Verifying the Composite Pattern using Separation Logic

Bart Jacobs * Jan Smans † Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium

{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract

Often, a module developer wishes to expose a graph of objects to client code, allowing client code to access the graph through any node directly, while maintaining hidden consistency conditions over the graph. In this note, we describe how to specify and verify such code using separation logic, using as an example a binary tree structure where each node keeps a count of its descendant nodes. The idea is to describe the tree structure as the separate conjunction of *the focus node's subtree* and *the focus node's context*. The description can be rewritten to use any other node as the focus node at any time. This enables an elegant modular proof of the tree implementation on the one hand, and client code on the other hand.

We describe how we verified an example program using the VeriFast program verifier prototype.

1. Introduction

Classical ownership systems force all accesses of a composite object structure to go through a designated root object. This makes it difficult to verify programs where a module exposes a graph of objects to client code, and client code accesses the component objects directly. In this paper we show an approach for verifying such programs using separation logic (Reynolds 2002). The general idea is that before accessing a node, the separation logic assertion that describes the structure is rewritten so that the target node is at the "top level" of the recursive description and can easily be separated out. For information hiding, a single abstract predicate (Parkinson and Bierman 2005) is used to represent the entire structure (Parkinson 2007).

2. Example

We illustrate the approach by showing a full functional correctness specification, an implementation, and client code for a tree structure module written in C. The implementation keeps a count in each node of the node's descendant nodes. The code is annotated so that it is verified automatically with our VeriFast tool (Jacobs and Piessens 2008), a program verifier based on symbolic execution with a separation logic representation of memory.

2.1 Specification and client

The specification of the tree module is shown in Figure 1. The module's functions, $create\_tree$, $tree\_add\_left$, $tree\_add\_right$, $tree\_get\_count$, $tree\_get\_parent$, and $tree\_dispose$, are specified in terms of the abstract predicate $tree(n, c, t)$, where n is the *focus node*, c is the focus node's *context*, and t is the focus node's *subtree*. The context and the subtree are values of the inductive datatypes $context$ and $tree$, respectively. The context is the part of the tree obtained by removing the focus node and its subtree; it is either $root$, indicating that the focus node is the root node of the tree; or $left\_context(pc, p, r)$, indicating that the focus node is the left child of a node p, whose own context is pc and whose right child's subtree is r; or $right\_context(pc, p, l)$, analogously indicating that the focus node is the right child of a node p, whose own context is pc and whose left child's subtree is l. The subtree is either nil, indicating that the focus node is a null pointer; or $tree(n, l, r)$, indicating that the focus node is n and its left and right child's subtrees are l and r, respectively.

The specification of function $tree\_get\_count$ uses the pure primitive recursive function $count$ over $tree$ values.

Predicate assertion arguments may be *patterns*, of the form $?x$, which bind x in the remainder of the symbolic execution. They are in effect existentially quantified logical variables.

Throughout, the star operator ($*$) indicates separate conjunction, and binds like conjunction, i.e. more weakly than relational operators.

Function $tree\_add\_left$ not only creates a new node and adds it as the left child of the specified node; it also makes the new node the new focus node. Similarly, $tree\_add\_right$ and $tree\_get\_parent$ make the new right child node and the parent node the focus node, respectively. As a result, the example client program in Figure 2 verifies as is.

2.2 Implementation and proof

The implementation of the tree module is shown in Figures 3 – 7. Figure 3 shows how predicate $tree$ is defined in terms of predicates $context$ and $subtree$. A predicate assertion $subtree(n, p, t)$ states that the heap contains a representation of a subtree t with root node n and parent pointer p, with consistent $count$ fields. An assertion $context(n, p, count, c)$ states that the heap contains a representation of a context c of node n with parent p, where the $count$ fields are consistent under the assumption that the count of the subtree at n is $count$. The assertion $malloc\_block\_node(n)$ indicates that pointer n points to a **struct** $node$ object allocated using $malloc$.

