VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0

K. Rustan M. Leino and Michał Moskal

Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Manuscript KRML 209, 21 July 2010.

Abstract. This paper defines a suite of benchmark verification problems, to serve as an acid test for verification systems that reason about full functional correctness of programs with non-trivial data-structure invariants. Solutions to the benchmarks can be used to understand similarities and differences between verification tools and techniques. The paper also gives a procedure for scoring the solutions.

0 Introduction

There are many program verification systems today, and we expect many more to be developed in the next decade. The systems differ in what programming language they handle, in the underlying logic and the specification features that they support, in the style in which specifications are written, in how one interacts with the system, and in the level of automation provided by the system. Yet, in the end, the tools all have a similar goal: to increase our confidence level in particular algorithms or pieces of code, and to do so in a convincing way without spending too much effort. So, how do we measure and compare the capabilities of various tools, how do we measure the improvements of a given tool over time, and how we compare how well tools live up to the common goal?

In this paper, we propose a set of challenging verification benchmarks, aimed to facilitate such measurements and comparisons. These benchmarks are for modular verification of functional correctness of programs where the data structures themselves rely on interesting consistency conditions. That is, the benchmarks were chosen as ones that involve useful and non-trivial data structures, not just non-trivial algorithms. Please note that these benchmarks are unlikely to be suitable for model-checking, testing, or extended static checking tools, with limited specification capabilities.

We propose the following five benchmarks, mostly freshman data structures.

0. **Constant-time spare array**, where `init`, `get` and `set` operations all take constant time. A simple warm-up problem exercising invariants over arrays.
1. **Composite pattern**, which has been proposed before. There is an update in the middle of a tree pointer structure, and then the algorithm walks the tree up to fix the invariant.
2. **Binary heap** is expected to be implemented with a integer-indexed data structure. We use heap-sort as a test-harness.

3. **Union-find** data structure for maintaining partitioning of a set. We use a maze generation algorithm (essentially Kruskal's spanning tree construction) as a test-harness.
4. **Red-black trees**. The classic. Likely most difficult in the set.

While challenging, we believe the benchmarks are solvable with state-of-the-art tools. We have started verification of some using the Dafny [6] and VCC [1] tools, and convinced ourselves that the others can be also verified. Thus, we expect the comparison to happen based on the performance and annotation overhead of the tools (see Sect. 1 below).

The benchmarks do not rely on object-orientation, type parameters, pointer arithmetic, unsafe features, particular concurrency handling, etc. It is thus possible to implement them in C or Java-like languages. The data structures and signatures that we give are imperative, thus performing verification in a functional programming language may be challenging.

The selection of benchmarks is clearly rather arbitrary. It would be ideal to have some objective benchmarks provided by outsiders, however we do not know of any such benchmarks. The situation is similar in the theorem prover competitions (CAST, SMT-COMP, SAT-race), where authors of the tools are allowed to submit benchmarks. The idea is, of course, that authors of *all* tools can submit benchmarks, giving a good mix in the end. For example, SMT-LIB welcomes any benchmarks, but during competition the benchmarks arising from actual applications (as opposed to random or handcrafted) are given higher weights. In that spirit, a verification benchmark is valuable as long as it is something users will want to verify. We we believe that this is the case for all benchmarks presented here. The initial feedback we have received about our benchmarks from the verification community was overall very positive.

Each benchmark comes with a signature of operations available on the data structure and a "test harness"—a short program measuring the completeness of specifications of the data structure. We intentionally do not give exact specifications of the data structure operations, as these may vary between tools.

We also give a scoring scheme for the benchmarks. The scores are broken down so that it is possible to get points even if not all interesting properties to verify have been verified. Parts of the scores come from conciseness of specifications and other overhead for the proof, as well as from the performance of the realized verification tool.

Our goals are:

– to benchmark the ability of verification tools to verify the correctness of algorithms/-executable code that involve non-trivial data structure invariants, and
– to benchmark their support for modular checking and facilities for abstraction.

