Local Reasoning and Dynamic Framing for the Composite Pattern and its Clients

Stan Rosenberg[⋆1], Anindya Banerjee[⋆⋆2], and David A. Naumann[⋆⋆⋆1]

[1] Stevens Institute of Technology, Hoboken NJ 07030, USA
[2] IMDEA Software Institute, Madrid, Spain

Abstract. The Composite design pattern is an exemplar of specification and verification challenges for sequential object-oriented programs. Region logic is a Hoare logic augmented with state dependent "modifies" specifications 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 SMT solvers. This paper uses region logic to specify and verify a representative implementation of the Composite design pattern. To evaluate efficacy of the specification, it is used in verifications of several sample client programs including one with hiding. Verification is performed using a verifier for region logic built on top of an existing verification condition generator which serves as a front end to an SMT solver.

1 Introduction

The Composite pattern [7] captures a frequently encountered 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 an encapsulated representation for an abstract set, this pattern exposes an interface that allows clients to directly access every node. The pattern was featured in a recent survey of challenges for reasoning about sequential object-oriented programs [11] and was the challenge problem of a workshop [18]. In this paper we present a novel solution aimed at current verification tools: indeed we machine-check the verification of the pattern and some sample clients using the Z3 SMT solver [6] via its Boogie 2 [14] 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 [11, 18] is an illustrative example. There is an operation, $getTotal$, that returns the number of descendants of a given node, counting the node itself. Method

⋆ Partially supported by US NSF awards CNS-0627338, CRI-0708330.
⋆⋆ Partially supported by US NSF award CNS-0627448, CM Project S2009TIC-1465 Prometidos, MICINN Project TIN2009-14599-C03-02 Desafios, EU IST FET Project 231620 Hats.
⋆⋆⋆ Partially supported by US NSF awards CNS-0627338, CRI-0708330, CCF-0915611.

$getTotal$ 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 cache the result by declaring in $Component$ an integer field, $total$, and to maintain the invariant that each node's $total$ is the number of all 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. Consider the parameterized predicate $ok(o)$, defined at the top of Fig. 1, which says that the $total$ at o is one plus the sum of $total$ of all the children of o. This has the attractive feature of being "local" to node o and its children. Moreover, if every[3] o of type $Composite$ satisfies $ok(o)$ then each $o.total$ is in fact the number of descendants.

The beauty of this formulation (stipulated in [11]) is that it does not involve recursion, which makes it more amenable to automated first-order reasoning. The notion of sequence sum, however, is inherently recursive. In other works this is avoided by treating composites as having exactly two children, as it is not the central issue of the challenge. Our work, however, alleviates reasoning about sequence sum by appealing to "local reasoning".

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. But 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 be responsible for maintaining it? We shall answer these questions without using specialized invariant disciplines or higher-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 [19]. A discipline is imposed to ensure that the object invariant depends only on the reps and that clients cannot update the reps directly, so clients cannot falsify a candidate invariant. Thus the invariant may be *hidden* [8] in the sense that it is not mentioned in the public specification of a method like add. (While verifying the implementation of add, the invariant is assumed as the precondition and asserted as postcondition.) 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 [7], that do not fit

[3] Quantification in region logic is bounded by a region expression.

well with ownership due to back and forth dependencies and reentrant callbacks. Proposed extensions of ownership that support hiding of invariants in these patterns [16, 21] seem ad hoc. The difficulties led some researchers to abandon the traditional notion of hiding encapsulated invariants [8] in favor of making them explicit in contracts, abstracted in some way for information hiding [5]. We address the first posed question by using explicit invariants—each visible method's pre- and postconditions are conjoined with all the invariants; e.g., specification of add requires and ensures that the $total$ is correct for each (allocated) node.

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 [10] 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[4]—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]. By specifying expressive (write) effects, we address the second question; i.e., effect specification of add allows us to conclude that the $total$ field of any node other than an ancestor was not written.

In previous work [1], 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. 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. 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. As witnessed in our previous work [2], a judicious use of regions in effect specifications facilitates local reasoning (c.f. [22]) and its automation, which served as impetus for this work. We also find that regions support information hiding, without recourse to induction, higher-order logic, or method calls in specifications [26]. The solution to the last posed question is sketched in Sect. 5 where we show how to hide a conjunct from a client's view of the specification of add.