Figure 4 shows the implementations of $create\_node$ and $create\_tree$. A **close** $p(\overline{v})$; ghost statement consumes the body of predicate p, with arguments $\overline{v}$ substituted for the parameters, removing from the symbolic heap the points-to assertions and abstract predicate assertions mentioned in the body, and then produces, i.e. adds to the symbolic heap, the abstract predicate assertion $p(\overline{v})$ itself.

Figures 5 and 6 show the implementation of functions $tree\_add\_left$ and $tree\_get\_count$, and auxiliary functions $subtree\_get\_count$ and $fixup\_ancestors$. Function $subtree\_get\_count$ relies on the count invariant expressed in the $subtree$ predicate so that it can simply return the value of the $count$ field. Function $tree\_add\_left$ first creates the new node, and then calls $fixup\_ancestors$, which recursively ascends the tree to adjust the $count$ fields of the new node's ancestors. The effect of the $fixup\_ancestors$ calls is framed

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

† Jan Smans is a Research Assistant of the Research Foundation - Flanders (FWO)

```
struct node;
typedef struct node *node;

inductive tree := nil | tree(node, tree, tree);

fixpoint int count(tree t) {
  switch (t) {
    case nil : return 0;
    case tree(n, l, r) : return 1 + count(l) + count(r);
  }
}

inductive context :=
  | root
  | left_context(context, node, tree)
  | right_context(context, node, tree);

predicate tree(node node, context c, tree subtree);

node create_tree();
  requires emp;
  ensures tree(result, root, tree(result, nil, nil));

node tree_add_left(node node);
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };

node tree_add_right(node node);
  ... analogous ...

int tree_get_count(node node);
  requires tree(node, ?c, ?t);
  ensures tree(node, c, t) * result = count(t);

node tree_get_parent(node node);
  requires tree(node, ?c, ?t) * c ≠ root * t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p * tree(p, pns, tree(p, t, r));
      case right_context(pns, p, l) :
        result = p * tree(p, pns, tree(p, l, t));
    };

void tree_dispose(node node);
  requires tree(node, root, _);
  ensures emp;
```

Figure 1. Specification of the tree module. Annotations are shown on a gray background.

```
int main()
  requires emp;
  ensures emp;
{
  // {}
  node node := create_tree();
  // {tree(n0, root, tree(n0, nil, nil))}
  node := tree_add_left(node);
  // {tree(n1, left_context(root, n0, nil), tree(n1, nil, nil))}
  node := tree_add_right(node);
  // {tree(n2, right_context(left_context(root, n0, nil), n1,
  //     nil), tree(n2, nil, nil))}
  node := tree_get_parent(node);
  // {tree(n1, left_context(root, n0, nil),
  //     tree(n1, nil, tree(n2, nil, nil)))}
  node := tree_add_left(node);
  // {tree(n3, left_context(left_context(root, n0, nil), n1,
  //     tree(n2, nil, nil)), tree(n3, nil, nil))}
  node := tree_get_parent(node);
  // {tree(n1, left_context(root, n0, nil), tree(n1,
  //     tree(n3, nil, nil), tree(n2, nil, nil)))}
  node := tree_get_parent(node);
  // {tree(n0, root, tree(n0, tree(n1, tree(n3,
  //     nil, nil), tree(n2, nil, nil)), nil))}
  tree_dispose(node);
  // {}
  return 0;
}
```

Figure 2. An example client program. Symbolic states are shown in blue.

by the fact that this function requires access only to the current node's context, not to the entire tree. A *fixup_ancestors* call takes a context that is consistent with respect to some unknown count, and returns the same context, made consistent with respect to the given count.

Ghost statement **open** $p(\overline{pat})$; is the reverse of **close**: it consumes the abstract predicate assertion $p(\overline{pat})$ and then produces the predicate's body. Contrary to the **close** statement, the arguments in an **open** statement can be patterns; the scope of the variables bound by these patterns extends to the end of the function body.

Function *tree_add_left* contains an **assert** statement; its only purpose here is to bind variable r to the *tree* value that represents the subtree below node *nodeRight*.

Figure 7 shows the implementation of functions *tree_get_parent* and *tree_dispose*, and auxiliary function *subtree_dispose*. As pointed out before, function *tree_get_parent* not only returns the parent node pointer of the specified node, but also makes the parent node the focus node of the description of the tree structure. The implementation effectively moves the specified node's parent's fields, and the specified node's sibling subtree, from the context part of the tree description into the subtree part.

