

# Local Reasoning and Dynamic Framing for the Composite Pattern and its Clients (October 9, 2009)

Stan Rosenberg<sup>1</sup>, Anindya Banerjee<sup>2</sup>, and David A. Naumann<sup>1</sup>

<sup>1</sup> Stevens Institute of Technology, Hoboken NJ 07030, USA

<sup>2</sup> IMDEA Software, Madrid, Spain

**Abstract.** The Composite design pattern poses a challenge for reasoning about invariants with non-local dependencies and its verification has been posed as an exemplar of specification and verification challenges for sequential object-oriented programs. Region logic is a Hoare logic augmented with state dependent 'modifies' specifications or 'dynamic frames' based on simple notations for object sets. Using ordinary first order logic assertions, it supports local reasoning and also the hiding of invariants on encapsulated state, in ways similar to separation logic but suited to off the shelf theorem provers. Using region logic, this paper provides a specification pattern for the Composite design pattern and verifies a representative implementation. To evaluate efficacy of the specification, it is used in verifications of several sample client programs. Verification is done by means of the Boogie verification condition generator which serves as front end to the Z3 SMT solver.

## 1 Introduction

The Composite pattern [9] captures a commonplace idiom in program design. The pattern centers on a collection of mutable data objects organized hierarchically, forming a rooted and possibly ordered tree. The operations include the addition and removal of subtrees anywhere in the tree. In contrast with the use of a tree as encapsulated representation for an abstract set, this pattern exposes an interface that allows clients to directly access every node. The pattern is featured in a recent survey of challenges for reasoning about sequential object-oriented programs [13] and was the challenge problem of a workshop [20]. In this paper we present a novel solution well suited to current verification tools: indeed we machine-check the verification of the pattern and some sample clients using the Z3 SMT solver [7] via its Boogie [3, 8] front end.

The usual presentation of the Composite pattern involves two classes: class *Component* has subclass *Composite*, and the latter maintains a set of children of type *Component*. For brevity we sometimes refer to objects of type *Component* as *nodes*. Any particular use of the Composite pattern will involve application-specific operations, often supported by invariants that involve many or all of the nodes. The challenge problem [13, 20] is an illustrative example. There is an operation, *getTotal*, that returns the number of descendants of a given node, counting the node itself. Method *getTotal* is declared in *Component*, because one purpose of the pattern is to provide clients with a single interface for components, whether or not they are composite. If *getTotal* is invoked more often than adding and removing subtrees, it may be desirable to declare in *Component*

an integer field, *total*, and to maintain the invariant that each node’s *total* is the number of descendants of the node. An invocation  $n.add(p)$ , which adds component  $p$  as child of composite  $n$ , increases the number of descendants of node  $n$  and of each of its ancestors. Method *add* must reestablish the ancestors’ invariants and the challenge problem is how to streamline the specification and verification.

An attractive technique for reasoning about object-oriented programs is to focus on *object invariants*, declared in classes and pertaining to each instance individually. For an example, suppose *Composite* declares field *children* which is a sequence of objects. Define the parameterized predicate

$$ok(o): \quad o.total = 1 + (\text{sum } i; 0 \leq i < \text{len}(o.children) \mid o.children[i].total)$$

where  $\text{len}(o.children)$  denotes the length of sequence  $o.children$ . This has the attractive feature of being “local” to node  $o$  and its children. Moreover, if every  $o$  of type *Composite* satisfies  $ok(o)$  then each  $o.total$  is in fact the number of descendants. This is a *decentralized* way to capture a non-local fact.

The beauty of this formulation (stipulated in [13]) is that it does not involve induction, which makes it amenable to automated first order reasoning. The notion of sequence sum, however, is inductive. In other work this is avoided by treating composites as having exactly two children, as it is not the central issue of the challenge. For us, using the recursive definition of sum will serve to provide contrast with the decentralized style of formulation.

Because adding  $p$  as child of  $n$  falsifies  $ok$  for ancestors of  $n$ , class *Component* includes field *parent*: *Composite* (with “protected” visibility). Parent pointers can be traversed in order to fix the invariant at each ancestor. What forces the implementation of *add* to fix the invariants of ancestor nodes? What lets us conclude that no other node’s *total* needs to be updated? How can the specifications be formulated so that clients are neither able to break the invariant nor directly responsible for maintaining it? We shall answer these questions without going beyond first order logic.

For some design patterns, reasoning about object invariants can be based on the idea that a client-visible object “owns” its *reps*, i.e., the objects that comprise its internal representation [21]. A discipline is imposed to ensure that the object invariant depends only on the reps and that clients cannot update the reps, so clients cannot falsify a candidate invariant. Thus the invariant may be *hidden* in the sense that it is not mentioned in the public specification of a method like *add*, but may be assumed as precondition in verifying the implementation of *add* (and must be asserted as postcondition) [10]. Ownership also supports reasoning that is “local” to the relevant part of the heap. A client can reason that the value of some query method invocation  $o.m()$  is preserved over updates of some distinct object  $o'$ , if  $o.m()$  is known to depend only on the reps of  $o$  and moreover distinct objects have disjoint reps.

The Composite pattern was posed as a challenge problem because client access to internal nodes of a tree, rather than just the root, is incompatible with ownership disciplines. There are other design patterns, such as Observer and Iterator [9], that do not fit well with ownership due to back and forth dependencies and reentrant callbacks. Proposed extensions of ownership that support hiding of invariants in these patterns [18, 22] seem ad hoc. The difficulties led some researchers to abandon the traditional notion

of hiding encapsulated invariants [10] in favor of making them explicit in contracts, abstracted in some way for information hiding [5]. But in practice many client programs manipulate a number of objects and thus explicit reasoning about invariants, even as abstract predicates, could impose a large burden.

Procedure specifications are often phrased in terms of an *effect* (or “modifies”) clause, separate from the designated postcondition (“ensures” clause), which lists the variables that may be written by the procedure. To deal with anonymous objects in the heap, and to hide effects on objects not supposed to be visible to clients, one technique is for the semantics of specifications to allow owned objects to be updated even when not explicitly mentioned in a write effect. Kassios [12] introduced a much more general technique, not for hiding but rather for abstracting from write effects. Auxiliary (“ghost”) state is used in expressions that denote sets of locations—for example, an object field *reps* may be used to hold the locations of the fields of all its rep objects—and such expressions are used in effect specifications. This is called *dynamic framing* because the locations on which an effect is allowed may be designated by an expression involving mutable ghost variables or fields of type “set of location”. Others have explored this idea using pure methods that return sets of locations [26].

