ spec_public @*/ int count = 0;
7       //@ in objectState;
8   /*@ protected invariant total
9     @       == 1 + (\sum int i;
10    @               0 <= i && i < count;
11    @               components[i].total); @*/

13  /*@ public model_program {
14    @  normal_behavior
15    @    requires c.parent == null;
16    @    assignable this.components;
17    @    ensures this.components.length
18    @              > this.count;
19    @  normal_behavior
20    @    assignable c.parent, this.objectState;
21    @    ensures c.parent == this;
22    @    ensures this.hasComponent(c);
23    @  addToTotal(c.total);
24    @ } @*/
25  public void addComponent(Component c) {
26    /*@ refining normal_behavior
27      @   requires c.parent == null;
28      @   assignable this.components;
29      @   ensures this.components.length
30      @         > this.count; @*/
31    { /* resize components, if necessary */ }
32    /*@ refining normal_behavior
33      @   assignable c.parent, this.objectState;
34      @   ensures c.parent == this;
35      @   ensures this.hasComponent(c); @*/
36    {
37      components[count] = c;
38      count++;
39      c.parent = this;
40    }
41    addToTotal(c.total);
42  }
43  private /*@ helper extract @*/
44    void addToTotal(int p) {
45    /*@ refining normal_behavior
46      @   requires 0 <= p;
47      @   assignable this.total;
48      @   ensures this.total
49      @         == \old(this.total) + p; */
50    { total += p; }
51    Component aParent = this.parent;
52    while (aParent != null) {
53      /*@ refining normal_behavior
54        @   assignable aParent.total, aParent;
55        @   ensures aParent.total
56        @         == \old(aParent.total) + p;
57        @   ensures aParent
58        @         == \old(aParent.parent); @*/
59      {
60        aParent.total += p;
61        aParent = aParent.parent;
62  } } }
63  /*@ pure @*/ boolean hasComponent(Component c) {
64      // ...
65 } }
```

Figure 4: JML model program specification for Composite, based on Leavens, Leino, and Müeller's specification [9, Figure 10].

```
/*@ public model_program {
  @   normal_behavior
  @     requires 0 <= p;
  @     assignable this.total;
  @     ensures this.total
  @           == \old(this.total) + p;
  @   Component aParent = this.parent;
  @   while (aParent != null) {
  @     normal_behavior
  @       assignable aParent.total, aParent;
  @       ensures aParent.total
  @             == \old(aParent.total) + p;
  @       ensures aParent = aParent.parent;
  @   }
  @ } @*/
void addToTotal(int p);
```

Figure 5: Extracted specification for `addToTotal`.

```
1       Composite root = new Composite();
2       Composite child = new Composite();
3       Component comp = new Component();

5       //@ assume root.total == 1;
6       //@ assume child.total == 1;
7       //@ assume comp.total == 1;
8       //@ assume root.parent == null;
9       //@ assume child.parent == null;
10      //@ assume comp.parent == null;
11      root.addComponent(child);
12      child.addComponent(comp);
13      //@ assert root.total == 3;
```

Figure 6: Clients reason by instantiating invariants for concrete contexts like this one, in which a tree of three components is built.

4. PROBLEMS & SOLUTIONS

The model programs of Figure 4 assist reasoning with invariants in two scenarios of interest: handling argument exposure for Composite clients, and when defining Composite subclass methods. We break the latter problem down into two mutually exclusive subproblems, the overriding of existing methods and the introduction of new ones.

4.1 Argument Exposure in Client Reasoning

Argument exposure occurs when an invariant, such as the one in Figure 4, depends on objects that are not under control of the object's methods [14]. In that figure, the invariant of `Composite` depends on the components in the `components` array. The challenge is how to maintain such an invariant when clients may change objects on which the invariant depends without calling methods directly on the object.

Let us consider how our specification in Figure 4 and the greybox approach (JML's model program technique) deal with this problem. In essence our solution is a special case of the visibility technique for maintaining invariants [9, 13]. To see this, note that the fields `total`, `components`, and `count` cannot be written by classes that do not see the invariant in Figure 4, because these fields are protected and private and the invariant is protected. Hence the invariant can be maintained in each subclass of `Composite`, by requiring all these subclasses to maintain it each time they change one of these three fields.

The key point of our specification is that the model program and the code it requires follow the chain of `parent` links upwards, and adjusting each total of each parent object. Since the precondition of `addComponent` requires that `c.parent` be null, no cyclic or aliased structure can be created using `addComponent`, thus there is always at most one parent for each Component `c`.

To see how this is done, consider the client code in Figure 6. This sets up the problematic case of a Composite object, `root`, that contains another Composite object, `child`, which itself contains the component `comp`. If `addComponent` maintains the invariant, then the assertion at the end of the figure should hold, even though line 12 modifies `child` without calling a method on its parent `root`. The invariant should apply when reasoning about the resulting heap structure, regardless of the order in which the components get added to each other.

The model programs described in Figures 4 and 5 are used in verification by substituting the model program's body for the call site of the method it specifies (with actuals replacing formals and care taken to avoid capture). In Figure 6, this means substituting in lines 14–23 of Figure 4 for each call to `addComponent`, renaming occurrences of the formals `c` and **`this`** to the appropriate instances. Furthermore, each of these substitutions exposes a call to `addToTotal`, so its model program body can be substituted similarly.

The resulting code resembles Figure 7. In this figure, lines 11–34 are the model program for `addComponent` substituted for the call on line 11 of Figure 6. Similarly, lines 35–58 are for the call on line 12 in the original. From this text and a Hoare-style proof system, we can verify that the closing assertion holds. This proof is straightforward after assuming the proof rules given in previous work [16] with a standard extension to handle while loops.

4.2 Overriding Composite's Methods

In subclasses of `Composite`, a developer might incorrectly try to override its methods `addComponent` or `addToTotal` in a way that violates the invariant or a model program specification. However, such an override would be incorrect in our technique, because not only are invariants inherited by subtypes in JML [7], but subtypes also inherit model program specifications. Thus methods inheriting a model program are subject to the same structural constraints as the overridden method. Though subclass implementors are free to refine the bodies of **`refining`** statements as long as they satisfy the contract behavior, all other exposed code must appear as it does in the model program. In this fashion, as long as the original model program preserves the invariant, subclass overrides of those methods cannot violate the invariant.

4.3 Extending Composite with New Methods

Subtypes also pose problems when they introduce new methods that do not override methods in their supertype(s). Such methods must preserve the inherited invariants, as would be the case in our example, but our technique does not yet provide direct support for this situation.

In our example, the way in which `Composite`'s invariant is maintained depends heavily on two assumptions: (a) `addComponent` is the only method that adds components to a composite, and (b) `addComponent` has a precondition that requires the parent of the added component to be null. An added method could violate these