This paper. Using a simplified version of our specification of the Composite pattern, Sect. 2 reviews the basics of region logic. It also introduces our approach of using explicitly quantified invariants as opposed to hiding quantification via a built-in notion of object invariant. In Sect. 3 we discuss automated verification in our approach, i.e., using an automated prover for code annotated with loop invariants and other assertions.

[4] We are considering a Java-like state model, so the mutable locations are pairs (o, f) with o an object reference and f a field name.

A number of publications on reasoning about design patterns [4, 9, 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 the specifications of Sect. 2 and shows their use in reasoning about nontrivial clients that manipulate several composites. Information hiding is sketched in Sect. 5. Sect. 6 discusses our experience and related work.

Contribution. We have successfully applied region logic to specify and verify the original challenge problem, and beyond, including well-chosen clients, which hints at the usability of our specification. We have implemented a verifier for region logic, VERL, built on top of the Boogie 2, Z3 tool chain. All of our code has been mechanically verified. It is available together with the verifier in [23].

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.[5]

Implementation. Fig. 1 depicts a simple implementation (including all annotations) 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 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. We use a mathematical sequence for simplicity, to avoid the distraction of heap-allocated arrays. 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 refine them to provide more precise write effects suitable for clients and to illustrate hiding of some of the invariants.

Specification of add. Public method add inserts an existing composite into the children of **self** and then invokes private method $addToTotal$ which repairs 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*, i.e., state which satisfies the preconditions) may be written. We list write effects following keyword **wr**. A *region* is a set of references; region expressions, G, have type **rgn** and can occur in assertions and in effects. The region expression $\varnothing$ denotes the empty region, whereas $\{E\}$ (singleton region) denotes a singleton set containing the value, possibly null, denoted by expression E. Region expressions of the form $G`f$ (read "$G$'s image under $f$") when used for their r-values are restricted to fields $f$ of reference type or of type **rgn**. If $f$ is a field of reference type then the r-value of $G`f$ is

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

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

$I0$: $\quad \forall o : Comp \mid o.total \ge 1$
$I1(r : \textbf{rgn})$: $\quad \forall o : Comp \in \textbf{alloc} - r \mid ok(o)$
$I2$: $\quad \forall p : Comp, o : Comp \mid o \in p.children \Leftrightarrow o.parent = p$
$I3$: $\quad \forall o : Comp, i : \textbf{int}, j : \textbf{int} \mid 0 \le i < j < len(o.children) \Rightarrow o.children[i] \neq o.children[j]$
$I(r : \textbf{rgn})$: $\quad I0 \wedge I1(r) \wedge I2 \wedge I3$ and $I \mathrel{\widehat{=}} I(\varnothing)$

```
public class Comp {
  seq<Comp> children; int total ; // initially total = 1 and children is empty sequence
  Comp parent; // initially parent = null

void add(Comp c)
  requires c ≠ null ∧ c.parent = null;
  requires self ≠ c ∧ I;
  ensures c.parent = self ∧ I;
  effects wr {c}`parent, {self}`children;
  effects wr alloc`total;
{
  assert c ∉ self.children;
  preserves I1({self}) {
    c.parent := self;
    self.children := [c] + self.children;
  }
  self.addToTotal(c.total);
}

int getTotal()
  requires I;
  ensures result = self.total ∧ I;
{
  result := self.total;
}

void addToTotal(int t)
  requires t ≥ 1;
  requires self.total + t =
    1 + sum i; 0 ≤ i < len(self.children) |
      self.children[i].total;
  requires I({self});
  ensures I;
  effects wr alloc`total;
{
  Comp p; int prv_total;
  p := self;
  while (p ≠ null)
    inv I({p})
    inv p ≠ null ⇒ p.total + t =
      1 + sum i; 0 ≤ i < len(p.children) |
        p.children[i].total;
  {
    assert p.parent ≠ null ⇒
      p ∈ p.parent.children;
    preserves I1({p} + {p.parent}) {
      prv_total := p.total;
      p.total := prv_total + t;
    }
    assert p ≠ p.parent ⇒ p ∉ p.children;
    p := p.parent;
  }
}
```

Fig. 1. Composite pattern: preliminary specifications and implementation. Complete code and annotations lifted from the VERL input file, `composite.rl`.

the set of v such that $v = o.f$ for some $o \in G$. However, if f : **rgn** then the r-value of $G\text{`}f$ denotes the union of the $f$-images (so we have no sets of sets). In effects, a use of $G\text{`}f$ refers to its l-value, and then f can have any type (c.f. **wr** $\{c\}\text{`}parent$, **wr alloc**`$total$ in Fig. 1).