In previous work [2], we formalized dynamic frames in *region logic*, a straightforward adaptation of Hoare logic that allows ghost fields/variables of type *region* in effect clauses. Regions are sets of object references (hence already present in OO languages, e.g., Java’s type `Set<Object>`). Our assertion language and effect clauses feature expressions of the form  $G^f$  where  $G$  is itself a region expression and  $f$  a field name (or data group [15]). As an r-value,  $G^f$  is the set of values in  $f$ -fields of objects in  $G$ . Effect specifications refer to the l-value, i.e., the locations of those fields. It was a pleasant surprise to find that the naïve use of regions in effect specifications enables local reasoning about the Composite pattern, in the sense of focusing on a small part of the heap. We find that regions support hiding and abstraction, without recourse to induction, second order logic [5], or method calls in specifications [26].

Using a simplified version of our specification of the Composite pattern, Sect. 2 reviews the basics of region logic. It also introduces an unusual feature of our approach, the use of “global invariants” that explicitly quantify over many objects rather than hiding that quantification via a built-in notion of object invariant. In Sect. 3 we discuss semi-automated verification in our approach, i.e., using an automated prover for code annotated with loop invariants and other assertions.

A number of publications on reasoning about design patterns [4, 11, 24] focus on verifying the classes that make up the pattern. But the test of specifications is in their use by clients. Sect. 4 refines our specifications and shows their use in local reasoning about nontrivial clients that manipulate several composites. Information hiding and abstraction are addressed in Sect. 5. Sect. 6 discusses our experience and related work.

Our main contribution amounts to a *specification pattern*, to guide program designers in specifying particular incarnations of the Composite design pattern. All of our code has been mechanically verified using Z3, an SMT solver, via its Boogie front end, and a skillfully crafted axiomatization of sequence sum from Leino and Monahan [17]. The complete code is at [www.cs.stevens.edu/~naumann/compVerf.tgz](http://www.cs.stevens.edu/~naumann/compVerf.tgz).

$$\begin{aligned}
I0 &\triangleq \forall o: \text{Comp} \mid o.\text{total} \geq 1 \\
I1(r:\text{rgn}) &\triangleq \forall o: \text{Comp} \in \text{alloc} - r \mid \text{ok}(o) \\
I2 &\triangleq \forall p: \text{Comp}, o: \text{Comp} \mid o \in p.\text{children} \Leftrightarrow o.\text{parent} = p \\
I3 &\triangleq \forall o: \text{Comp}, i:\text{int}, j:\text{int} \mid 0 \leq i < j < \text{len}(o.\text{children}) \Rightarrow (o.\text{children}[i] \neq o.\text{children}[j]) \\
I(r:\text{rgn}) &\triangleq I0 \wedge I1(r) \wedge I2 \wedge I3 \quad \text{and} \quad I \triangleq I(\text{emp})
\end{aligned}$$

```

public class Comp{
  seq(Object) children; int total; // initially total = 0 and children is empty sequence
  Comp parent; // initially parent = null

  public void add(Comp c)
    requires c  $\neq$  null  $\wedge$  c  $\neq$  self  $\wedge$  c.parent = null  $\wedge$  I
    ensures c.parent = self  $\wedge$  I
    effects wr (c) parent, (self) children, alloc total
    { c.parent := self; self.children := [c] + self.children; self.addToTotal(c.total); }

  private void addToTotal(int t)
    requires self  $\neq$  self.parent
    requires self.total = 1 - t + sum i; 0  $\leq$  i < len(self.children) | self.children[i].total
    requires I(self)
    ensures I(emp)
    effects wr alloc total
    { Comp p; int prv_total;
      p := self;
      while (p  $\neq$  null)
        inv I(p)
        inv p  $\neq$  null  $\Rightarrow$  p  $\neq$  p.parent
        inv p  $\neq$  null  $\Rightarrow$  p.total = 1 - t + sum i; 0  $\leq$  i < len(p.children) | p.children[i].total
        { prv_total := p.total; p.total := prv_total + t; p := p.parent; } }

  public int getTotal()
    requires I
    ensures result = self.total  $\wedge$  I
    { result := self.total; }
}

```

Fig. 1. Composite pattern: preliminary specifications and implementation.

## 2 Region logic in a composite nutshell

In this section, we consider a simple illustrative implementation of the Composite pattern accompanied by specifications in region logic. Salient features of region logic are introduced as we explain the specifications.<sup>1</sup>

*Implementation.* Fig. 1 depicts a simple implementation of the composite pattern. The pattern centers on a collection of mutable data objects organized as a tree. Class *Comp* is used to represent leaf nodes as well as internal nodes of the tree. Field *parent* contains

<sup>1</sup> For a more thorough exposition of region logic please refer to [2]. In the journal version (under preparation) we generalize and simplify some of the features of the logic, and those changes are also adopted herein.

an immediate ancestor (if any) of the current object (**self**) and field *total* contains a count of all descendants including **self**. Field *children* is a sequence of objects. Unlike an array, a sequence is not heap-allocated. Addition of an element to a sequence is performed using the  $+$  operation. Sequence membership is written as  $o \in p.children$ , which is a shorthand for  $\exists i: \mathbf{int} \mid 0 \leq i < len(p.children) \wedge o = p.children[i]$ .

Note that the specifications in Fig. 1 are preliminary; later we revise them to provide more precise write effects and to hide some of the invariants.

*Specification of add.* Public method *add* inserts an existing composite into the children of **self** and invokes private method *addToTotal* that updates the total of **self** and all of its ancestors (if any).