Having been inspired by a first suite of verification benchmarks, presented at VSTTE 2008 by Weide *et al.* [10], and having attempted to solve those using Dafny [7], we have tried to give more precise problem statements, in particular through the test harness mechanism. Also, unlike the VSTTE 2008 benchmarks, we do not aim at incrementality (*i.e.*, that one benchmark would build on another); rather, we give independent problems where the harness makes use of the abstract specification of the data structure in the same benchmark.

We expect the benchmarks to be suitable tasks to most of the tools listed below (the list is of course not expected to be exhaustive): the general purpose interactive provers (Coq, Isabelle/HOL, ACL2); interactive verification environments (Ynot, KeY, KIV, RESOLVE); symbolc execution tools (jStar, VeriFast, VeriCool); and deductive verifiers using automatic provers in the backend (Dafny, VCC, Chalice, Jahob, and the Why tools (including Krakatoa and Frama-C)). Three efforts to undertake the benchmarks are already underway, by Jan Smans (using VeriCool 3 [8]) and by us (using VCC and Dafny). We have setup an open-source project[0], to collect solutions.

1 Requirements and Scoring

We expect *modular* solutions, *i.e.*, one should verify correctness of a reasonably abstract specification (each problem specifies at least one such abstraction), and then verify the harness using only the specification, not the implementation of the data structure.

We also expect that the verification includes the safety of all operations, like absence of null dereferences, array bounds errors, and other precondition violations.

Each problem is scored on a scale from 0 to 100 points, inclusive. The description of the problem gives scores for various subtasks, which add up to 80. Usually some of the subtasks depend on others. If any points are scored for any of those subtasks, then additional points are added:

- 5 points are added if the solution guarantees termination, *i.e.*, total correctness of the test harness and the other code is proven
- $6 - 2\ln\frac{g}{p}$ or 10 points (whichever is less) are added, where g is the number of tokens in the specification and all the additional lemmas/guidance needed for the verifier to verify the problem in batch mode and p is the number of tokens in the executable part of the program (including the harness)
- $5 - \frac{1}{2}\ln t$ or 5 points (whichever is less) are added, where t is the number of seconds the verifier needs in batch mode to verify the problem (including the harness)

Should the score for a benchmark come out negative (*i.e.*, the benchmark takes a couple of million years to verify or requires millions of lines of annotations), then the score is to be treated as 0 for that benchmark.

The result of the benchmark suite is the average number of points scored in individual benchmarks, where any benchmark not attempted counts as 0.

The definition of token will necessarily vary between languages. For languages with C-like syntax, we provide a script[1] to count them (which may need to be tweaked for a particular verifier).

The runtime should be measured as the wall-clock time on a modern (as of 2010) machine, *i.e.*, about a 3GHz PC. To reward parallel processing, any number of processors/cores are allowed, provided they all run concurrently during the interval wall-clock time reported. The measurement must, however, be done on an actual machine—it should not be constructed as a number that one, in principle, could obtain.

[0] `http://vacid.codeplex.com/`

[1] `http://research.microsoft.com/~moskal/count-tokens.zip`

Ideally, we would like to measure the tool's response time to the user while the user is developing the specification. That would mean computing the response time for a single method (or whatever the unit of specification/verification is) for a *failed* verification. However, since this is difficult to compare between different tools, we have instead settled for measuring the time needed for a successful verification of each entire benchmark.

Let us shortly motivate the scoring. As for the subtasks, we think that solving the problem at all is more important than annotation overhead or performance (this is similar to various theorem prover competitions). In particular, solving another subtask should not lower the overall score. This is why each subtask is worth at least 20 points. Other than that the amounts of points are just a guess, in some cases educated by our own attempts to solve these, or similar, problems.

As for the overhead, let us look at some data points. The extended static checking tools (*e.g.*, ESC/Java [4]) have an overhead of about 0.1 (which gives 10 points). Tools aiming at functional verification are unlikely to do any better. Automatic deductive tools, like Dafny and VCC, tend to have overhead roughly between 1 and 4 (6 to 4 points). Finally, the impressive L4.verified [5] project, using interactive Isabelle/HOL prover to verify functional correctness of an operating system, is reporting overhead of 20 (which gives 0 points). Any higher overhead gives negative points on our simple benchmarks.