2.3 Non-contiguous focus changes

The tree module specification, as shown in Figure 1, requires client code to navigate the tree contiguously; i.e., to access a given node, the client must navigate to this node using *tree_get_parent*, *tree_get_left*, and *tree_get_right* calls (the latter functions are not shown). In this subsection, we show how the tree module's specification can be extended so that clients can change focus not only to adjacent nodes but to any node in the tree.

```
struct node {
  struct node *left;
  struct node *right;
  struct node *parent;
  int count;
};

predicate subtree(node node, node parent, tree t) =
  switch (t) {
    case nil : node = 0;
    case tree(node0, leftNodes, rightNodes) :
      node = node0 * node ≠ 0 *
      node→left ↦ ?left *
      node→right ↦ ?right *
      node→parent ↦ parent *
      node→count ↦ count(t) *
      malloc_block_node(node) *
      subtree(left, node, leftNodes) *
      subtree(right, node, rightNodes);
  };

predicate context(node n, node p, int count, context c) =
  switch (c) {
    case root : p = 0;
    case left_context(pns, p0, r) :
      p = p0 * p ≠ 0 *
      p→left ↦ n *
      p→right ↦ ?right *
      p→parent ↦ ?gp *
      p→count ↦ ?pcount *
      malloc_block_node(p) *
      context(p, gp, pcount, pns) *
      subtree(right, p, r) *
      pcount = 1 + count + count(r);
    case right_context(pns, p0, l) :
      ... analogous ...
  };

predicate tree(node node, context c, tree subtree) =
  context(node, ?parent, count(subtree), c) *
  subtree(node, parent, subtree);
```

Figure 3. Struct and predicate declarations

The additional specification elements are shown in Figure 8. The main new element is the specification of the *change_focus lemma function*. A lemma function, or *lemma* for short, is like a regular C function, but it is declared in an annotation; it is not allowed to have any side-effects; and the verifier checks that it terminates. Unlike a regular function call, a lemma function call is a ghost statement. The sole purpose and effect of calling a lemma function is to rewrite the symbolic state into a different, but equivalent, form. Lemma *change_focus* takes a tree structure description with focus node $n0$, and a node pointer n that is contained in the tree structure at a path p, and rewrites the tree structure description so that the focus node is n. The new *context* and *tree* values are specified using the fixpoint functions *combine*, *context_at*, and *subtree_at*.

Figure 9 shows an example of a piece of client code whose verification is enabled by lemma *change_focus*.

Figure 10 shows the implementation, i.e. the proof, of lemma *change_focus*. The proof uses two auxiliary lemmas, *go_to_root* and *go_to_descendant*. Lemma *go_to_root* operates by recursion

```
node create_node(node p)
  requires emp;
  ensures subtree(result, p, tree(result, nil, nil));
{
  node n := malloc(sizeof(struct node));
  n→left := 0; close subtree(0, n, nil);
  n→right := 0; close subtree(0, n, nil);
  n→parent := p;
  n→count := 1;
  close subtree(n, p, tree(n, nil, nil));
  return n;
}

node create_tree()
  requires emp;
  ensures tree(result, root, tree(result, nil, nil));
{
  node n := create_node(0);
  close context(n, 0, 1, root);
  close tree(n, root, tree(n, nil, nil));
  return n;
}
```

Figure 4. Implementation of function *create_tree*

(i.e., induction) on the structure of the *context* value; similarly, lemma *go_to_descendant* operates by recursion (induction) on the structure of the *path* value.

VeriFast checks that each lemma function terminates, by disallowing loops and indirect recursive calls, and by checking that at each direct recursive call, either the callee's footprint is a strict subset of the caller's footprint, or the lemma function's body is a switch statement on one of its parameters, and the value of this parameter for the callee is a component of the value of this parameter for the caller. The footprint of a call is the footprint of the precondition; the footprint of an assertion is the set of memory locations that are present in each heap that satisfies the assertion. VeriFast checks the footprint requirement for a given recursive call by checking that after consuming the callee's precondition, at least one points-to assertion is left in the symbolic state.

