Partial Solutions to VerifyThis 2016 Challenges 2 and 3 with VeriFast

Bart Jacobs
iMinds-DistriNet, Dept. of Computer Science, KU Leuven, Belgium
bart.jacobs@cs.kuleuven.be

ABSTRACT

We describe our partial solutions, using our VeriFast separation-logic based tool for modular formal verification of C and Java programs, to Challenges 2 and 3 of the VerifyThis 2016 Verification Competition, involving the verification of crash-freedom and certain correctness properties of code fragments implementing constant-space tree traversal and a tree barrier.

CCS Concepts

•**Theory of computation** → **Program verification;**

1. INTRODUCTION

VeriFast[1] is a research prototype being developed in our group of a tool for modular formal verification of correctness properties of single-threaded and multithreaded C and Java programs. It takes as input source code annotated with specifications written in a form of separation logic, as well as auxiliary logical definitions and proof hints. It then, without further user interaction and typically in a matter of seconds, reports either "0 errors found" or a failing symbolic execution trace. It operates by symbolically executing each C function or Java method in isolation, starting from a symbolic state representing an arbitrary execution state that satisfies the precondition, and checking at each return point that the postcondition is satisfied. Symbolic execution of a call uses the callee's specification instead of its implementation; similarly, symbolic execution of a loop requires the presence of a user-specified loop invariant.

Using VeriFast, we participated in the VerifyThis 2016 Verification Competition organized by Marieke Huisman, Rosemary Monahan, and Peter Müller on April 2, 2016 at ETAPS 2016 in Eindhoven. Twelve other teams participated, using 8 different tools: Dafny (4×), Why3 (2×), KIV, KeY, CIVL, VerCors, Viper, and mCRL2. We ended up being declared the Best Team.

[1] `https://github.com/verifast/verifast`

FTfJP'16, July 19, 2016, Rome, Italy

DOI: http://dx.doi.org/10.1145/2955811.2955818

The competition consisted of three challenges, involving the verification of crash-freedom and certain correctness properties of code fragments implementing matrix multiplication, constant-space tree traversal, and a tree barrier. In this paper, we describe partial solutions to Challenges 2 and 3. These are completed and cleaned-up versions of the solutions we submitted during the competition. The versions discussed here are in directory `examples/verifythis2016` at `https://github.com/verifast/verifast`.

2. BACKGROUND: SEPARATION LOGIC

In VeriFast, a method specification specifies the method's *heap effect* (i.e. the set of memory locations read or written by the method) by means of *separation logic* [4], as follows. In separation logic, assertions (including method preconditions and postconditions) are interpreted under *partial heaps*. Unlike a regular heap, a partial heap specifies the value for only a subset of the fields of the currently allocated objects (and for only a subset of the elements of the currently allocated arrays). Correspondingly, expressions in separation logic assertions are not allowed to dereference memory locations, since that would not make sense when interpreted under a partial heap that does not include that location. Rather, separation logic assertions may refer to the value at a memory location ℓ only by means of a *points-to assertion* $\ell \mapsto v$ (denoted `l |-> v` in VeriFast), which holds under a partial heap h if h includes location ℓ and associates value v with it. Furthermore, an assertion $P * Q$, where $*$ is called the *separating conjunction* and is denoted by `&*&` in VeriFast, holds for a partial heap h if h can be split into disjoint subheaps h_1 and h_2 such that P holds for h_1 and Q holds for h_2. It follows that an assertion $\ell \mapsto v * \ell' \mapsto v'$ implies $\ell \neq \ell'$.

In separation logic, a correctness judgment $\{P\}\ s\ \{Q\}$ holds if, when executing statement s in a heap h that extends some partial heap h_0 such that $h = h_0 \uplus h_\mathsf{F}$ and h_0 satisfies P, then s does not go wrong and if it terminates, the final heap h' can be written as $h' = h'_0 \uplus h_\mathsf{F}$ such that h'_0 satisfies Q. In other words, $\{P\}\ s\ \{Q\}$ implies that allocated memory locations not asserted by P are not modified by s. Stated differently still, s only has *permission* to modify the locations asserted by P.