The assertions $G \subseteq G'$ and $G \# G'$ say, respectively, that region G is a subset of G' and $G \cap G' \subseteq \{\mathsf{null}\}$. In particular, $G\text{`}f \subseteq G$ says that $G$ is closed under $f$ and $G\text{`}f \# G'$ 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. A read effect **rd** $G\text{`}f$ 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,[6] but any smaller range can be specified as bound. Thus in $I2$, both p and o range over **alloc** whereas the o in $I1(r)$ ranges over all objects in the region **alloc** $- r$. Here '$-$' denotes set subtraction. Write $I1$ for $I1(\varnothing)$ and I for $I(\varnothing)$.

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 ($c.parent =$ **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: the $parent$ field of c, the $children$ field of **self**, and the $total$ field of any allocated component. Condition I is intended to be invariant in the sense that it holds in all client-visible states; so it appears as both pre- 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 that p is o's unique parent iff o is p's child; $I3$ says that $children$ does not contain any duplicates. In conjunction with the invariant, I, the specification of add says that c was added to $children$ of **self**: initially, $c.parent =$ **null** and $I2$ together entail $c \notin$ **self**$.children$; finally, $c.parent =$ **self** and $I2$ together entail $c \in$ **self**$.children$. An astute reader will note that the specification is partially correct, but *not* totally correct. The reason is 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 such a case the call to add will diverge. Note, however, that the I-invariants entail[7] acyclicity. The strengthened preconditions in Sect. 4 prevent add from creating any cycle. (They should suffice to show *total* correctness of add.)

Last but not least, Fig. 1 contains the requisite annotations. Aside from the standard ones, i.e., loop invariants and **assert** statements, there appear **preserves** annotations. We shall explain them in Sect. 3 under the rubric of "Localized framing".

Proof system by example. The proof system of region logic features "local rules" empowered by the FRAME rule which we will see soon. The formal details can be gleaned from [1]; here we explain those informally. Let's consider proving a part of the speci-

[6] The semantics is instrumented in that newly allocated objects are automatically added to **alloc**. A command that allocates must report effect **wr alloc**.

[7] The proof has not been mechanized but can be easily shown by induction.

fication needed in the proof of add, in particular, establishing the assertion $I1(\{\textbf{self}\})$ which is required, as a conjunct of $I(\{\textbf{self}\})$, immediately before the invocation of $addToTotal$. From the local specification ("small axiom") of $c.parent := \textbf{self}$ we get $c \neq \textbf{null}$ as the precondition, $c.parent = \textbf{self}$ as the postcondition, and $\textbf{wr}\,\{c\}\text{`}parent$ as the write effect which licenses the update. Observe locality at play: the rule refers only to the immediate state of the assignment at hand: $c, \textbf{self}$ and $\{c\}\text{`}parent$; the write effect specifies only the location which is pertinent to the field update. Intuitively, one can deduce that an assertion that does not "depend" on the write effect must be preserved by the field update: in this case, the truth of $I1$ is unaffected by the update of $c.parent$ because $I1$ does not read the $parent$ field of any object in **alloc**. Consequently, we can conjoin $I1$ to the pre- and postconditions. Then by the standard rule of CONSEQUENCE we can weaken the postcondition to obtain $I1(\{\textbf{self}\})$. For the next command that updates $\textbf{self}.children$, $I1(\{\textbf{self}\})$ holds in the pre-and postcondition because the write effect is $\{\textbf{self}\}\text{`}children$, whereas $I1(\{\textbf{self}\})$ reads the $children$ field of all objects in **alloc** *except* **self**.