As usual **requires** and **ensures** clauses express pre- and postconditions. The **effects** clause expresses write effects, that is, what variables and fields (of objects in *add*'s pre state) may be written. In other words write effects describe a command's footprint. We list write effects following keyword **wr**. A *region* is a set of references; region expressions have type **rgn** and can occur in assertions and in effects. Region expression,  $\langle E \rangle$ , (singleton region) where  $E$  is an expression of reference type, denotes a set containing the single object, possibly null, denoted by  $E$ . For any region expression,  $G$ , the notation  $G^f$  (read:  $G$  image  $f$ ) designates the  $f$  fields of objects in  $G$ . This notation is used in the effect specification **wr**  $G^f$  (c.f., **wr**  $\langle c \rangle^parent$  in Fig. 1) that says the  $f$ -fields of  $G$ -objects may be written. In contexts where a region expression is expected, like **wr**  $G^f h$ , the meaning of  $G^f$  is the region of all well-defined references  $o.f$  such that  $o \in G$ . (In case  $f: \mathbf{rgn}$ , it is a union of all well-defined regions  $o.f$ .) The assertions  $G \subseteq G'$  and  $G \# G'$  say, respectively, that region  $G$  is a subset of  $G'$  and  $G \cap G' \subseteq \{\mathbf{null}\}$ . In particular,  $G^f \subseteq G$  says  $G$  is closed under  $f$  and  $G^f \# G'$  says that  $G'$  is disjoint from  $G$ 's  $f$ -image (but allows null in the intersection). The dual of write effects is read effects. Whereas write effects express a footprint of a command, read effects express a frame of an assertion, that is, variables and fields whose modification may cause a change in the assertion's denotation. Read effect, **rd**  $G^f$ , of an assertion says that the meaning of the assertion can vary with updates to  $f$  fields of  $G$ -objects, i.e., it depends on those fields.

In quantified assertions such as  $I0$  (see Fig. 1), the bound variable ranges over allocated (thus non-null) references only. The default range is **alloc**, i.e., the region of all allocated references,<sup>2</sup> but a smaller range can be given as bound. Thus in  $I2$ , both  $p$  and  $o$  range over **alloc** whereas in  $I1(r)$ ,  $o$  ranges over all objects in the region **alloc**  $- r$ , where  $-$  denotes set subtraction. We sometimes write  $I1$  for  $I1(\mathbf{emp})$  and  $I$  for  $I(\mathbf{emp})$ .

Leaving aside condition  $I$ , the specification of *add* says: Given an initial state where  $c$  is an allocated component distinct from **self** and has no ancestors ( $c.parent = \mathbf{null}$ ), a final state is one in which  $c$ 's parent is **self**. Furthermore the following updates (but no other) are licensed by the write effects: *parent* field of  $c$ ; *children* field of **self** and the *total* field of any allocated component (**alloc** <sup>$f$</sup>  *total*). Condition  $I$  is intended to be invariant in the sense that it holds in all client-visible states; so it appears as both pre-

<sup>2</sup> The semantics is instrumented in that newly allocated objects are automatically added to **alloc**. A command that allocates must report effect **wr alloc**.

and postcondition of *add* and *getTotal*. The conjunct *I0* says every component's *total* is positive; *I1*(*r*) says every component except those in *r* has as *total* one more than the sum of its children's *total*; *I2* says *p* is *o*'s unique parent iff *o* is *p*'s child; *I3* says *children* does not have any duplicates. In conjunction with the invariant, *I*, the spec of *add* says that *c* was added to *children* of **self**: initially, *c.parent* = **null** and *I2* together entail  $c \notin \mathbf{self}.children$ ; finally, *c.parent* = **self** and *I2* together entail  $c \in \mathbf{self}.children$ . We remark that the precondition of *add* does not preclude the creation of a multi-node cycle; e.g., consider *b.add(a)* where  $a \neq b$  and *a.parent* = **null** but *b.parent* = *a*. In this case the call to *add* will diverge! Nonetheless the code is partially correct. The preconditions in Sect. 4 will prevent such calls.

*Proof system by example.* In a nutshell, the proof system of region logic consists of “local rules” empowered by the **Frame** rule. The formal details can be gleaned from [2]; here we explain those informally. Let's consider proving a part of the spec of *add*, in particular, establishing the assertion  $\mathbf{self} \neq \mathbf{self}.parent$  (no self-cycle)<sup>3</sup> which is needed immediately before the invocation of *addToTotal*. By standard rule of **Consequence**, we can show that the precondition of *add* (specifically,  $I0 \wedge I1(\mathbf{emp}) \wedge I2$ ) establish the assertion. However, the local specification (“small axiom”) of *c.parent* := **self** says something quite different (and nothing about the assertion  $\mathbf{self} \neq \mathbf{self}.parent$ ):

$$\{c \neq \mathbf{null}\} c.parent := \mathbf{self} \{c.parent = \mathbf{self}\} [\mathbf{wr} \langle c \rangle^c parent]$$

It says: if *c* is allocated in the pre-state, then in the post-state, *c.parent* = **self** and the write effect  $\langle c \rangle^c parent$  licenses the update. (Observe locality at play: the rule refers only to immediate state of the assignment at hand: *c*, **self** and  $\langle c \rangle^c parent$ .) Intuitively, one can deduce that an assertion that does not “depend” on the write effect must be preserved by the field update: the truth of  $c \neq \mathbf{self}$  is unaffected by update of *c.parent*. Consequently, we can conjoin  $c \neq \mathbf{self}$  to the pre- and postconditions:

$$\{c \neq \mathbf{null} \wedge c \neq \mathbf{self}\} c.parent := \mathbf{self} \{c.parent = \mathbf{self} \wedge c \neq \mathbf{self}\} [\mathbf{wr} \langle c \rangle^c parent]$$

However, we have to be cautious about conjoining the assertion  $\mathbf{self} \neq \mathbf{self}.parent$ : we have to show that its truth is unaffected despite the update to *parent*. But this follows because  $c \neq \mathbf{self}$  holds in the precondition. Consequently the field update is to the *parent* field of an object (*c*) different from the object (**self**) under consideration in the assertion. Thus it is sound to conjoin  $\mathbf{self} \neq \mathbf{self}.parent$  to the pre- and postconditions.

The above informal discourse is justified by the **Frame** rule of region logic,

$$\frac{\vdash \{P\} C \{P'\} [\bar{\epsilon}] \quad P \vdash \bar{\delta} \mathbf{frm} Q \quad P \Rightarrow \bar{\delta} \star \bar{\epsilon}}{\vdash \{P \wedge Q\} C \{P' \wedge Q\} [\bar{\epsilon}]}$$

read:  $\bar{Q}$  is preserved by *C* under precondition *P* if  $\bar{\epsilon}$ , the write effects of *C*, is separate from  $\bar{\delta}$ , the read effects of *Q*. The *frames judgement*  $P \vdash \bar{\delta} \mathbf{frm} Q$  asserts  $\bar{\delta}$  are indeed the read effects of *Q*. (We use syntax-driven analysis for read effects of atomic assertions, and an inductive definition of this judgement for all other formulas [2].)