In plain separation logic, asserting a memory location at all in a precondition implies asserting permission for the statement to modify that memory location. To allow a precondition to assert a memory location without asserting write permission, VeriFast uses an extension of separation logic with *fractional permissions* [1]. Under this extension,

assertions are interpreted under *fractional heaps*, which are partial heaps that additionally associate with each location that they contain a positive real number, called that location's *fraction*. A fraction of 1 denotes read-write permission, whereas a fraction less than 1 denotes read-only permission. A fractional points-to assertion $\ell \overset{\pi}{\mapsto} v$ (denoted by `[`π`]`ℓ`|->`v in VeriFast) holds for a fractional heap h if h contains location ℓ with (at least) fraction π. We can reinterpret the above definition of the meaning of correctness judgments to support fractional permissions as follows: let h_0, h_0', and h_F range over fractional heaps, interpret $\uplus$ as performing a pointwise addition of fractions, and interpret real heaps h and h' as fractional heaps that associate each location with fraction 1. Under that interpretation, if a correctness judgment holds and the precondition asserts only fractional permission for some location, then the statement does not modify that location.

In VeriFast, an assertion `[`π`]`P holds under a fractional heap h if P holds under some heap h' such that $h = \pi h'$, where $\pi h'$ denotes pointwise multiplication of each fraction in h' by π. Furthermore, `[_]`P denotes $\exists \pi.$ `[`π`]`P. Assertions of the form `[_]`P have the useful property that `[_]`$P \Leftrightarrow$ `[_]`$P *$ `[_]`P, which makes them preferable in cases where the locations described by P are not modified.

Separation logic supports concurrency trivially: if $\{P\}\ s\ \{Q\}$ and $\{P'\}\ s'\ \{Q'\}$, then $\{P * P'\}\ s\ ||\ s'\ \{Q * Q'\}$ (where $||$ denotes parallel composition). Conceptually, thread s *owns* the memory locations (or fractions thereof) asserted by P. Locks enable *ownership transfer* between threads: if we associate with each lock ℓ a *lock invariant* I_ℓ that asserts the permissions *owned by the lock* when it is not held, then we have $\{P * I_\ell\}$ **init** ℓ $\{P\}$, $\{P\}$ **acquire** ℓ $\{P * I_\ell\}$ and $\{P * I_\ell\}$ **release** ℓ $\{P\}$. Conceptually, acquiring a lock transfers ownership of the permissions held by the lock to the thread, and initializing or releasing a lock transfers the permissions asserted by the lock invariant to the lock.

3. CHALLENGE 2: TREE TRAVERSAL

The following Java code was given:

```
class Tree {
  Tree left, right, parent;
  boolean mark;
  static void markTree(Tree root) {
    Tree x = root; Tree y;
    do {
      x.mark = true;
      if (x.left == null && x.right == null) {
        y = x.parent;
      } else {
        y = x.left; x.left = x.right;
        x.right = x.parent; x.parent = y;
      }
      x = y;
    } while (x != null);
  }
}
```

The task was to prove that, if method `markTree` is given a non-null root of a well-formed tree (the root's `parent` field holds null and the `parent` field of each child of node n holds n) and each node has 0 or 2 children, then it does not crash, it terminates, it preserves the tree shape, and it sets each node's `mark` bit. (We ignore the bonus task, which we did not address.)

In this section, we present a proof that this code has this property.

Our specification is as follows:

```
static void markTree(Tree root)
//@ requires tree(root, false, null, ?rootShape);
//@ ensures tree(root, true, null, rootShape);
```

where predicate `tree` is defined as follows:

```
predicate tree(Tree node, boolean marked;
               Tree parent, tree shape) =
    node.left |-> ?left &*&
    node.right |-> ?right &*&
    node.mark |-> ?mark &*&
    (marked ? mark == true : true) &*&
    node.parent |-> parent &*&
    left == null ?
        right == null &*&
        shape == empty(node)
    :
        right != null &*&
        tree(left, marked, node, ?leftShape) &*&
        tree(right, marked, node, ?rightShape) &*&
        shape == nonempty(node, leftShape, rightShape);
```

and type `tree` is defined as follows:

```
inductive tree =
  empty(Tree node)
| nonempty(Tree node, tree left, tree right);
```

The construct `?rootShape` is a binding construct. It introduces the *logical variable* `rootShape`, whose scope extends to the end of the assertion, or, if it is introduced in a precondition, to the end of the method specification. The specification of `markTree`, then, states that the given tree can have any shape, and, in the post-state, it will have the same shape.