As for the time, for interactive work in an IDE the response time should be in the sub-second range, and thus the 1 second overall time gives the maximal 5 points. Otherwise, the verification tool is likely to be used similarly to the compiler, which gives warnings about the code in the ballpark of 1 to 10 minutes (3 to 2 points). Finally, the response time of over an hour seems to effectively prevent development of specification, and thus the overall time between 1 and 6 hours gets between 1 and 0 points. Anything longer gives negative points.

The pseudo-code that we use below to describe data structures, the signatures of operations, and the various test harnesses is similar to Java or C#. We use type **`uint`** to denote the unsigned integers used to index into arrays, mostly to clearly distinguish them from values stored in the arrays (*i.e.*, **`int`**'s); however, there is no requirement on solutions to use unsigned types. It is permitted and encouraged (but currently does not yield any extra points) to use a generic type instead of **`int`**, and a particular value instead of `0` for all collection data-structures. We use a statement **`assert`**`(P)` to say that we require condition `P` to be verified to always hold (at least for full-points solutions; see exact description with each task).

1.0 Solution Description

To facilitate comparison between tools, solutions should report for each benchmark:

- which verification tasks were solved
- whether termination was proven
- the size in tokens of executable code and annotations; include separate numbers for the data-structure itself and the harness (the assertions in harness count as executable code, as they only emulate a use of the data-structure)

- the time it takes to verify the solution with multiple cores, and also with a single core
- if applicable, the longest verification time of a method (also multi- and single-core)
- test machine configuration (in particular, number of cores, if they were used, as well as any unusual resource requirement)
- the approximate time and number of people it took to develop the annotations, together with some indication of familiarity of the specification developers with the verification tools and techniques
- and, of course, the score computed as indicated above.

2 Constant-Time Sparse Array

For this benchmark, the task is to implement and verify an array where all three basic operations (create, get, and set) take constant time (Exercise 2.12 in [0]; see also [9] p. 271). Any memory requested from the underlying memory allocator (typically `malloc()` or **`new`**) should be treated as containing arbitrary values. The solution should use three arrays: one for the actual values stored in the array, and two more for marking which indices are already initialized, as in the program below.

```
class SparseArray {
  int val[MAXLEN];
  uint idx[MAXLEN], back[MAXLEN];
  uint n;
  static SparseArray create(uint sz) {
    SparseArray t = new SparseArray();
    n = 0;
    return t;
  }
  int get(uint i) {
    if (idx[i] < n && back[idx[i]] == i) return val[i];
    else return 0;
  }
  void set(uint i, int v) {
    val[i] = v;
    if (!(idx[i] < n && back[idx[i]] == i)) {
      assert(n < MAXLEN); // (*), see Verification Tasks
      idx[i] = n; back[n] = i; n = n + 1;
    }
  }
}
```

Verify that the program above meets the usual abstract interface of an array. The `get()` and `set()` methods should require the index to be within bounds, and the `create()` method may require that `sz <= MAXLEN`.

In a language like Java, the internal arrays allocated by the class above would already be `0`-initialized; however, our benchmark stipulates that the implementation is not allowed to make use of that fact in the verification. One way to ensure that is to make the constructor take the arrays as input.

The following test harness should be verified without looking at the particular implementation of the array.

```
void sparseArrayTestHarness() {
  SparseArray a = create(10), b = create(20);
  assert(a.get(5) == 0 && b.get(7) == 0);
  a.set(5, 1); b.set(7, 2);
  assert(a.get(5) == 1 && b.get(7) == 2);
  assert(a.get(0) == 0 && b.get(0) == 0);
}
```

2.0 Verification Tasks

0. Verify the correctness of the array implementation against your specification, assuming `n < MAXLEN` at the place marked with `(*)` in the code. Also, verify the correctness of the test harness using the specifications of the array. *50 points*.
1. As above, but without the assumption `(*)`. In other words, verify that the assertion holds (which ensures that the program does not reference the array outside its bounds). *30 points*.

3 Composite Pattern

This benchmark, which has been used as a specification and verification challenge at SAVCBS 2008 [3], involves a set of nodes connected acyclicly via `parent` links. In addition to graph structure information, each node stores an integer `val` as well as the sum of the `val` fields in the tree rooted at the node. The key is to keep these `sum` fields up-to-date.