<sup>3</sup> This condition is crucial to establish  $I1(\mathbf{emp})$ . Note that *add*'s specs. prevent *self-cycles*.

The antecedent  $P \Rightarrow \bar{\delta} \star \bar{\varepsilon}$  asserts that the precondition may be assumed to prove that read effects are separate from the write effects. Formally, we call  $\bar{\delta} \star \bar{\varepsilon}$  a *separator*. The function  $\star$  computes a conjunction  $Q$  of disjointness formulas such that in  $Q$ -states, writes allowed by  $\bar{\varepsilon}$  cannot falsify a formula framed by  $\bar{\delta}$ . (Frames judgements in conjunction with separators formalize the notion of dependence—whether or not an assertion may depend on write effects.)

In the preceding, the read effects of  $c \neq \mathbf{self}$  and  $\mathbf{self} \neq \mathbf{self.parent}$  are  $\mathbf{rd} \ c, \mathbf{self}$  and  $\mathbf{rd} \ \mathbf{self}, \langle \mathbf{self} \rangle^{\mathbf{c}} \mathbf{parent}$ , respectively. To conjoin  $c \neq \mathbf{self}$  by  $\mathbf{Frame}$  we needed to discharge  $\mathbf{rd} \ c, \mathbf{self} \star \mathbf{wr} \ \langle c \rangle^{\mathbf{c}} \mathbf{parent}$  which evaluates to  $\mathbf{true}$ ; to conjoin  $\mathbf{self} \neq \mathbf{self.parent}$  we needed to discharge  $\mathbf{rd} \ \mathbf{self}, \langle \mathbf{self} \rangle^{\mathbf{c}} \mathbf{parent} \star \mathbf{wr} \ \langle c \rangle^{\mathbf{c}} \mathbf{parent}$  which yields  $\langle \mathbf{self} \rangle \# \langle c \rangle$ . Now precondition  $c \neq \mathbf{self}$  implies  $\langle \mathbf{self} \rangle \# \langle c \rangle$ .

### 3 Region logic can Boogie: automated verification

We describe how to encode programs specified in region logic as BoogiePL programs thereby arriving at an automated verification platform. We also share our experience with the encoding and verification as it pertains to the Composite.

*Boogie.* The Boogie [3] verification platform consists of a procedural verification language BoogiePL (bpl) [8], a verification condition generator (VCGen) and a first-order theorem prover. Given a specified bpl program, and a list of procedures to verify, VCGen computes the weakest precondition of each specified procedure relative to its implementation; these verification conditions (VCs) together with the background predicate, which axiomatizes the semantics of BoogiePL, as well as any additional user defined axioms are handed off to a prover, such as Z3.

A bpl program typically consists of global variables, procedures, special commands (e.g., **assume**, **assert**, **havoc**), uninterpreted functions and axioms. Procedure specifications consist of pre-/postconditions and write effects (of global variables). Specifications are two-state, allowing a postcondition to refer to the pre state by way of **old**; e.g.,  $\mathbf{old}(x) = x$  equates the values of  $x$  in the pre- and post states. BoogiePL comes equipped with some basic types: **bool**, **int**, **ref**, **name** (for field names), as well as map types (corresponding to the theory of arrays). For example,  $[\mathbf{ref}] \mathbf{bool}$  denotes a map,  $\mathbf{ref} \rightarrow \mathbf{bool}$ ; the map also encodes a set of references, that is, a region. There is also an encoding of sequences as maps that we take from the Boogie distribution.

*Encoding region logic.* First, we expose the heap, by declaring a global variable *Heap* which is a map indexed by (**ref**, **name**) pairs. To track allocation, we declare a global field *alloc*; e.g.,  $\mathbf{Heap}[o, \mathbf{alloc}] = \mathbf{true}$  denotes that  $o$  is allocated. We also make sure that *Heap* is well defined, i.e., contains no dangling references, by generating appropriate axioms. Field access and field update statements can now be expressed in terms of *Heap*: e.g.,  $x.f := y$  is encoded as  $\mathbf{Heap}[x, f] := y$ .

The translation of region assertions and hence pre- and postconditions is straightforward using the above representation. To translate write effects, including for example  $\mathbf{wr} \ G^{\mathbf{c}} f$ , we generate the following postcondition in bpl:

$$\forall o : \mathbf{ref}, g : \mathbf{name} \mid o \neq \mathbf{null} \wedge \mathbf{old}(\mathbf{Heap})[o, \mathbf{alloc}] \Rightarrow \\ \mathbf{Heap}[o, g] = \mathbf{old}(\mathbf{Heap})[o, g] \vee (\mathbf{old}(G)[o] \wedge g = f)$$

That is, for any object  $o$ , allocated in the pre state, and any field  $g$ , if the value  $o.g$  has changed, then  $o$  must belong to  $G$ , evaluated in the pre state, and  $g$  must be  $f$ . There will be additional disjuncts if there are additional write effects. When **wralloc** is not contained in the write effects, we also generate the postcondition

$$\forall o: \mathbf{ref} \mid o \neq \mathbf{null} \wedge \mathit{Heap}[o, \mathit{alloc}] \Rightarrow \mathbf{old}(\mathit{Heap})[o, \mathit{alloc}]$$

saying that no allocation transpired.