```
Composite root = new Composite();
Composite child = new Composite();
Component comp = new Component();

//@ assume root.total == 1;
//@ assume child.total == 1;
//@ assume comp.total == 1;
//@ assume root.parent == null;
//@ assume child.parent == null;
//@ assume comp.parent == null;
normal_behavior
  requires child.parent == null;
  assignable root.components;
  ensures root.components.length
        > root.count;
normal_behavior
  assignable child.parent, root.objectState;
  ensures child.parent == root;
  ensures root.hasComponent(child);
{
  normal_behavior
    requires 0 <= child.total;
    assignable root.total;
    ensures root.total
         == \old(root.total) + child.total;
  Component aParent = root.parent;
  while (aParent != null) {
    normal_behavior
      assignable aParent.total, aParent;
      ensures aParent.total
           == \old(aParent.total) + c.total;
      ensures aParent = aParent.parent;
  }
}
normal_behavior
  requires comp.parent == null;
  assignable components;
  ensures child.components.length
        > child.count;
normal_behavior
  assignable comp.parent, child.objectState;
  ensures comp.parent == child;
  ensures child.hasComponent(comp);
{
  normal_behavior
    requires 0 <= comp.total;
    assignable child.total;
    ensures child.total
         == \old(child.total) + comp.total;
  Component aParent = child.parent;
  while (aParent != null) {
    normal_behavior
      assignable aParent.total, aParent;
      ensures aParent.total
           == \old(aParent.total) + comp.total;
      ensures aParent = aParent.parent;
  }
}
//@ assert root.total == 3;
```

Figure 7: The client code of Figure 6 after substituting the bodies of the model programs for `addComponent` and `addToTotal` and renaming field references to the appropriate instances.

assumptions, allowing aliased Composite structures to be created that our proof does not handle. Since the invariant about lack of aliasing is not stated explicitly in our specification, it is not clear how this part of our argument would apply to subtypes. To avoid this problem, one would have to write a static (i.e., global) invariant that described the required lack of aliasing. But this fix seems specific to the Composite design pattern, and it is not clear how our technique could be generalized to handle it.

5. DISCUSSION

At the class level, our example model programs describe the set of methods that are responsible for maintaining the representation invariant. They provide an abstract overview of where and how the invariant is maintained. The only way subcomponent membership can change is by calling `addComponent` and the only way `total` is updated is by a call to `addToTotal`. No calls to `addComponent` occur within this class, but if they did, a model program exposing that call could be written.

Model programs enable modular descriptions of the internal structure of code in ways that are useful for client reasoning. By choosing model programs to control the static structure of subclass implementations, our solution relies on the mechanical textual matching described in our previous work [16]. JML's **`refining`** statements clearly identify which specification statements are refined where inside of an implementation, while the **`normal_behavior`** specification statements use pure methods to hide specific representation details.

This is not to say that the working definitions used by our solution are perfect. Adding nondeterministic loops, conditionals and other constructs to the model program syntax would increase flexibility when matching implementations against a model program specification. Also, work on this paper highlighted a number of visibility issues for model programs that have not previously been investigated [10]. A basic issue is defining rules that respect visibility for model program specifications. There is also a complication posed by **`extract`**, which may pull out specifications and code that are legal within a method, but which may refer to private data. If the extracted model program is to be public, then such private data is not understandable by all clients, and so should be disallowed. We finessed this problem in our example by declaring all fields as **`spec_public`**, but this is certainly an area where more work is needed.

6. CONCLUSION

We have described how model program specifications can be used to specify and verify invariants in complex heap data structures created using the Composite design pattern. Our solution is a fruitful combination of the visibility technique for invariants with the greybox specification technique. The combination is fruitful because the greybox technique allows specifiers to describe exactly how a method must update all invariants that might be violated. In our example, `addComponent` is specified to update all of the `total` fields of all parents. This detail is crucial in maintaining the invariant for all `Composite` objects.

Acknowledgments

The authors were supported in part by the NSF grant CNS-06-27354 and CNS 08-08913.

7. REFERENCES

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

[2] R. J. R. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.

[3] R. J. R. Back and J. von Wright. Refinement calculus, part i: sequential nondeterministic programs. In *REX workshop: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 42–66, New York, NY, 1990. Springer-Verlag.

[4] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] 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.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[9] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19(2):159–189, 2007.

[10] 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.

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

[12] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program.*, 9(3):287–306, 1987.

[13] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, Oct. 2006.

[14] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[15] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–8, Dec. 1972.

[16] 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.