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

$$\text{FRAME}\quad \frac{\vdash \{\,P\,\}\ C\ \{\,P'\,\}\,[\overline{\varepsilon}] \qquad P \vdash \overline{\delta}\ \textbf{frm}\ Q \qquad P \Rightarrow \overline{\delta}\ \textit{/.}\ \overline{\varepsilon}}{\vdash \{\,P \wedge Q\,\}\ C\ \{\,P' \wedge Q\,\}\,[\overline{\varepsilon}]}$$

read: Q is preserved by C under precondition P if $\overline{\varepsilon}$, the write effects of C, is separate from $\overline{\delta}$, the read effects of Q. The *frames judgement* $P \vdash \overline{\delta}\ \textbf{frm}\ Q$ asserts $\overline{\delta}$ are at least 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 [1].) The antecedent $P \Rightarrow \overline{\delta}\ \textit{/.}\ \overline{\varepsilon}$ asserts that the precondition may be assumed to prove that the read effects are separate from the write effects. We call $\overline{\delta}\ \textit{/.}\ \overline{\varepsilon}$ a *separator*. The function $\textit{/.}$ computes a conjunction R of disjointness formulas such that in R-states, writes allowed by $\overline{\varepsilon}$ cannot falsify a formula framed by $\overline{\delta}$. (Frames judgements in conjunction with separators formalize the notion of (in)dependence—whether or not an assertion may depend on write effects.)

Above, the read effects of $I1$ are $\textbf{rd}\,\textbf{alloc}, \textbf{alloc}\text{`}children, \textbf{alloc}\text{`}total$, which do not refer to $parent$, hence are separated from $\textbf{wr}\,\{c\}\text{`}parent$; thus $I1$ can be conjoined by FRAME. Read effects of $I1(\{\textbf{self}\})$ are $\textbf{rd}\,\textbf{alloc}, \textbf{self}, (\textbf{alloc} - \{\textbf{self}\})\text{`}children, (\textbf{alloc} - \{\textbf{self}\})\text{`}total$. These are separated from $\textbf{wr}\,\{\textbf{self}\}\text{`}children$ because $\textbf{alloc} - \{\textbf{self}\}$ is disjoint from $\{\textbf{self}\}$. So $I1(\{\textbf{self}\})$ can be conjoined by FRAME.

3 Region logic can Boogie: automated verification

We describe key steps in translating programs specified in region logic to Boogie 2 programs. We also share our experience with the translation and verification as it pertains to the Composite.

VERL. Our VErifier for Region Logic [23] translates a program specified in region logic to a Boogie 2 [14] program. We started with Dafny [13] and adapted its specification language while keeping its programming language mostly the same. Key features

of VERL's specification language include the full generality of region assertions and effects as well as "localized framing"—code blocks annotated with formulas whose truth must be preserved by essentially appealing to FRAME. A distinguishing feature of VERL is an automatic (syntax-directed) computation[8] of read effects of formulas and expressions. For example, only the **sum** expression needed a read effect specification; read effects of all other formulas and invariants were inferred automatically.

Boogie. The Boogie 2 [14] verification platform consists of an intermediate procedural verification language Boogie [14], a verification condition generator (VCGen) and an SMT solver. Given a specified Boogie program, and a list of procedures to verify, VCGen computes the weakest precondition of each specified procedure relative to its implementation and specified loop invariants. These verification conditions (VCs) are handed off to a prover, such as Z3, together with the "background predicate" that axiomatizes the semantics of Boogie and any additional user-defined axioms (which typically encode the semantics of the source language).

A Boogie program may consist of logical declarations and definitions, procedures, as well as specifications thereof. The logical definitions may consist of variables, constants, function symbols and axioms. Procedure implementation can use ordinary assignment, control-flow commands such as **while**, typically annotated with loop invariants, **if**-**then**-**else**, **return** and **goto**, procedure call commands, as well as special (meta) commands: **assume**, **assert**, **havoc**. Procedure specifications consist of pre-/postconditions and write effects (of global variables). Specifications can be 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. Boogie comes equipped with some primitive types: **bool**, **int**, type constructors, as well as map types (corresponding to the theory of arrays). For example, given a type constructor **ref** we can define the map type $\mathbf{rgn} \mathrel{\widehat{=}} \mathbf{ref} \to \mathbf{bool}$ to encode regions as characteristic functions.