*Framing axiom.* Recall from Sect. 2 that one of the antecedents of the Frame rule is a frames judgement,  $P \vdash \bar{\delta} \mathbf{frm} Q$ . The denotation of a frames judgement is key to the soundness of Frame; it says, if pre and post states agree on  $\bar{\delta}$ , and  $P$  holds in the pre state then  $Q$  has the same truth value in both states. As we shall see shortly, such judgements (recast as axioms in bpl) are indispensable when reasoning with recursively defined functions in Boogie. Here is the issue. As usual in modeling the heap using an array, updates result in case splits [6] on whether a given location was updated or not. In particular, reasoning about preservation of our quantified invariants requires case splits on whether the relevant objects were updated. In the worst case, the number of such case splits is exponential. An orthogonal problem is that an invariant may contain an uninterpreted function,  $f$ , whose value depends on the heap, and whose defining axioms are recursive. (C.f.,  $ok(o)$  in Fig. 1 where the definition of sequence sum is recursive.) Consequently, the prover is confronted with establishing  $f(h') = f(h)$ , where  $h, h'$  refer to the initial and final heaps resp. This may not be tractable, e.g., due to recursion. Here is where a frames judgement, appropriately encoded as a *framing axiom* expressing a two state relation can be utilized. We explain by way of the  $ok$  predicate.

The frames judgement corresponding to  $ok(o)$  can be derived as,

$$\mathbf{true} \vdash \mathbf{rd} \ o, \langle o \rangle \text{‘} children, o.children \text{‘} total \ \mathbf{frm} \ ok(o)$$

The framing axiom is

$$\begin{aligned} \forall h1, h2, o \mid h1[o, children] = h2[o, children] \wedge \\ (\forall p \mid p \neq \mathbf{null} \wedge p \in h1[o, children] \Rightarrow h1[p, total] = h2[p, total]) \Rightarrow \\ ok(h1, o) = ok(h2, o) \end{aligned}$$

It says: If two heaps,  $h1, h2$ , agree on  $o.children$ , and on the  $total$  fields of each allocated  $p$  in  $o.children$ , then  $ok(o)$  has the same value in both heaps. Note that the soundness of the axiom is a direct consequence of the frame agreement lemma [2, Lemma 4]. By analogy with the Frame rule, we sometimes want the prover to instantiate the axiom with  $h1$  as the heap before an update and  $h2$  as the heap afterwards. Then instead of reasoning about the recursion in  $ok$ 's definition, the prover need only establish that  $h1, h2$  agree on the read effects. To achieve this effect we annotate the code with an assertion following the update, which is judiciously chosen to trigger the prover to use the framing axiom.

*Our experience.* A quantified formula in bpl can be annotated with *triggers* [17]—any set of terms that determines how the SMT solver instantiates the quantifiers. These

terms limit the number of quantifier instantiations. In the framing axiom above, we used the triggers  $ok(h1, o)$  and  $ok(h2, o)$ . Thus, the framing axiom is instantiated with a term  $t$  only if the terms  $ok(h1, t)$  and  $ok(h2, t)$  occur in the *e-graph* [17]. By placing **assert**  $I1(h1)$ ; **assert**  $I1(h2)$  at an opportune location, we ensure that the triggers are enabled, i.e., there are terms in the e-graph which match the triggers. Our experience is that finding a “right” set of triggers can be quite tricky, yet it is arguably a well-spent effort since the end result is fully automatic.

Another challenge we have encountered is the axiomatization of sequence sums, the only recursive ingredient in our specifications. Our axiomatization uses Leino’s and Monahan’s [17] axioms for set comprehensions. The axioms are necessarily incomplete but capture sufficient properties to be useful in practice, as evidenced in [17]. We needed to add an axiom (provable by induction but not automatically) to express that under  $I0$ , sequence sums are non-negative. As with the triggering of the framing axiom, we needed additional **assert** statements to help guide the prover.

#### 4 Smaller footprints for client reasoning

Consider a simple client program:  $a.add(b)$ , where  $a, b: Comp$ . By method call rule, we substitute actuals  $a$  and  $b$  for formals **self** and  $c$  resp. in the specification of  $add$  in Fig. 1, to obtain

$$\{ P \} a.add(b) \{ P' \} [\bar{\epsilon}]$$

where  $P: b \neq \mathbf{null} \wedge b \neq a \wedge b.parent = \mathbf{null}$

$P': b.parent = a$

$\bar{\epsilon}: \mathbf{wr} \langle b \rangle^{\mathbf{parent}}, \langle a \rangle^{\mathbf{children}}, \mathbf{alloc}^{\mathbf{total}}$

and we elide the invariant. A client could appeal to **Frame** to show, e.g.,  $a.parent = x$  is preserved: the obligation is to show  $P \Rightarrow \mathbf{rd} \langle a \rangle^{\mathbf{parent}}, \mathbf{rd} x \star \bar{\epsilon}$  which amounts to  $P \Rightarrow \langle a \rangle \# \langle b \rangle$ . The disjointness evaluates to **true** using  $P$ .

On the other hand reasoning about  $total$  will not work because the effect,  $\mathbf{wr} \mathbf{alloc}^{\mathbf{total}}$ , is too coarse. In detail, if the predicate to be preserved is  $b.total = t$ , then **Frame** requires establishing the disjointness  $\langle b \rangle \# \mathbf{alloc}$  — which is patently false.

We now consider clients that want to reason about  $total$  across calls to the  $add$  method. Our solution is based on exposing smaller, fine-grained footprints to the client. Consider composites  $c0, \dots, c4$  and client code

```
tBefore := c2.getTotal();
c0.add(c1); c1.add(c2); c0.add(c3); c3.add(c4);
tAfter := c2.getTotal();
assert tBefore = tAfter; // c2's total is preserved
```

Here the composite tree at  $c0$  is updated so that  $c2$  is a child of  $c1$  which in turn is a child of  $c0$ ; similarly  $c3$  is a child of  $c0$  and  $c4$  is a child of  $c3$ . To show the preservation of  $c2$ 's  $total$ , the key information that the client needs is disjointness: roughly, the trees need to have disjoint descendants and  $c1, \dots, c4$  must be roots (i.e., their parents are **null**). Moreover,  $c2$  must be disjoint from ancestors of  $c0$  or  $c1$  or  $c3$  —and *it is only the total field of these ancestors that ever gets written by the calls to add*. Accordingly we

**requires**  $c \neq \mathbf{null} \wedge c.\mathit{parent} = \mathbf{null} \wedge c.\mathit{root} \neq \mathbf{self}.\mathit{root} \wedge I \wedge J \wedge K$   
**ensures**  $c.\mathit{parent} = \mathbf{self} \wedge I \wedge J \wedge K$   
**effects**  $\mathbf{wr} \langle c \rangle \langle \mathit{parent}, \mathit{ancestors}(\mathbf{self}) \langle \mathit{total}, \mathit{desc} \rangle, c.\mathit{desc} \langle \mathit{root}, \langle \mathbf{self} \rangle \langle \mathit{children} \rangle$