A client is allowed access to any node in the set. That is, clients can hold pointers to any node. When the value of a node is updated, all relevant `sum` fields must be updated as well. This can be accomplished using recursion or by a loop like:

```
  void update(int v) {
    int diff = v - val;
    val = v;
    for (CompositeNode p = this; p != null; p = p.parent) {
      p.sum = p.sum + diff;
    }
  }
```

The signature of the class is:

```
class CompositeNode {
  CompositeNode parent;
  CompositeNode left, right;
  int val, sum;
  static CompositeNode create(int v);
  void addChild(CompositeNode child);  // connect 'this' as the parent of 'child'
  void dislodge();                     // disconnect 'this' from its parent
  void update(int v);
}
```

Here, we have limited nodes to two children, but programs are also allowed to support an unbounded number of children. The precondition of `addChild()` should prevent cycles from being created and prevent a node from getting more children than the implementation supports. Method `dislodge()` severs a node's tie to its parent. All methods must keep all `sum` fields up-to-date.

```
void compositeHarness()
{
  CompositeNode a = create(5), b = create(7);
  a.addChild(b);
  assert(a.sum == 12);
  b.update(17);
  assert(a.sum == 22);

  CompositeNode c = create(10);
  b.addChild(c); b.dislodge();
  assert(b.sum == 27);
}
```

It is allowed to modify the specification above to introduce an "manager" object to keep track of disjoint trees of composites.

3.0 Verification Tasks

0. Verify correctness of the harness. *80 points*.

4 Binary Heap

A binary min-heap (see [2], Chapter 6) is a nearly full binary tree, where the nodes maintain the *heap property*, that is, each node is smaller than each of its children. The heap should be stored in an integer-indexed collection (*e.g.*, an array). The following three operations should be provided:

```
class Heap {
  static Heap create(uint sz);
  void insert(int e);
  int extractMin();
}
```

The `create(sz)` method creates a new heap of maximum capacity `sz` (this restriction is optional). The `insert()` method should allow inserting an element multiple times so that `extractMin()` will return it multiple times.

The test harness consists of two procedures to be verified separately: a simple implementation of heap sort, and one use case.

```
void heapSort(int[] arr, uint len) {
  uint i;
  Heap h = create(len);
  for (i = 0; i < len; ++i) h.insert(arr[i]);
  for (i = 0; i < len; ++i) arr[i] = h.extractMin();
}
void heapSortTestHarness() {
  int[] arr = { 42, 13, 42 };
  heapSort(arr, 3);
  assert(arr[0] <= arr[1] && arr[1] <= arr[2]);
  assert(arr[0] == 13 && arr[1] == 42 && arr[2] == 42);
}
```

One may find the operation that “bubbles-up” an entry upon insertion to be similar in spirit to the composite pattern: we break the invariant at some point, and then move up fixing it. The motivation for including both benchmarks is that the composite pattern is supposed to be implemented with pointer structures, and the heap with an array (or another integer-indexed collection). Applications are likely to need both and thus we test for both.

4.0 Verification Tasks

0. Verify the that the heap sort returns an array that is sorted (in particular, verify the first assertion in the harness). *40 points*.
1. Verify that the heap represents a multiset, and thus that the heap sort produces a permutation of the input (in particular, verify the second assertion). *40 points*.

5 Union-Find

This benchmark makes use of the union-find data structure:

```
class UnionFind {
  static UnionFind create(uint sz);
  uint getNumClasses();
  int find(int a);
  void union(int a, int b);
}
```

Method `create()` creates a union-find data structure consisting of `sz` elements, identified by the integers from `0` to less than `sz`. Initially, each element is in an equivalence class by itself; that is, it is its own representative. In other words, the number of equivalence classes, which is returned by `getNumClasses()`, is initially `sz`. As usual, `find()` returns the representative element for a given element, and `union()` merges two equivalence classes.