Encoding region logic. The encoding of the heap is similar to Dafny, except allocated objects are represented by the **alloc** region. Thus, the heap is essentially a pair consisting of the global variable $Heap$—a map indexed by $(\mathbf{ref}, \mathbf{Field}\,\alpha)$ pairs, where α ranges over any type [17], and the global region variable **alloc**; e.g., $\mathbf{in}(o, \mathbf{alloc})$ says that o is allocated, where $\mathbf{in} : \mathbf{ref} \times \mathbf{rgn} \to \mathbf{bool}$.

The translation of region assertions and hence pre- and postconditions is straightforward. To translate write effects, including for example $\mathbf{wr}\, G \text{`} f$, we conjoin the following postcondition in Boogie:

$$\forall \langle \alpha \rangle\, o : \mathbf{ref}, g : \mathbf{Field}\,\alpha \mid \mathbf{in}(o, \mathbf{old}(\mathbf{alloc})) \Rightarrow \\ Heap[o,g] = H[o,g] \vee (\mathbf{in}(o, [\![G]\!]_H) \wedge g = f)$$

where $H \mathrel{\widehat{=}} \mathbf{old}(Heap)$, and $[\![\cdot]\!]_H$ is a translation function from VERL to Boogie, parameterized by the heap variable; e.g., $[\![x.f]\!]_H \mathrel{\widehat{=}} H[x,f]$. 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 additional write effects are specified. When **wr alloc** is not specified in the

[8] Derived from frames judgements formalized in our earlier work [1].

write effects, we also conjoin the postcondition $\mathbf{old}(\mathbf{alloc}) = \mathbf{alloc}$ that asserts absence of allocation.

Localized framing. Local reasoning can aid the prover in two ways: firstly, by avoiding direct reasoning about complex formulas, and secondly by reducing the number of case splits performed when reasoning about heap updates. VERL supports code blocks annotated with **preserves** clauses as already witnessed in Fig. 1. For example, the **preserves** annotation in add instructs VERL to conjoin $I1(\{\mathbf{self}\})$ by essentially instantiating FRAME. In detail, **preserves** $P\ \{C\}$ is encoded as

$$H := Heap; [\![C]\!]_{Heap};\ \mathbf{assert}\ H, Heap \text{ agree on } \overline{\varepsilon};\ \mathbf{assume}\ [\![P]\!]_H = [\![P]\!]_{Heap};$$

where $\vdash \overline{\varepsilon}\ \mathbf{frm}\ P$ has been established, e.g., by a syntax-directed analysis. Prior to executing C we snapshot the heap into H. The assert statement ensures the heaps, before and after the execution of C, agree on the read effects $\overline{\varepsilon}$—roughly, for every o which was allocated before the execution of C, and for every $\mathbf{rd}\, G\text{`}f$ in $\overline{\varepsilon}$, if $\mathbf{in}(o, [\![G]\!]_H)$, then $H[o,f] = Heap[o,f]$—thus we can assume P is preserved by C. The soundness of the above is a direct consequence of the frame agreement lemma [1, Lemma 4] that underlies soundness of the FRAME rule. Therefore, a **preserves** annotation establishes preservation of arbitrary formulas over the enclosed updates by merely appealing to the formulas' read effects as opposed to using the formulas' actual meaning. We call this "localized framing" to contrast with "framing axioms" [26, 13].

Our experience. The sequence sum axiomatization draws on the axioms of Leino and Monahan [15]. We needed an additional axiom to express that the **sum** distributes over catenation. Our earlier verification efforts relied on framing axioms of [13] for all invariants. However, the prover exhibited difficulty (manifested by timeouts) in reasoning about the preservation of formulas containing **sum**. By switching to localized framing we were able to avoid timeouts and remove a significant number of **assert** annotations needed to guide the prover. By default, VERL does not generate framing axioms. However, a declaration of a function can be tagged to override the default. We used this feature to generate a single framing axiom for the function which encodes sequence sum. While the **preserves** annotations deal with the framing of $I1$, the generated framing axiom is used to reason about the preservation of **sum** expressions.

4 Refining specifications: smaller footprints for client reasoning

The specifications in Fig. 1 are weak: they permit cycle creation (c.f. Sect. 2) and the effect $\mathbf{wr}\,\mathbf{alloc}\text{`}total$ is too imprecise for some client reasoning (as we see soon). This section refines the specification of add.

