

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 every¹ *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

¹ Quantification in region logic is bounded by a region expression.