$\mathit{ancestors}(o : \mathit{Comp}) : \{p \mid o \in p.\mathit{desc}\}$

$J0 : \forall o : \mathit{Comp} \mid o.\mathit{desc} \langle \mathit{desc} \rangle \subseteq o.\mathit{desc}$

$J1 : \forall o : \mathit{Comp} \mid o \in o.\mathit{desc}$

$J2 : \forall o : \mathit{Comp} \mid o.\mathit{desc} \subseteq o.\mathit{root}.\mathit{desc}$

$J3 : \forall o : \mathit{Comp}, p : \mathit{Comp}, q : \mathit{Comp} \mid o \in p.\mathit{desc} \wedge o \in q.\mathit{desc} \Rightarrow p.\mathit{root} = q.\mathit{root}$

$J4 : \forall o : \mathit{Comp} \mid o.\mathit{parent} = \mathbf{null} \Rightarrow o.\mathit{root} = o$

$J5 : \forall o : \mathit{Comp} \mid o.\mathit{parent} \neq \mathbf{null} \Rightarrow o.\mathit{root} = o.\mathit{parent}.\mathit{root} \wedge o \in o.\mathit{parent}.\mathit{desc}$

$J6 : \forall o, p : \mathit{Comp} \mid o \in p.\mathit{desc} \wedge o \neq p \Rightarrow p.\mathit{parent} \neq \mathbf{null} \wedge o.\mathit{parent}.\mathit{desc} \subseteq p.\mathit{desc}$

$J : J0 \wedge J1 \wedge J2 \wedge J3 \wedge J4 \wedge J5 \wedge J6$

$K : \forall o : \mathit{Comp} \mid \forall i : \mathbf{int} \mid 0 \leq i < \mathit{len}(o.\mathit{children}) \Rightarrow o.\mathit{children}[i] \in o.\mathit{desc}$

**Fig. 2.** Strengthened spec. of *add* and definitions of ancestors and invariants.

need a revised spec. for *add* that uses ghost state to pin down ancestors and descendants. These specs. appear in Fig. 2 together with invariants  $J, K$  that are critical in providing disjointness information.

*Specs. of add.* Ghost field  $\mathit{desc} : \mathbf{rgn}$  keeps track of the set of descendants of a node and  $\mathit{root}$  denotes the root of a composite tree. Ancestors are defined in terms of descendants. Invariants  $J0, J1$  constrain  $\mathit{desc}$  to be a reflexive, transitive relation;  $J2$  states that descendants of  $o.\mathit{root}$  include those of  $o$ ; <sup>4</sup>  $J3$  states that components with distinct roots have disjoint descendants;  $J4, J5$  constrain every  $\mathit{parent}$  path to have the same root;  $J6$  says that for any  $o$  which is a *proper* descendant of  $p$ ,  $o.\mathit{parent}.\mathit{desc}$  must be included in  $p.\mathit{desc}$ , whence by  $J1$ ,  $o.\mathit{parent} \in p.\mathit{desc}$ .  $K$  constrains  $\mathit{desc}$  to include  $\mathit{children}$ . The postcondition of *add* is the same as before, modulo  $J \wedge K$ . However, note that  $c \in \mathbf{self}.\mathit{root}.\mathit{desc}$  is entailed by the postcondition. (From  $c.\mathit{parent} = \mathbf{self}$  and  $J5$ , we obtain  $c \in \mathbf{self}.\mathit{desc}$ ;  $J2$  finishes the proof.) Finally, the most precise write effect wrt.  $\mathit{total}$  is  $\mathbf{wr} \mathit{ancestors}(\mathbf{self}) \langle \mathit{total} \rangle$ . It says that *add* may modify  $\mathit{total}$  of every ancestor of  $\mathbf{self}$  (including  $\mathbf{self}$ ).

Note how multi-node cycles are precluded by way of precondition  $c.\mathit{root} \neq \mathbf{self}.\mathit{root}$ , which, together with  $J3$  entails that  $c$ 's descendants are disjoint from  $\mathbf{self}$ 's descendants.

*Client verification.* We have mechanically verified the client above using the spec. of *add* in Fig. 2. The reasoning is this. Suppose that before the first call to *add* the client code establishes  $J$ , and moreover that  $c1, \dots, c4$  have no parents and that the roots of  $c0, \dots, c4$  are pairwise disjoint. Then we can show the preservation of  $c2$ 's  $\mathit{total}$  as follows: Consider the first call,  $c0.\mathit{add}(c1)$ . Since  $c2.\mathit{root} \neq c0.\mathit{root}$ , by  $J3$ , we obtain  $c2.\mathit{desc} \# c0.\mathit{desc}$ , which entails  $\mathit{ancestors}(c0) \# \langle c2 \rangle$ . Consequently, we can show by Frame that  $c2$ 's  $\mathit{total}$  is preserved over  $c0.\mathit{add}(c1)$ , owing to effect

<sup>4</sup> It is a consequence of  $I, J4, J5$  that  $o.\mathit{root}$  is always non- $\mathbf{null}$  (see Sect. 6).

$\mathbf{wr} \text{ ancestors}(c0)^{\text{total}}$ . We can also verify that  $c2.\text{root}$  is distinct from the other roots. So, for the next call,  $c1.\text{add}(c2)$ ,  $c2.\text{root}$  is distinct from  $c1.\text{root}$  and we reason exactly as before to show preservation of  $c2.\text{total}$ , etc.

*Implementation of add.* There are only two changes from Fig. 1: subsequent to the addition of  $c$  to  $\mathbf{self}.\text{children}$ , we perform two bulk updates of ghost fields  $\text{desc}$  and  $\text{root}$ . The  $\text{desc}$  field of all objects in  $\text{ancestors}(\mathbf{self})$  is updated to contain  $c$ , and the  $\text{root}$  field of all objects in  $c.\text{desc}$  is updated to point to  $\mathbf{self}.\text{root}$ .