We believe the reader will easily see that this specification expresses the stated property, except for termination. Notice that VeriFast uses C's ternary expression notation for conditional assertions b `?` P `:` Q, which in separation logic would be expressed as $b \wedge P \vee \neg b \wedge Q$. (VeriFast supports disjunction of boolean expressions b `||` b', but it does not directly support general disjunction of assertions.)

Verifying this program means finding a loop invariant, and, for proving termination, a loop variant. Ours are as follows:

```
/*@ invariant x != null &*&
      inv(?xIsNew, x, root, rootShape, ?stepsLeft); @*/
//@ decreases stepsLeft;
```

where `xIsNew` is true if node `x` is visited for the first time, and predicate `inv` is defined as follows:

```
predicate inv(boolean xIsNew, Tree x,
              Tree root, tree rootShape, int stepsLeft) =
  xIsNew ?
    tree(x, false, ?parent, ?xShape) &*&
    stack(parent, x, xShape,
          root, rootShape, ?stepsLeft1) &*&
    stepsLeft1 >= 0 &*&
    stepsLeft == node_count(xShape) * 2 - 1 + stepsLeft1
  :
    stack(x, ?child, ?childShape,
          root, rootShape, stepsLeft) &*&
    stepsLeft >= 0 &*&
    tree(child, true, x, childShape);
```

The definition of predicate `inv` is based on the observation that the algorithm encodes in the fields of the nodes on the path (in the original tree) between the root and the current node the information that, in a naive recursive tree marking

procedure, would be stored on the call stack. We use predicate `stack` to describe these nodes and to relate their state to the original tree. Specifically, `stack`(h, c, cs, r, rs, m) describes all nodes of the tree except for those of the subtree (in the original tree) rooted in c, the *current child* of h, the head of the stack; furthermore, it states that if the shape of c is cs, then the stack corresponds to an original tree with root r and shape rs. Finally, m is the number of loop iterations needed to mark the unvisited nodes described by the predicate.

The definition of `inv` distinguishes between two cases: if node `x` is being visited for the first time, then the "call stack" starts at `x`'s parent, and `x` is its parent's current child; furthermore, the number of loop iterations remaining equals two iterations per node below `x` (one to enter the node's subtree and one to leave it), plus one (to leave `x`'s subtree), plus the number of iterations needed to process the call stack. Otherwise, `x` itself is the head of the call stack, and the algorithm has just finished marking its current child.

Mathematical function `node_count` is defined as follows:

```
fixpoint int node_count(tree tree) {
    switch (tree) {
        case empty(node): return 1;
        case nonempty(node, left, right): return
            1 + node_count(left) + node_count(right);
    }
}
```

Predicate `stack` is defined as follows:

```
predicate stack(Tree parent, Tree current, tree cShape,
                Tree root, tree rootShape,
                int stepsLeft) =
    current != null &*&
    parent == null ?
        root == current &*& rootShape == cShape &*&
        stepsLeft == 0
    :
        parent.left |-> ?left &*&
        parent.right |-> ?right &*&
        parent.mark |-> true &*&
        parent.parent |-> current &*&
        exists<boolean>(?currentIsLeftChild) &*&
        currentIsLeftChild ?
            tree(left, false, parent, ?rightShape) &*&
            left != null &*&
            stack(right, parent,
                  nonempty(parent, cShape, rightShape),
                  root, rootShape, ?stepsLeft1) &*&
            stepsLeft1 >= 0 &*&
            stepsLeft == node_count(rightShape) * 2
                         + 1 + stepsLeft1
        :
            tree(right, true, parent, ?leftShape) &*&
            right != null &*&
            stack(left, parent,
                  nonempty(parent, leftShape, cShape),
                  root, rootShape, ?stepsLeft1) &*&
            stepsLeft1 >= 0 &*&
            stepsLeft == 1 + stepsLeft1;
```

The predicate's body distinguishes between three cases. If the head of the stack (denoted by parameter `parent`) is null, then the current child is the root. Otherwise, either the left child or the right child of the head (in the original tree) is the current child. If the left child is the current child, then field `left` of the head currently holds the head's original right child, field `right` holds its original parent, and field `parent` holds its original left child. The first conjunct in this branch of the predicate describes the right child's subtree, which has not yet been marked. The third describes the tail of the stack.

