The 2014 Verified Software Competition

N. Shankar (with Ernie Cohen, Marcelo Frias, and Peter Müller)

Computer Science Laboratory
SRI International
Menlo Park, CA

July 15, 2014

Overview

- Competitions are becoming popular as
  1. Buzz-generating events
  2. Long-running experiments to monitor progress
  3. Mechanisms to communicate success stories to non-experts
- Most competitions, e.g., CASC, SMT-COMP, SATLive, etc., measure performance of programs on a set of benchmark problems.
- The Verified Software Competition (VSComp) is unique in that it measures the effectiveness with which modern verification tools can be used to formalize and solve problems.
- Like a Formula I race that measures the effectiveness of the vehicle and the driver. (V. Klebanov)

History of VSComp

- First VSComp was organized with Peter Müller at VSTTE 2010 in Edinburgh.
- J-C. Filliâtre, Andrei Paskevich, and Aaron Stump organized one in 2011.
- The third one was organized by Joe Kiniry in 2013, but there was only one (partial) submission.
- At the Orlando WG 1.9 meeting, Peter Müller, Marcelo Frias, Ernie Cohen and I volunteered to organize the 2014 event.
- We decided to craft five problems spanning a range of challenges: algorithms, heaps, debugging, and concurrency.
- Need more discussion on what constitutes a useful competition.

The 2014 VSComp

- The competition took place during the weekend of June 14-15, 2014 starting at 9AM GMT on Saturday and ending at 9AM GMT on Monday.
- Teams could consist of at most three members.
- The test consisted of five verification problems that had to be solved with the aid of mechanized tools.
- Teams could employ multiple verification tools .
- The solutions were made available for peer review at the end of the competition.
- The competition was hosted at the website `vscomp.org`.

Problem 1: Patience Solitaire

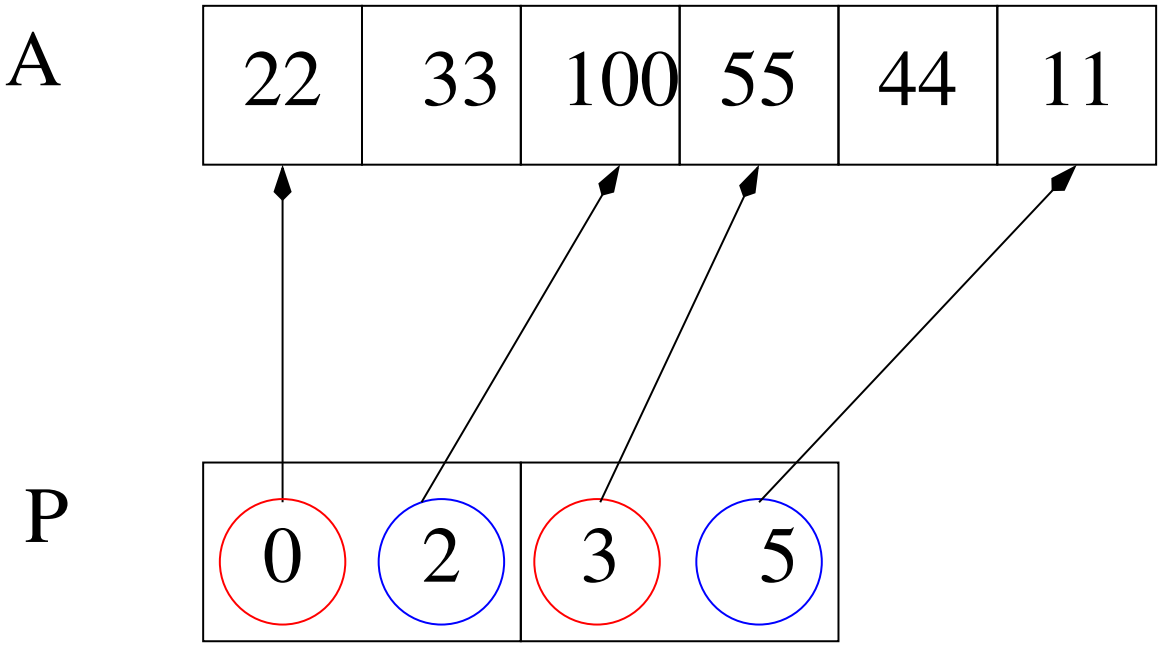
- Patience Solitaire is played by taking cards one-by-one from a deck of cards and arranging them face up in a sequence of stacks arranged from left to right.
- The very first card from the deck is kept face up to form a singleton stack.
- Each subsequent card is placed on the leftmost stack where its card value is no greater than the topmost card on that stack.
- If there is no such stack, then a new stack is started to right of the other stacks.