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 $\{\,P\,\}\ a.add(b)\ \{\,P'\,\}\,[\overline{\varepsilon}]$, where $P \mathrel{\widehat{=}} b \neq \mathbf{null} \wedge b \neq a \wedge b.parent = \mathbf{null}$, $P' \mathrel{\widehat{=}} b.parent = a$, $\overline{\varepsilon} \mathrel{\widehat{=}} \mathbf{wr}\{b\}\text{`}parent, \{a\}\text{`}children, \mathbf{alloc}\text{`}total$, and we elide the invariant $I$. A client could appeal to FRAME to show, e.g., $a.parent = x$ is preserved: the obligation is to show $P \Rightarrow (\mathbf{rd}\,\{a\}\text{`}parent, x) \mathbin{/\!\!.} \overline{\varepsilon}$ which amounts to

$P \Rightarrow \{a\}\#\{b\}$. The disjointness evaluates to **true** using P. On the other hand, reasoning about $total$ will not work because the effect, **wr alloc‘**$total$, is too coarse. In detail, if the assertion to be preserved is $b.total = t$, then FRAME requires establishing the disjointness $\{b\}\,\#\,$**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 $c0, \ldots, c4$ and the client code in Fig. 2. Here the composite tree

```
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
```

Fig. 2. Client reasoning about the preservation of $total$ across calls to add.

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 $c1, \ldots, c4$ must be roots (i.e., their parents are **null**). Furthermore, the effect specification of add must pin down the region whose $total$ field is permitted to be written so that the client can deduce that $c2$ is not in this region. Consequently we need revised specifications for add and $addToTotal$ — see Fig. 3. The figure also contains the definition of $ancestors$ (in terms of $descendants$), and the supporting invariants J, K that capture sufficient "structural" information.

Specifications of add. We add ghost field $desc$: **rgn** to keep track of the set of descendants of a node. We also add ghost field $root : Comp$ to point to the root of a composite tree. Note, by maintaining descendants and roots, we can express a common idiom: components with distinct roots have disjoint descendants. The set of ancestors is defined in terms of descendants by a means of a comprehension expression. (This saves us a ghost field declaration and corresponding updates.) Invariants $J0, J1$ constrain $desc$ to be a reflexive, transitive relation; $J2$ states that $root$ is always non-null, and that descendants of $o.root$ include those of o; $J3$ states that components with distinct roots have disjoint descendants; $J4, J5$ constrain every $parent$ path to have the same root; $J6$ says that for any o which is a *proper* descendant of p, $o.parent.desc$ must be included in $p.desc$, whence by $J1$, $o.parent \in p.desc$. So $J6$ helps pin down that $desc$ contains only reachable components.

The above ghost fields and invariants were derived out of necessity to strengthen the specification of add. We are currently unaware of any general technique to derive the "right" set of essential annotations. However, we have some evidence to believe that the chosen invariants may be helpful in reasoning about other tree-like structures. For example, we can 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.

The postcondition of add is the same as before but with $J \wedge K$ conjoined. However, note that $c \in$ **self**$.root.desc$ is entailed by the postcondition. (From $c.parent =$ **self** and

requires $c \neq \textbf{null} \wedge c.parent = \textbf{null} \wedge c.root \neq \textbf{self}.root \wedge I \wedge J \wedge K$
ensures $c.parent = \textbf{self} \wedge I \wedge J \wedge K$
effects $\textbf{wr}\, \{c\}\text{`}parent,\ ancestors(\textbf{self})\text{`}(total, desc),\ c.desc\text{`}root,\ \{\textbf{self}\}\text{`}children$

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

$J0\colon \quad \forall o : Comp \mid o.desc\text{`}desc \subseteq o.desc$
$J1\colon \quad \forall o : Comp \mid o \in o.desc$
$J2\colon \quad \forall o : Comp \mid o.root \neq \textbf{null} \wedge o.desc \subseteq o.root.desc$
$J3\colon \quad \forall o : Comp, p : Comp, q : Comp \mid o \in p.desc \wedge o \in q.desc \Rightarrow p.root = q.root$
$J4\colon \quad \forall o : Comp \mid o.parent = \textbf{null} \Rightarrow o.root = o$
$J5\colon \quad \forall o : Comp \mid o.parent \neq \textbf{null} \Rightarrow o.root = o.parent.root \wedge o \in o.parent.desc$
$J6\colon \quad \forall o, p : Comp \mid o \in p.desc \wedge o \neq p \Rightarrow o.parent \neq \textbf{null} \wedge o.parent.desc \subseteq p.desc$
$I\colon \quad I0 \wedge I1 \wedge I2 \wedge I3$ (as in Fig. 1)
$J\colon \quad J0 \wedge J1 \wedge J2 \wedge J3 \wedge J4 \wedge J5 \wedge J6$
$K\colon \quad \forall o : Comp \mid \forall i : \textbf{int} \mid 0 \leq i < len(o.children) \Rightarrow o.children[i] \in o.desc$