If the original right child is the current child, then field `left` currently holds the original parent, and field `right` holds the original left child, which has already been marked.

The definition uses the `exists` predicate to simulate a disjunction between two assertions. It is defined as follows:

```
predicate exists<t>(t x) = true;
```

Using this simple pattern, one can encode $\exists x : \tau.\ P(x)$ as `exists<`τ`>(?`x`) &*&` $P(x)$ and, more specifically, $P \vee Q$ as `exists<boolean>(?b) &*& b ?` P `:` Q.

To finish the proof, what remains is to prove that the loop invariant holds initially, that it implies memory safety of the loop body, that it is preserved by the loop body, and that it, conjoined with the negation of the loop condition, implies the postcondition.

To establish the loop invariant initially, we need the following ghost commands:

```
Tree x = root; Tree y;
//@ tree_nonnull(x);
//@ close stack(null, root, rootShape, root, rootShape, 0);
//@ close inv(true, x, root, rootShape, _);
do
```

The first ghost command is an invocation of the `tree_nonnull` lemma, to establish that `x` is not null. The lemma is defined as follows:

```
lemma void tree_nonnull(Tree t)
  requires tree(t, ?marked, ?parent, ?shape);
  ensures tree(t, marked, parent, shape) &*& t != null;
{
  open tree(t, marked, parent, shape);
  close tree(t, marked, parent, shape);
}
```

The lemma establishes the property simply by applying the definition of predicate `tree`. The advantage of calling this lemma over applying the definition directly is that calling the lemma avoids the case split caused by the conditional assertion in the definition of the predicate when applying it.

The other ghost commands apply the definition of predicates `stack` and `inv` to establish that the predicates hold for the given arguments. Notice that the final argument for predicate `inv` is not specified; it is inferred by VeriFast.

To establish that the loop invariant implies that we have sufficient permission to perform the field accesses performed by the loop body, we need the following ghost commands:

```
//@ open inv(_, _, _, _, _);
//@ if (!xIsNew) open stack(x, _, _, _, _, _);
x.mark = true;
```

If `x` is new, then the write permissions for `x`'s fields are in the `tree` predicate, which is automatically unfolded by VeriFast because it is declared as *precise* by using a semicolon in the parameter list to separate the *input parameters* from the *output parameters*. Otherwise, however, the write permissions for `x`'s fields are in the `stack` predicate, which is not automatically unfolded because it is not declared as precise, so we unfold it manually.

To establish that the loop body preserves the loop invariant, we distinguish two cases. If `x` has no children, we simply apply the definition to establish the invariant:

```
if (x.left == null && x.right == null) {
  y = x.parent;
  //@ close inv(false, y, root, rootShape, _);
```

In the other case, we have more work:

```
x.right = x.parent; x.parent = y;
/*@
if (xIsNew) {
  assert tree(y, false, x, ?leftShape);
  close exists(true);
  close stack(x, y, leftShape, root, rootShape, _);
  close inv(true, y, root, rootShape, _);
} else {
  open exists(?markedLeftSubtree);
  if (markedLeftSubtree) {
    close exists(false);
    assert tree(y, false, x, ?rightShape);
    tree_nonnull(x.right);
    close stack(x, y, rightShape, root, rootShape, _);
    close inv(true, y, root, rootShape, _);
  } else {
    close inv(false, y, root, rootShape, _);
  }
}
@*/
```

There are three cases: if `x` is new, `x` becomes the head of the stack and its left child becomes the current child. The `close exists(true);` command encodes that the first branch of the disjunction is to be chosen when folding the `stack` predicate in the next command. To establish `inv`, we need the property that function `node_count` always returns a positive value. We make VeriFast aware of it by means of the following lemma:

```
lemma_auto void node_count_positive(tree tree)
  requires true;
  ensures node_count(tree) >= 1;
{
  switch (tree) {
    case empty(node):
    case nonempty(node, left, right):
      node_count_positive(left);
      node_count_positive(right);
  }
}
```

The lemma establishes the property by structural induction on the `tree` value. (Note that **`switch`** statements over inductive values do not have fall-through semantics, so no **`break`** statement is needed before the second **`case`**.) Declaring the lemma as **`lemma_auto`** causes VeriFast to emit this lemma as an axiom, causing the SMT solver to instantiate it for each term of the form `node_count`(t) it encounters.