## 5 Abstraction and information hiding

Our illustrative code uses for  $\text{children}$  a mathematical sequence rather than a heap-allocated array or other practical data structure which would have its own internal representation. To cater for such a representation, while maintaining an appropriate abstraction boundary, the specifications of mutator methods like  $\text{add}$  should include the effect  $\mathbf{wralloc}$  to allow allocation of internally used objects, as well as an effect like  $\mathbf{wrself}.\text{rep}^{\text{any}}$  where  $\text{rep}$  is a region field that holds the current “owned” representation objects. (Our previous work [2] shows how to specify ownership using regions, in a way that can be combined with the specifications presented herein.) Writes to fields of  $\mathbf{self}$ , other than spec-public ones like  $\text{parent}$ , should be abstracted by a data group [15], e.g., in Fig. 2 the effect  $\mathbf{wr}(\mathbf{self})^{\text{children}}$  should be abstracted.

Effects on private fields can be hidden, i.e., omitted from specifications used by client code, and conceivably effects on owned objects could be omitted (as they are in the Boogie methodology, for example [18]). But we have not investigated such hiding of effects, because for reasoning about query methods it is useful to specify (in suitably abstracted form) what part of encapsulated state may be written by mutator methods and read by particular query methods [26, 19]. Indeed, to that end, finer grained (but still abstract) regions could be used, rather than a single pool of owned objects. This topic is not our main concern, so we chose to streamline the presentation by using mathematical sequences and omitting the effects mentioned above.

One dimension of the Composite challenge problem is information hiding. Whereas we argue for abstracting effects in interface specifications, representation invariants — of which  $I1$  is an example — should be completely hidden from clients, to streamline the specifications and avoid unnecessary proof obligations on clients. The idea is very standard: The implementation of a method is verified with respect to a contract in which the invariant is an explicit pre- and postcondition, but the invariant does not appear in the contract used to reason about clients [10]. This mismatch is justified as follows: The invariant is supposed to depend only on state that is encapsulated in some fashion that ensures clients cannot write that part of the state. This mismatch is used, for example, in systems that restrict object invariants to depend only on owned objects [3, 18]. Because ownership is widely applicable, we expect a full-functioned verification system would include streamlined use of ownership, e.g., with type-based syntax that is desugared to a region-based formulation. However, the point of the Composite example is that it does not fit with ownership and therefore needs another treatment.

As a more general technique for hiding of internal invariants, we propose that a module can declare a *dynamic encapsulation boundary*, i.e., a read effect, in suitably

abstract terms, that delimits its encapsulated state and frames the invariant to be hidden. (It is dynamic in that our effects are stateful, just like dynamic frames in method contracts.) Framing of the invariant involves nothing more than the framing judgement discussed in Sects. 2 and 3. For it to be sound to hide the invariant, client code must respect the dynamic encapsulation boundary: it is subject to the proof obligation that it does not write within the dynamic encapsulation boundary. This notion can be formalized by a second order rule of framing, as described in [2] and formalized in [1].

Our second order frame rule, adapted from separation logic, applies to a single invariant formula, which is one reason why we use formulas like  $I1$  which appear alarmingly global. As our treatment of Composite illustrates, this does not preclude that the invariant can be split into conjuncts with disjoint footprints, one of which pertains to the current receiver of a method call, so that the other can be handled by ordinary framing.

Let us consider the invariants  $I$  and  $K$ . These can be framed by the effect **rdalloc**<sup>6</sup>(*desc, parent, children, root, total*). For this to be a dynamic effect boundary, we require that clients never write any of these fields. In general, enforcement of a dynamic encapsulation boundary may require reasoning about regions, but in this case it is entirely a matter of scope. Field *children* should be private to class *Comp*. Because they are used in public contracts, the other fields need to be *private, spec-public* in the terminology of JML and similar formalisms. That is, they cannot occur in client code but are allowed in specifications visible to clients. That includes in annotation of client code—but not the target of ghost assignments in client code. So it is impossible for clients to write within the boundary, which licenses hiding  $I$  and  $K$ : Neither should appear in the specifications with respect to which clients are verified, but they are conjoined to those pre- and post-conditions for purposes of verifying the implementations of *add* and other public methods.

Invariant  $J$  is framed by **rdalloc**<sup>6</sup>(*desc, parent, root*) and indeed for reasons of scope clients respect the boundary **rdalloc**<sup>6</sup>(*desc, parent, children, root, total*). Invariant  $J$  provides information needed for reasoning about clients as in Sect. 4. So  $J$  could be exported to the client as a public invariant [14]. That is, like  $I$  and  $K$  it is omitted from the public contracts, so clients are not responsible for establishing it. But it may be assumed at any point in client code. Because this is not an entirely standard notion, we have refrained from using it in our verifications; instead, we include  $J$  as explicit pre- and post-condition in the public specification of *add*.

## 6 Discussion

*On automating local reasoning about global invariants.* In order to have a precise footprint for *add* we need to consider the ancestors of a node; ancestors are defined in terms of descendants. To reason about descendants we need universally quantified formulas with explicit ghost state (such as *desc, root*). There are two aspects to this reasoning: we need enough invariants – but not necessarily the minimal set – to get the inductive properties of interest (e.g., transitivity of descendants, and a limited form of reachability) and we need to tackle framing issues that arise because of universally quantified formulas.

The ubiquitous use of global invariants, witness the prevalence of universal quantifiers (often nested), apparently contradicts notions such as object-centric invariants, locality, or adherence to a particular programming methodology (see, e.g., the Composite verification in [27]). To our surprise, however, we have not witnessed any difficulties in verification: *add* verifies in 3.5s, *addToTotal* in 3s, and the clients in less than 1s using Boogie version 0.90. When heap updates occur, the prover automatically performs case splits on the current receiver thereby explicitly proving each instantiation of the quantified invariants that may have been affected by the update. For decentralized invariants such as *J, K* that take the form of universal quantification of local conditions (involving *desc, root, children, parent*) over arbitrary objects, our experience has been that Boogie’s VCGen works very well: Z3 has no difficulty verifying the VCs and does not need local reasoning as embodied in the Frame rule (Sect. 2). On the other hand, predicates such as *ok* are not decentralized in that they directly refer to uninterpreted functions such as sequence sum which is recursive. The verification of such predicates during a heap update is potentially problematic because it can lead to repeated unfolding of the inductive axiomatization of sum. To avoid this unfolding we have used the framing axiom (c.f. Sect. 3).<sup>5</sup> To trigger the axiom we manually guided Z3 by inserting asserts after each relevant heap update. The framing axiom is, in effect, the Boogie encoding of a key lemma [2, Lemma 4] needed to prove the soundness of region logic’s Frame rule.