The test harness constructs a random maze. The maze has size `n` times `n`. Thus, it is a graph with `n*n` nodes. The maze will be a spanning tree of that graph. The maze construction makes use of union-find as illustrated here:

```
uint rand();
void buildMaze(uint n) {
  UnionFind u = create(n*n);
  while (u.getNumClasses() > 1) {
    uint x = rand() % n, y = rand() % n, d = rand() % 2;
    uint w = x, z = y;
    if (d == 0) w++; else z++;
    if (w < n && z < n) {
      int a = y*n + x, b = w*n + z;
      if (u.find(a) != u.find(b)) {
        output edge ((x,y), (w,z));
        u.union(a, b);
      }
    }
  }
}
```

The verification should be performed for an arbitrary `rand()` function, even if an implementation is actually provided with the solution. The test harness shown above is acceptable for verifying partial correctness, but needs to be suitably modified in order to stand a chance of scoring points for termination.

The maze is considered correct if it is a spanning tree. More precisely, the task in this benchmark is to show that all `n*n` nodes are reachable (either from a particular node or that nodes are pairwise connected) and that there are `n*n-1` edges. (From these two properties, it follows that nodes are uniquely reachable.)

5.0 Verification Tasks

0. Verify the correctness of the maze creation. *60 points*.
1. Include path compression and node balancing in the implementation of the union-find algorithms. *20 points*.

The task 1 involves the two standard optimizations that make union-find efficient. We did not split them further, as node balancing is expected to be very easy to implement and verify. For task 0, any correct implementation of union-find is acceptable.

6 Red-black Trees

A red-black tree [2] is a commonly used kind of binary search tree where each node, in addition to the usual data and pointers, carries a bit of information referred to as the *color* (traditionally, either *red* or *black*). Tree operations maintain approximate balance by using rotation guided by colors of nodes. The task is to implement and verify a red-black tree providing a dictionary interface, like the one below:

```
class RedBlackTree {
  static RedBlackTree create(int defaultValue);
  void replace(int key, int value);
  void remove(int key);
  int lookup(int key);
}
```

The method `create(d)` creates a new dictionary mapping all keys to `d`. `replace(k, v)` replaces the current value associated with `k` by `v`. `remove(k)` is functionally equivalent to `replace(k, d)`, where `d` is the default value provided upon construction, but see Sect. 6.0 below. Finally, `lookup(k)` returns the current value associated with `k`.

The following test harness should be verified without looking at the particular implementation of red-black trees; that is, the interface above should be specified using appropriate abstraction, *e.g.*, a map or a set of pairs.

```
void redBlackTestHarness() {
  RedBlackTree a = create(0), b = create(1);
  a.replace(1, 1); b.replace(1, 10);
  a.replace(2, 2); b.replace(2, 20);
  assert(a.lookup(1) == 1 && a.lookup(42) == 0);
  assert(b.lookup(1) == 10 && b.lookup(42) == 1);
  a.remove(1); b.remove(2);
  assert(a.lookup(1) == 0 && a.lookup(42) == 0);
  assert(b.lookup(2) == 1 && b.lookup(42) == 1);
}
```

6.0 Verification Tasks

0. Prove the correctness of the test harness and tree implementation against the specifications of the tree (it is allowed to implement `remove()` by a call to `replace()`). *40 points*
1. Implement `remove()` so it really removes a node from the tree ([2], Section 13.4). *20 points*.
2. Prove the red-black balancing invariant, that is, that a red node cannot be a parent of another red node and that every path from a leaf to the root contains the same number of black nodes. *20 points*.

Like in the heap, the color fix-up operation is similar in spirit to the composite pattern. However here, the invariant involved is much more complex—the benchmark is designed to test scalability of the tool.

7 Conclusion and Future Work

The VACID-0 test provides a basis for comparing verification tools on implementations of non-trivial data structures. We hope that these and other solutions will lend themselves to interesting comparisons, and that they will help shape future editions of verification benchmarks.

References

0. Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.
1. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Invited paper.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, September 2001.
3. Robby (editor). Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008). Technical Report CS-TR-08-07, University of Central Florida, 2008.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, May 2002.
5. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *POPL*, pages 207–220. ACM, 2009.
6. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, LNCS. Springer, 2010. To appear.
7. K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In *VSTTE 2010*, LNCS. Springer, 2010. To appear.
8. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
9. James A. Storer. *An Introduction to Data Structures and Algorithms*. Birkhauser Boston, 2001.
10. Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008*, volume 5295 of *LNCS*, pages 84–98. Springer, October 2008.