If `x` is not new, we have just marked either the left subtree or the right subtree. The **`open`** `exists(?markedLeftSubtree)` command binds ghost variable `markedLeftSubtree` to true if the first disjunct was true when unfolding `stack` at the top of the loop body, and false otherwise. If we just marked the left subtree, the right child has just become the current child, so the second disjunct holds when folding `stack`, which we encode using the **`close`** `exists(false);` command. If we just marked the right subtree, we are now returning to the parent.

To establish the postcondition after exiting the loop, we just need to apply the definition of predicates `inv` and `stack`:

```
} while (x != null);
//@ open inv(_, _, _, _, _);
//@ open stack(null, _, _, _, _, _);
```

This completes the proof. VeriFast checks the proof in less than a tenth of a second.

Discussion. 79 lines of annotations for 17LOC means an overhead of 5×. However, arguably what is stated by the proof is what would be stated in a manual proof for human readers; arguably the proof is as elegant, and VeriFast supports this challenge as well as one might wish for.

4. CHALLENGE 3: TREE BARRIER

The following code was given:

```
class Node {
  final Node left, right, parent;
  volatile boolean sense;
  void barrier()
    // requires !sense
    // ensures !sense
  {
    if (left != null) while (!left.sense) { }
    if (right != null) while (!right.sense) { }
    sense = true;
    if (parent == null) sense = false;
    while (sense) { }
    if (left != null) left.sense = false;
    if (right != null) right.sense = false;
  }
}
```

(We omit the `version` field, which is only relevant for Task 2, which we did not address.) Consider an immutable well-formed tree of N `Node` objects. Assume N threads are running, each associated with a distinct node and repeatedly calling `barrier()` on its node. Assume that initially all `sense` fields hold **`false`**. Task 1 (the only task we addressed) was to prove that at any point, if n`.sense` holds **`true`** for any node n then m`.sense` holds **`true`** for all nodes m in the subtree rooted in n.

We proved (an encoding of) this property for the following slight variant of the given code:

```
class Node {
  final Node left, right, parent;
  final AtomicBoolean sense;
  //@ final tree leftTree, rightTree;
  //@ boolean senseValue, grabbed, takenBack;

  static void grab(Node child) {
    if (child != null)
      for (;;)
      { boolean r = child.sense.get(); if (r) break; }
  }
  static void ungrab(Node child) {
    if (child != null) { child.sense.set(false); }
  }
  void barrier() {
    grab(left); grab(right);
    { sense.set(true); }
    if (parent == null) { sense.set(false); } else {
      for (;;)
      {{ boolean r = sense.get(); if (!r) break; }}
    }
    ungrab(left); ungrab(right);
  }
}
```

Mainly, we replaced the **`volatile`** field (which VeriFast does not support) by an `AtomicBoolean` (provided by the Java Platform API in package `java.util.concurrent.atomics`), for which we used the following specification:

```
/*@
```

```
typedef lemma void get_op(predicate(boolean) inv,
                          predicate() pre,
                          predicate(boolean) post)();
  requires inv(?value) &*& pre();
  ensures inv(value) &*& post(value);

typedef lemma void set_op(predicate(boolean) inv,
                          boolean value,
                          predicate() pre,
                          predicate() post)();
  requires inv(?value0) &*& pre();
  ensures inv(value) &*& post();
@*/

class AtomicBoolean {
  //@ predicate valid(predicate(boolean) inv);

  AtomicBoolean();
  /*@ requires exists<predicate(boolean)>(?inv) &*&
               inv(false); @*/
  //@ ensures valid(inv);

  boolean get();
  /*@ requires [_]valid(?inv) &*&
               is_get_op(?op, inv, ?pre, ?post) &*&
               pre(); @*/
  //@ ensures post(result);

  void set(boolean value);
  /*@ requires [_]valid(?inv) &*&
               is_set_op(?op, inv, value, ?pre, ?post) &*&
               pre(); @*/
  //@ ensures post();
}
```

To understand this specification [2], it is helpful to think of the class as being implemented using a lock, as follows:

```
class AtomicBoolean {
  boolean value;
  // lock_inv: value |-> ?v &*& inv(v);
  synchronized boolean get()
  { /*@ op(); @*/ return value; }
  synchronized void set(boolean v)
  { /*@ op(); @*/ value = v; }
}
```