3. Conclusion

We propose a way to specify and verify modules that expose a graph of objects, while maintaining hidden invariants over this graph and allowing clients to access the graph at any node directly. In this approach, the graph structure is described by a separation logic assertion. Before accessing a node, the assertion is rewritten to separate out this node. We illustrated the approach by showing a specification and an implementation of a binary tree module, annotated to enable verification using our VeriFast program verifier.

VeriFast is available at

http://www.cs.kuleuven.be/~bartj/verifast/

References

Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report 520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

```
int subtree_get_count(node node)
  requires subtree(node, ?parent, ?nodes);
  ensures subtree(node, parent, nodes) *
    result = count(nodes);
{
  int result := 0;
  open subtree(node, parent, nodes);
  if (node ≠ 0) { result := node→count; }
  close subtree(node, parent, nodes);
  return result;
}

void fixup_ancestors(node n, node p, int count)
  requires context(n, p, _, ?c);
  ensures context(n, p, count, c);
{
  open context(n, p, _, c);
  if (p ≠ 0) {
    node left := p→left;
    node right := p→right;
    node grandparent := p→parent;
    int leftCount := 0;
    int rightCount := 0;
    if (n = left) {
      leftCount := count;
      rightCount := subtree_get_count(right);
    } else {
      leftCount := subtree_get_count(left);
      rightCount := count;
    }
    {
      int pcount := 1 + leftCount + rightCount;
      p→count := pcount;
      fixup_ancestors(p, grandparent, pcount);
    }
  }
  close context(n, p, count, c);
}
```

Figure 5. Helper functions for function *tree_add_left*

Matthew Parkinson. Class invariants: the end of the road? In *IWACO 2007*, 2007.

Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL 2005*, 2005.

J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS 2002*, 2002.


```
node tree_add_left(node node)
  requires
    tree(node, ?c, ?t) *
    switch (t) {
      case nil : false;
      case tree(n0, l, r) : l = nil;
    };
  ensures
    switch (t) {
      case nil : false;
      case tree(n0, l, r) :
        tree(result, left_context(c, node, r),
          tree(result, nil, nil));
    };
{
  open tree(node, c, t);
  node n := create_node(node);
  open subtree(node, ?parent, t);
  node nodeRight := node→right;
  assert subtree(nodeRight, node, ?r);
  {
    node nodeLeft := node→left;
    open subtree(nodeLeft, node, nil);
    node→left := n;
    close context(n, node, 0, left_context(c, node, r));
    fixup_ancestors(n, node, 1);
  }
  close tree(n, left_context(c, node, r), tree(n, nil, nil));
  return n;
}

node tree_add_right(node node)
  ... analogous ...

int tree_get_count(node n)
  requires tree(n, ?c, ?t);
  ensures tree(n, c, t) * result = count(t);
{
  open tree(n, c, t);
  int result := subtree_get_count(n);
  close tree(n, c, t);
  return result;
}
```

Figure 6. Implementation of function *tree_add_left*

```
node tree_get_parent(node node)
  requires tree(node, ?c, ?t) ∗ c ≠ root ∗ t ≠ nil;
  ensures
    switch (c) {
      case root : false;
      case left_context(pns, p, r) :
        result = p ∗ tree(p, pns, tree(p, t, r));
      case right_context(pns, p, r) :
        result = p ∗ tree(p, pns, tree(p, l, t));
    };
{
  open tree(node, c, t);
  open subtree(node, _, t);
  node p := node→parent;
  close subtree(node, p, t);
  open context(node, p, count(t), c);
  assert context(p, ?gp, ?pcount, ?pns);
  switch (c) {
    case root :
    case left_context(pns, p0, r) :
      close subtree(p, gp, tree(p, t, r));
    case right_context(pns, p0, l) :
      close subtree(p, gp, tree(p, l, t));
  }
  assert subtree(p, gp, ?pt);
  close tree(p, pns, pt);
  return p;
}

void subtree_dispose(node node)
  requires subtree(node, _, _);
  ensures emp;
{
  open subtree(node, _, _);
  if (node ≠ 0) {
    subtree_dispose(node→left);
    subtree_dispose(node→right);
    free(node);
  }
}

void tree_dispose(node node)
  requires tree(node, root, _);
  ensures emp;
{
  open tree(node, root, _);
  open context(node, _, _, root);
  subtree_dispose(node);
}
```