*More highlights from the Composite verification.* Our verification is based on a couple of postulates, namely,  $o.root \neq \mathbf{null}$  and  $o.root.root = o.root$  for any  $o : Comp$ . The former is used in the verification of *add* and also ensures that *J2* is well defined. The latter is not used directly but its presence speeds up the verification considerably. An alternative to the postulates would be to maintain them as invariants at the cost of more verification. The postulates are sound based on offline inductive proofs (which could be mechanized, e.g., in Isabelle/HOL) that show that they are implied by  $I \wedge J4 \wedge J5$ . As a sanity check of the *I* axioms, we have used Z3 to mechanically verify the absence of self cycles: if  $ok(o)$  holds for any object  $o$ , then  $o \notin o.children$ . This fact is used in *add* in establishing  $\mathbf{self} \neq \mathbf{self.parent}$  of the precondition of *addToTotal* as well as in the second loop invariant of *addToTotal*. An interesting property of our invariants is that *desc* is constrained to be the (reflexive) transitive closure. Whereas *J0, J1* postulate that *desc* is transitive, *J6* helps pin down that *desc* contains only reachable components. We can meta-prove using induction on the length of a *parent* path, that *desc* is the *smallest*, owing to *J6* and acyclicity which follows, by induction, from the *I* invariants.

**Related work.** We draw heavily on Kassios’ [12] dynamic framing, which has been explored in a number of research efforts (e.g., [26]), as well as the frame rule and local reasoning in separation logic [23]. Because Kassios developed his ideas in a relational calculus of refinement, his effect specifications can be freely mixed with functional specifications, e.g., to express that a write effect takes place only under a certain condition. In contrast, our adoption of the popular “modifies clause” format fits with standard verification techniques but makes it awkward to express conditional effects (although it

<sup>5</sup> A similar view is expressed in Leino’s Marktoberdorf lectures [16, pp. 32].

can be done if pure methods are used to compute footprints as in [26]). In recent work, Smans et al [25] avoid the need for a modifies clause somewhat in the manner of separation logic, but instead of a non-standard connective they use special “access predicates” with a permission-based semantics and special program constructs. Inductively defined pure functions are needed to construct access footprints for recursive data structures.

The most closely related works directly address the Composite challenge. Bierhoff and Aldrich [4] achieve fully automated checking of the *add* implementation using tpestates (and no theorem proving at all) to express the *total* invariant, our *I1*, in finite state form (i.e., the parity of each *total*). Permissions and data groups are used to track dependencies between tpestates of different objects, to enforce separation and allow sharing (fractional permissions) where needed. The program needs to be instrumented with *pack/unpack* notations, to an extent similar to the ghost assignments needed in our approach. The specification notations also use operators from linear type systems. Presumably, their types and permissions could be used for reasoning about clients at the level of precision we have considered.

Jacobs et al [11] present a specification of the Composite using separation logic with a number of inductive definitions, e.g., instead of the decentralized *I1* the main invariant uses an inductive definition of the descendant count to specify the value of *total* at each node. The logic has been implemented in a tool that verifies the implementation of *add* as well as a client that constructs a tree with several nodes. A very interesting feature is that the specification describes a tree together with a focus node, to facilitate client access at any node. A “lemma function” is used in annotations to move the focus around, with the effect of folding and unfolding the inductive definition of a tree-with-focus. A dispose operation is included. Second order predicates are used for hiding, as in [5].

Shaner et al [24] address invariant *I1* and an implementation of *add* essentially like ours (which follows [13] but avoids arrays). The specification of *add* uses JML’s model program feature which stipulates the implementation must call *addToTotal* properly. The idea is to ensure preservation of a hidden invariant by specifying “mandatory calls” that must also be made in any override of a method like *add*. Framing for clients is not addressed in detail. Summers and Drossopoulou [27] verify the Composite by using a novel methodology based on object invariants. However, they do not consider clients, nor do they consider write effects. Their encoding, like that used by Müller in unpublished work on the Priority Inheritance protocol which resembles the Composite pattern, makes use of decentralized invariants on ghost state similar to ours.

*Conclusion* Bierhoff and Aldrich [4] nicely summarize the challenge of the Composite pattern: “If nodes depend on invariants over their children then it becomes challenging to verify that adding a child to a node correctly notifies the node’s parents of changes.” We have used elementary and mostly familiar means to specify the Composite pattern and to mechanically verify its implementation and its clients. The specification of the key method, *add*, is in our view fairly succinct and transparent. Its verification, and the verification of interesting client code, relies on a number of global invariants that capture inductive properties in decentralized ways.

## References

1. A. Banerjee and D. A. Naumann. A logical account of Hoare’s mismatch. Available from <https://www.cs.stevens.edu/~naumann/publications/BanerjeeNaumann09.pdf>.
2. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.
3. M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *VSTTE*, 2005.
4. K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In [20].
5. G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, 2005.
6. A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
7. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
8. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
11. B. Jacobs, J. Smans, and F. Piessens. Verifying the composite pattern using separation logic. In [20].
12. I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, volume 4085 of *LNCS*, pages 268–283, 2006.
13. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.
14. G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395, 2007.
15. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPLSA*, 1998.
16. K. R. M. Leino. Specification and verification of object-oriented software. 2008. Lecture notes for the Marktoberdorf International Summer School.
17. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order smt solvers. In *SAC*, 2009.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
19. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *ESOP*, 2008.
20. Robby et al. Seventh international workshop on specification and verification of component systems (SAVCBS). Technical Report CS-TR-08-07, School of Electrical Engineering and Computer Science, University of Central Florida, 2008.
21. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, 2006.
22. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 365:143–168, 2006.
23. P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
24. S. M. Shaner, H. Rajan, and G. T. Leavens. Model programs for preserving composite invariants. In [20].
25. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, 2009.
26. J. Smans, B. Jacobs, F. Piessens, and W. Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, 2008.
27. A. J. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. Personal communication, Sept. 18, 2009. Available from <https://www.doc.ic.ac.uk/~scd/ConsiderateReasoning.pdf>.