That is, the lock invariant is parameterized by a client-provided predicate `inv`, which may mention the value. Furthermore, each method specification is parameterized by client-provided predicates `pre` and `post`, where `post` may mention the return value, if any. Method `set` requires the client to supply a *lemma pointer* `op` that establishes `inv` for the new value. Specifically, the supplied lemma must satisfy the *lemma type* `set_op`, which is parameterized by four *lemma type parameters* `inv`, `value`, `pre`, and `post`. Similarly, method `get` requires a *lemma pointer* that establishes the client-supplied postcondition. We shall see below how this allows us to verify rich properties, including the Task 1 property.

In fact, we encode the property to be verified into the `inv` predicate. Notice that since `AtomicBoolean` is in reality implemented using atomic machine instructions, an `AtomicBoolean` object's `inv` predicate really holds in each execution state after its creation. The predicate we use is as follows:

```
predicate_ctor Node_inv(Node node)(boolean value) =
    [_]tree1(node, ?parent, ?tree) &*&
    [1/2]node.senseValue |-> value &*&
    value ?
        [1/2]senseValuesTrue0(tree) &*&
        [1/2]node.grabbed |-> ?grabbed &*&
        [1/2]node.takenBack |-> false &*&
        grabbed ?
            parent != null
        :
            (parent == null ? true : senseValuesTrue(tree))
    :
        [1/2]node.grabbed |-> false &*&
        [1/2]node.takenBack |-> ?takenBack &*&
        takenBack ?
            true
        :
            [1/2]node.senseValue |-> false &*&
            senseValuesTrue0(tree);
```

Keyword `predicate_ctor` introduces a *predicate constructor*, which is simply a predicate which can be partially applied. Here, we partially apply predicate `Node_inv` to a node to obtain a unary predicate, as required by `AtomicBoolean`.

Predicate `tree1`(n, p, t) asserts that n is the non-null root of a subtree with parent p and shape t, where t is of inductive datatype `tree` defined as follows:

```
inductive tree =
  empty
| tree(Node node, tree left, tree right);
```

The predicate itself is defined as follows, by mutual recursion with predicate `tree`:

```
predicate tree(Node node, Node parent; tree tree) =
    node == null ?
        tree == empty
    :
        tree1(node, parent, tree);

predicate tree1(Node node; Node parent, tree tree) =
    [_]node.sense |-> ?sense &*&
    [_]sense.valid(Node_inv(node)) &*&
    [_]node.left |-> ?left &*&
    [_]node.parent |-> parent &*&
    [_]node.leftTree |-> ?leftTree &*&
    [_]tree(left, node, leftTree) &*&
    [_]node.right |-> ?right &*&
    [_]node.rightTree |-> ?rightTree &*&
    [_]tree(right, node, rightTree) &*&
    tree == tree(node, leftTree, rightTree);
```

We introduce a ghost field `senseValue`, whose value is in each execution state equal to the value of the `AtomicBoolean`. By passing fractions of this ghost field around, we can mention the value of the `AtomicBoolean` outside of its invariant.

If n`.sense` is set, its invariant contains `[1/2]senseValuesTrue0`(t), where t is the shape of the subtree below n. This encodes the Task 1 property: this predicate asserts a fraction of the `senseValue` field of each node in t (except for the root) and asserts that its value is `true`. It is defined by mutual recursion with predicate `senseValuesTrue`:

```
predicate senseValuesTrue(tree tree;) =
    switch (tree) {
        case empty: return true;
        case tree(node, left, right): return
            [1/2]node.senseValue |-> true &*&
            [1/2]senseValuesTrue0(tree);
    };

predicate senseValuesTrue0(tree tree;) =
    switch (tree) {
        case empty: return true;
        case tree(node, left, right): return
            senseValuesTrue(left) &*&
            senseValuesTrue(right);
    };
```

We introduce two further boolean ghost fields: `grabbed` and `takenBack`. Field n.`grabbed` denotes whether n's parent has "grabbed" a fraction of the `senseValue` fields of the subtree t rooted in n, as described by predicate `senseValuesTrue`(t). While the parent has not yet grabbed these, they are in n's invariant[2]. The parent puts these fractions back into its children's invariants when it clears their `sense` bits. They remain there until the child thread "takes them back" out of the invariant into thread-local ownership in preparation for ungrabbing its own children. This event is registered by field `takenBack`.