Fig. 3. Strengthened specification of add, definition of $ancestors$ and invariants. In $J0$, $o.desc\text{`}desc$ is a region expression whose r-value is the union of all $p.desc$ where p ranges over elements of the region $o.desc$.

$J5$, we obtain $c \in \textbf{self}.desc$; $J2$ finishes the proof.) Finally, the most precise write effect for field $total$ is $\textbf{wr}\, ancestors(\textbf{self})\text{`}total$. It says that add may modify $total$ of every ancestor of **self** (including **self**). Observe how cycles are precluded by the precondition $c.root \neq \textbf{self}.root$, which, together with $J3$ entails that c's descendants are disjoint from **self**'s descendants.

Client verification. We have mechanically verified the client in Fig. 2 using add's specification in Fig. 3. Note, the client code needs no annotations; Z3 proves the preservation of $c2$'s $total$ automatically. For a lack of space, we do not sketch a decutive proof but note the key insight: $\textbf{wr}\, c.desc\text{`}root$ helps establish the requisite root disjointedness after each $add$ which in turn with $\textbf{wr}\, ancestors(c)\text{`}total$ establishes that $c2.total$ was not written.

Implementation of add. We require two changes to Fig. 1: subsequent to the addition of c to $\textbf{self}.children$, we perform two bulk updates[9] of ghost fields $desc$ and $root$. The $desc$ field of all objects in $ancestors(\textbf{self})$ is updated to contain c, and the $root$ field of all objects in $c.desc$ is updated to point to $\textbf{self}.root$. See `composite.rl` in distribution; methods `add_simple, addToTotal_simple` correspond to Fig. 1 while `add, addToTotal` correspond to the strengthened version, i.e., this section.

5 Information hiding

One dimension of the Composite challenge problem that we explore is information hiding. We argue that representation invariants—of which $I1$ is an example—should be

[9] Specification statements as embodied in Dafny and more generally in refinement calculus.

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 post-condition, but the invariant does not appear in the contract used to reason about clients [8]. This mismatch is justified as follows: the invariant is supposed to depend only on the state that is encapsulated, and clients cannot write to that part of the state.

As a more general technique for hiding of internal invariants, we propose [20] that a module can declare a *dynamic 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 boundary: it is subject to the proof obligation that it does not write within the dynamic boundary. In other words, intermediate steps in client code execution are required to respect the boundary, so that the write effects of the client are separate from the boundary. This notion can be captured by a second order rule of framing, as exemplified and formalized in [20].

In the sequel, we consider the clients from Sect. 4. Let us consider the invariants I and K. These can be framed by the effect **rd alloc‘**$(desc, parent, children, root, total)$. For this to be a dynamic boundary, we require that clients never write any of these fields. In general, enforcement of a dynamic 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 be read or written in client code but are allowed in specifications visible to clients. Because it is impossible for the clients to write within the boundary, it is sound to hide I and K, i.e., omit them from the specifications with respect to which the clients are verified.

Invariant J is framed by **rd alloc‘**$(desc, parent, root)$ and again for reasons of scope the clients respect the boundary **rd alloc‘**$(desc, parent, children, root, total)$. 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 [12]. 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. Boogie does not include this feature and instead of complicating our translation, we found it suffices to include J as explicit pre- and post-condition in the public specification of add. (In the distribution, the version with hidden invariants is in files `composite.rl, client.rl`; look for methods `addHidden`, `client_hiding`, resp.)

