Model Programs for Preserving Composite Invariants

SAVCBS 2008 Challenge Problem Solution by Steve Shaner, Hridesh Rajan, Gary T. Leavens
Iowa State and University of Central Florida
Support from US NSF grants CNS-06-27354 and CNS-08-0891
Summary

Problem

- Specify and verify invariants for composite, which extend outside an object.

Approach: JML model programs

- Grey-box [Büchi-Weck97,99] specification

Verification modular by:

- Copy rule / substitution [Morgan88]
- Structurally-restricted refinement

Contribution

- Exposes only code for maintaining invariant
Setting

- JML
  - Pre- and postcondition specifications
  - Several verification tools:
    - ESC/Java2
    - Jack, LOOP, ...

- Java
  - Libraries:
    - Swing
    - I/O frameworks
    - Jakarta Commons
Grey-Box (Ref. Calc.) Approach
[Büchi-Weck97, 99]
Grey-Box Specifications in JML
[Shaner-Leavens-Naumann07]

```java
/*@ normal_behavior
@ requires c.count == 0;
@ ensures c.count > 0;
@*/
lnr.actionPerformed();

/*@ refining
@ normal_behavior
@ requires c.count == 0;
@ ensures c.count > 0;
@*/
{ c.count++;
}
lnr.actionPerformed();
```
Soundness of JML’s Approach
[Shaner-Leavens-Naumann07]

Structural Refinement:
- Exposed code appears exactly as shown
- Specification statements refined by refining statements
  - Cannot call methods with exposed calls

Behavioral Subtyping:
- Overriding methods must structurally refine
Composite Design Pattern

Component
- parent
- total

Composite
- components[]
- count
- addComponent(Component)
Sample Assertion to Prove

Composite root = new Composite();
Composite child = new Composite();
Component comp = new Component();
//@ assume root.total == 1 && child.total == 1;
//@ assume comp.total == 1;
//@ assume root.parent == null && child.parent == null;
//@ assume comp.parent == null;

root.addComponent(child);
child.addComponent(comp);
//@ assert root.total == 3;
Specification of Component

class Component {
    protected /*@ spec_public nullable @*/
        Composite parent;
    protected /*@ spec_public */ int total = 1;
    //@ protected invariant 1 <= total;
}
Specification of Composite (Data and Invariant)

class Composite extends Component {
  private /*@ spec_public @*/
  Component[] components = new Component[5];
  private /*@ spec_public @*/
  int count = 0;
  /*@ protected invariant */
  @
  total == 1 + (\sum int i;
  @
  0 <= i && i < count;
  @
  components[i].total);
}
Composite's Specification (addComponent)

```java
/*@ public model program { 

  normal_behavior
  requires c != this && c.parent == null;
  assignable this.components;
  ensures this.components.length > this.count;

  normal_behavior
  assignable c.parent, this.objectState;
  ensures c.parent == this;
  ensures this.hasComponent(c);

  this.addToTotal(c.total);

  } */

public void addComponent(Component c)
```
Composite's Specification (addToTotal)

/*@ private 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 == \old(aParent.parent);
}
} @*/

private /*@ helper @*/ void addToTotal(int p)
Verification Using Copy Rule

[ Morgan88 ]

To verify method call:

- Substitute model program specification
- Replace formals with actuals
  - Avoid capture

Usual rules for other statements
Rule for Method Calls
(Copy Rule + Substitution)

\[\text{specFor}(T', m) = \text{mp}(S'),\]
\[\text{methType}(T', m) = \mathbf{y:U \rightarrow void,}\]
\[\mathbf{this:T', z:U \mid - S'},\]
\[T \leq T',\]
\[S' \text{ doesn't assign to } \mathbf{y},\]
\[S = S' [x,z/\text{this},\mathbf{y}],\]
\[\Gamma, x:T' \mid - P \{ S \} Q\]

\[\Gamma, x:T \mid - P \&\& x \text{ instanceof } T' \{ x.m(z); \} Q\]
Verification of Sample

Composite root = new Composite();
Composite child = new Composite();
Component comp = new Component();
//@ assume root.total == 1 && child.total == 1;
//@ assume comp.total == 1;
//@ assume root.parent == null && child.parent == null;
//@ assume comp.parent == null;

root.addComponent(child);
child.addComponent(comp);
//@ assert root.total == 3;
root.addComponent(child);

\[
\rightarrow
\]

\text{normal\_behavior}
\begin{align*}
\text{requires} & \quad \text{child} \neq \text{root} \land \text{child.parent} = \text{null}; \\
\text{assignable} & \quad \text{root.components}; \\
\text{ensures} & \quad \text{root.components.length} > \text{root.count};
\end{align*}

\text{normal\_behavior}
\begin{align*}
\text{assignable} & \quad \text{child.parent, root.objectState}; \\
\text{ensures} & \quad \text{child.parent} = \text{root}; \\
\text{ensures} & \quad \text{root.hasComponent(child)}; \\
\text{root.addToTotal(child.total)};
\end{align*}
root.addToTotal(child.total);

\[
\begin{align*}
\{ & \text{normal\_behavior} \\
& \text{requires} \ 0 \leq \text{child.total}; \\
& \text{assignable} \ root.total; \\
& \text{ensures} \ root.total = \text{\textbackslash old}(root.total) + \text{child.total}; \\
\end{align*}
\]

Component aParent = root.parent;
while (aParent != null) {

\[
\begin{align*}
\{ & \text{normal\_behavior} \\
& \text{assignable} \ aParent.total, aParent; \\
& \text{ensures} \ aParent.total \\
& \quad = \text{\textbackslash old}(aParent.total) + \text{child.total}; \\
& \text{ensures} \ aParent = \text{\textbackslash old}(aParent.parent); \\
\end{align*}
\]
Discussion

 Argument exposure in invariant:
  - Exposed code shows how total is updated
  - Precondition of addComponent ==> no cycles
  - Special case of visibility technique

 Subtypes and overriding methods:
  - Inherit model program specifications
  - So implementations must refine

 Subtypes and new methods:
  - Inherit invariant
  - Must not make cycles (not specified!)
Conclusions

- Simple specification technique
  - Exposing code needed for invariant
  - Hiding rest of the code
- Simple verification technique
  - Copy rule
  - Limited depth of recursion can be handled
Questions?

jmlspecs.org

Thanks to David Naumann.
Other Solutions

Hallstrom and Soundarajan:
- Requires calls to operation() like our calls to addToTotal()
- Can’t specify states in which calls made
- other specs like JML’s history constraints
- More sophisticated mathematics
Other Solutions

Bierhoff and Aldrich:
- Abstraction of functional behavior
- Careful treatment of aliasing issues
Other Solutions

Jacobs, Smans, Piessens:
- Doesn’t expose any code
- More mathematically sophisticated
  - Specifies effects on parents using context
- Only treats binary trees