The remainder of the proof, then, is to show that this invariant is maintained by each operation on a `sense` bit performed by method `barrier`. This, of course, depends on `barrier`'s specification, which is as follows:

```
void barrier()
  //@ requires valid();
  //@ ensures  valid();
```

where predicate `valid` is defined as:

```
predicate valid() =
  [_]tree1(this, ?parent, ?thisTree) &*&
  [1/2]this.senseValue |-> false &*&
  [1/2]this.takenBack |-> true &*&
  (parent == null ? [1/2]this.grabbed |-> false : true) &*&
  [_]this.left |-> ?left &*& child(this, left) &*&
  [_]this.right |-> ?right &*& child(this, right);
```

and predicate `child` is defined as:

```
predicate child(Node parent, Node child;) =
    parent != null &*&
    child == null ?
        true
    :
        [_]tree1(child, parent, _) &*&
        [1/2]child.grabbed |-> false;
```

That is, when not inside the barrier, a thread's `sense` bit is **`false`** and its children are not grabbed.

We describe one `sense` bit operation proof; the others are similar. In particular, we consider the proof of helper method `grab`, whose specification is as follows:

```
static void grab(Node child)
//@ requires child(?parent, child);
//@ ensures child_grabbed(parent, child);
```

where predicate `child_grabbed` is defined as:

```
predicate child_grabbed(Node parent, Node child;) =
    parent != null &*&
    child == null ?
        true
    :
        [1/2]child.grabbed |-> true &*&
        [_]tree1(child, parent, ?childTree) &*&
        senseValuesTrue(childTree);
```

It asserts that the child is now grabbed as well as that the sense values of the child's subtree are true. The loop invariant is simply

```
//@ invariant child(parent, child);
```

The proof of the `child.sense.get()` call is as follows:

```
/*@
predicate pre() = child(parent, child);
predicate post(boolean value) =
    child_grabbed(parent, child) : child(parent, child);
lemma void get_op()
    requires Node_inv(child)(?value) &*& pre();
    ensures Node_inv(child)(value) &*& post(value);
{
    open pre();
    open Node_inv(child)(value);
    if (value) child.grabbed = true;
    close Node_inv(child)(value);
    close post(value);}
@*/
//@ open [_]tree1(child, _, _);
/*@ produce_lemma_function_pointer_chunk(get_op) :
      get_op(Node_inv(child), pre, post)()
      { call(); }; @*/
//@ close pre();
boolean result = child.sense.get();
//@ open post(result);
```

We define block-local predicates `pre` and `post` and block-local lemma `get_op` to instantiate the `get` method's specification. The **`produce_lemma_function_pointer_chunk`** ghost command tells VeriFast to check that lemma `get_op`'s specification is compatible with that of the `get_op` lemma type defined as part of the `AtomicBoolean` specification above, and if that check succeeds, to produce an `is_get_op` chunk that expresses the compatibility, as required by method `get`'s precondition.

Discussion. 190 lines of annotations for 25LOC means an overhead of 8×. Less would be better.

In this proof, the inter-thread protocol is encoded using fractional permissions and ghost fields. In more complex cases, this encoding can be cumbersome, and more direct alternatives are called for (e.g. [3]); however, in the present case this simple approach sufficed.

If we add support for **`volatile`** fields to VeriFast, we will probably treat them as if they were `Atomic`T objects.

5. CONCLUSIONS

We presented partial solutions in VeriFast to Challenges 2 and 3 of the VerifyThis 2016 Verification Competition. VeriFast features showcased include the encoding of disjunction, the use of dummy fractions (`[_]`) to easily handle immutable data structures, and VeriFast's support for reasoning about fine-grained concurrency.

As of this writing, no solutions to these challenges by other teams have appeared on the VerifyThis home page, so a comparison is not possible. However, multiple alternative solutions to Challenge 1 have been proposed, all of which are more elegant than our solution in VeriFast. Our choice of defining matrices and matrix multiplication recursively was a poor one. Using matrix comprehension and quantification leads to a more elegant proof.

6. REFERENCES

[1] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.

[2] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, 2011.

[3] R. Jung et al. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

[4] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[2] i.e. owned by n's `AtomicBoolean`