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, as witnessed by the prevalence of universal quantifiers (often nested), ostensibly contradicts notions such as object-centric invariants, locality, or adherence to a particular programming methodology (see, e.g., the Composite verification in [27]). However, as we demonstrated, our approach is to use local reasoning in order to establish global invariants. In many cases, when updates are "shallow", the prover can automatically find the right instantiations without going astray. In more difficult cases, typically involving definitions inductive in flavor, we appeal to the user to add **preserves** annotations. Relying on such annotations is not all that different from relying on loop invariants; the user usually has some intuition about *what* invariants and *where* in the code. Arguably, we still achieve a high degree of automation in exchange for a reasonable request of user guidance.

Related work. We draw heavily on Kassios' [10] 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 [22]. 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. 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", acc, with a permission-based semantics and special program constructs. Every read/write of an expression $E.f$ is permitted by asserting $acc(E.f)$.

The most closely related works directly address the Composite challenge. Bierhoff and Aldrich [4] achieve fully automated checking of the add implementation using typestates (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 typestates 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 [9] present a specification of the Composite using separation logic with a number of inductive definitions, e.g., instead of the non-inductive $I1$ 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. Abstract 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 [11] 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] propose a methodology for (a) specifying object invariant semantics, i.e., which invariant(s) must hold and at what (program) location; (b) verifying preservation of invariants by computing an upper approximation on the set of objects for which an invariant may get invalidated and asserting the invariant holds for this set, thereby establishing that the invariant holds for all objects. The methodology is applied to the Composite problem by specifying and verifying an implementation of add which is nearly identical, (but weaker, e.g., no effects are specified and postcondition "forgets" that c was added) to our preliminary specification depicted in Fig. 1.

We expect that in future other automatic verification tools will address the Composite challenge as well. Rustan Leino has recently informed us of his specification and implementation of the Composite in Dafny (personal communication, May 2010).

Future work. While Sect. 5 shows how invariants I and K may be hidden, the full handling of abstraction is outside the scope of this paper. For that, one would need to verify that representations of internal heap-based data structures are such that client reasoning is unaffected: to wit, whether $children$ is stored in an array or a list instead of a sequence, should not affect the behavior of add on client observable objects.

Automatically inferred **preserves** clauses could potentially relieve a number of required user annotations. A simple static analysis which computes the write effects of a command can be used to infer locations in code where relevant assertions must be preserved owing to separation (of reads from writes).

VERL currently uses quantified axioms to encode region assertions. Such an encoding does not constitute a decision procedure, yet we conjecture that an integrated decision procedure would improve reasoning about regions. A decision procedure for quantifier-free region assertions has been sketched in the first author's thesis proposal and will be implemented in an SMT solver.

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. In our view, the specifications of the methods are fairly succinct and transparent. Their verification, and the verification of interesting client code, relies on a number of global invariants that capture inductive properties in non-inductive ways.

Acknowledgements. We thank Mike Barnett, Sophia Drossopoulou, Rustan Leino, Peter Müller, Shaz Qadeer, Jan Smans, and Alex Summers for helpful discussions. We thank the referees for their careful reading of the manuscript and for numerous suggestions on improving the presentation.

References

1. A. Banerjee, D. A. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.
2. M. Barnett, A. Banerjee, and D. A. Naumann. Boogie meets regions: a verification experience report. In *VSTTE*, pages 177–191, 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 [18].
5. G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, 2005.
6. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
9. B. Jacobs, J. Smans, and F. Piessens. Verifying the composite pattern using separation logic. In [18].
10. I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, volume 4085 of *LNCS*, pages 268–283, 2006.
11. 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.
12. G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, 2007.
13. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
14. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2010.
15. K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT solvers. In *SAC*, 2009.
16. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
17. K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, 2010.
18. Robby et al. *Proc. Seventh SAVCBS Workshop*. Technical Report CS-TR-08-07, School of Electrical Engineering and Computer Science, University of Central Florida, 2008.
19. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Programming*, 62(3):253–286, 2006.
20. D. A. Naumann and A. Banerjee. Dynamic boundaries: Information hiding by second order framing with first order assertions. In *ESOP*, 2010.
21. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, 365:143–168, 2006.
22. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
23. S. Rosenberg, A. Banerjee, and D. A. Naumann. Verifier for region logic (VERL). `http://www.cs.stevens.edu/~naumann/pub/VERL/`.
24. S. M. Shaner, H. Rajan, and G. T. Leavens. Model programs for preserving composite invariants. In [18].
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. In *VMCAI*, 2010.