Figure 7. Implementation of functions *tree_get_parent* and *tree_dispose*

```
fixpoint tree combine(context c, tree t) {
  switch (c) {
    case root : t;
    case left_context(pns, p, right) :
      combine(pns, tree(p, t, right));
    case right_context(pns, p, left) :
      combine(pns, tree(p, left, t));
  }
}

inductive path := here | left(path) | right(path);

fixpoint bool contains_at(tree t, path p, node n) {
  switch (t) {
    case nil : return false;
    case tree(rootNode, l, r) : return
      switch (p) {
        case here : n = rootNode;
        case left(p) : contains_at(l, p, n);
        case right(p) : contains_at(r, p, n);
      };
  }
}

fixpoint context context_at(context c, tree t, path p) {
  switch (p) {
    case here : return c;
    case left(p) : return switch (t) {
      case nil : c;
      case tree(n, l, r) :
        context_at(left_context(c, n, r), l, p);
    };
    case right(p) : ...analogous...
  }
}

fixpoint tree subtree_at(tree t, path p)) {
  switch (t) {
    case nil : return nil;
    case tree(n, l, r) : return switch (p) {
      case here : t;
      case left(p) : subtree_at(l, p);
      case right(p) : subtree_at(r, p);
    };
  }
}

lemma void change_focus(node n0, path p, node n);
  requires tree(n0, ?c, ?t) ∗ contains_at(combine(c, t), p, n);
  ensures tree(n, context_at(root, combine(c, t), p),
    subtree_at(combine(c, t), p));
```

Figure 8. Specification of lemma *change_focus*

```
int main()
  requires emp;
  ensures emp;
{
  node root := create_tree();
  node l := tree_add_left(root);
  node lr := tree_add_right(l);
  change_focus(lr, left(here), l);
  node ll := tree_add_left(l);
  change_focus(ll, left(right(here)), lr);
  node lrr := tree_add_right(lr);
  change_focus(lrr, here, root);
  tree_dispose(root);
  return 0;
}
```

Figure 9. Client code that uses *change_focus* for non-contiguous navigation

```
lemma void go_to_root(node n, context c)
  requires tree(n, c, ?t);
  ensures tree(_, root, combine(c, t));
{
  switch (c) {
    case root :
    case left_context(pcn, p, r) :
      open tree(n, c, t);
      open context(n, _, _, _);
      assert context(p, ?gp, _, _);
      close subtree(p, gp, tree(p, t, r));
      go_to_root(p, pcn);
    case right_context(···) : ... analogous ...
  }
}

lemma void go_to_descendant(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(t, p, n);
  ensures tree(n, context_at(c, t, p), subtree_at(t, p));
{
  switch (p) {
    case here :
      open tree(n0, c, t);
      open subtree(n0, ?p, t);
      switch (t) {
        case nil :
        case tree(n00, l, r) :
          close subtree(n0, p, t);
          close tree(n0, c, t);
      }
    case left(p) :
      open tree(n0, c, t);
      open subtree(n0, ?p, t);
      switch (t) {
        case nil :
        case tree(n00, l, r) :
          node left := n0→left;
          close context(left, n0, count(l),
            left_context(c, n0, r));
          close tree(left, left_context(c, n0, r), l);
          go_to_descendant(left, p, n);
      }
    case right(p) : ... analogous ...
  }
}

lemma void change_focus(node n0, path p, node n)
  requires tree(n0, ?c, ?t) * contains_at(combine(c, t), p, n);
  ensures tree(n, context_at(root, combine(c, t), p),
    subtree_at(combine(c, t), p));
{
  go_to_root(c);
  assert tree(?rootNode, _, _);
  go_to_descendant(rootNode, p, n);
}
```

Figure 10. Proof of lemma *change_focus* and auxiliary lemmas