Patience Solitaire

- If the input sequence is 9, 7, 10, 9, 5, 4, and 10, then the stacks develop as
  $\langle [9] \rangle$
  $\langle [7, 9] \rangle$
  $\langle [7, 9], [10] \rangle$
  $\langle [7, 9], [9, 10] \rangle$
  $\langle [5, 7, 9], [9, 10] \rangle$
  $\langle [4, 5, 7, 9], [9, 10] \rangle$
  $\langle [4, 5, 7, 9], [9, 10], \langle 10 \rangle \rangle$
- Verify the claim is that the number of stacks at the end of the game is the length of the longest (strictly) increasing subsequence in the input sequence. In the above example, the subsequence is $7, 9, 10$, i.e., the second, fourth, and last elements of the sequence.

Problem 2: Partition Refinement

- A set S can be represented by an array A of distinct elements.
- A partition P of S splits S into a disjoint subsets $S_0, \ldots, S_{m-1}$.
- By rearranging the array elements in A, we can maintain each S_i as a set of contiguous elements of the array A, where each subset S_i is represented by a structure p_i that maintains *first* and *last* index in A for S_i.



Refining a Partition

- Now, given a subset X of S, we want to refine the partition $S_0, \ldots, S_{m-1}$ so that we refine each S_i into two sets $S_i \cap X$ and $S_i \setminus X$.
- The program should take the set S represented by A, the partition $S_0, \ldots, S_{m-1}$ represented as indicated below, and the set X, and return a representation of a new partition of S that refines each S_i into one or two nonempty subsets representing $S_i \cap X$ and $S_i \setminus X$.

Partition Refinement: State

The state of the algorithm consists of

1. The array A of size N represents a set S of positive integers. The elements in A can be rearranged as long as the set of integers is unchanged.
2. The input partitioning set X is given as a list of positive integers.
3. The partial map D represents an "inverse" of A so that $D(A(i)) = i$ for $0 \leq i < N$. It is needed to determine the position in array A of a given element of the S.
4. The partition P is an aggregate of intervals, e.g., a sequence, where each interval p contains fields *first* and *last* representing a contiguous segment of the array A. Each such partition p represents the subset $[\![p]\!]_A$ defined as $\{x \in S | \exists i : p.first \leq i \leq p.last : A(i) = x\}$. The algorithm also adds a field $p.count$ (with a default value of 0) that is used in implementing the refinement algorithm.
5. A map F from the indices of A to partitions in P capturing the relationship that index i is in partition $p = P(F(i))$, i.e., $p.first \leq i \leq p.last$.

The Partition Refinement Algorithm

- *refine*(A, D, P, F, X): First, successively invoke *refineOne*(A, D, P, F, x) over each x in X, then apply *makeNewPartitions*(P, F) to the structures P and F returned by the first phase to spawn new intervals from the ones that have been refined.

```
refineOne(A, D, P, F, x){
  let i = D(x),
      p = F(i),
      j = p.first + p.count
   in if (i >= j){
        swap(A(i), A(j));
        D(A(i)) := i;
        D(A(j)) := j;
        p.count++;
      }
}
```

Create New Partitions

```
makeNewPartitions(P, F){
    for (j = 0; j < NumPartitions; j++){
     let p = P(j)
      in if (p.count > 0 &&
              p.count < p.last - p.first + 1){
        newp := new Partition;
        newp.first := p.first + p.count;
        newp.last := p.last;
        newp.count := 0;
        P(NumPartitions) := newp;
        p.last := p.first + p.count - 1;
        p.count := 0;
        for (i=newp.first; i <= newp.last; i++){
          F(i) := NumPartitititions;
        };
        NumPartitions++;
      }
    }
}
```

Example

For example, if

1. A is $\langle 0 \mapsto 22, 1 \mapsto 33, 2 \mapsto 100, 3 \mapsto 55, 4 \mapsto 44, 5 \mapsto 11 \rangle$
2. D is the inverse of A and is $\langle 100 \mapsto 2, 11 \mapsto 5, 22 \mapsto 0, 33 \mapsto 1, 44 \mapsto 4, 55 \mapsto 3 \rangle$
3. P is $\langle 0 \mapsto \langle \mathit{first} \mapsto 0, \mathit{last} \mapsto 2 \rangle, 1 \mapsto \langle \mathit{first} \mapsto 3, \mathit{last} \mapsto 5 \rangle \rangle$
4. F is $\langle 0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1 \rangle$
5. X, the input, is $\{11, 22, 44\}$.

Processing each element of the input X yields a state where A is

$$\langle 0 \mapsto 22, 1 \mapsto 11, 2 \mapsto 55, 3 \mapsto 33, 4 \mapsto 100, 5 \mapsto 44 \rangle,$$

D is $\langle 100 \mapsto 4, 11 \mapsto 1, 22 \mapsto 0, 33 \mapsto 3, 44 \mapsto 5, 55 \mapsto 2 \rangle$, P is

$$\left\langle \begin{array}{l} 0 \mapsto \langle \mathit{first} \mapsto 0, \mathit{last} \mapsto 0 \rangle, \\ 1 \mapsto \langle \mathit{first} \mapsto 1, \mathit{last} \mapsto 2 \rangle, \\ 2 \mapsto \langle \mathit{first} \mapsto 3, \mathit{last} \mapsto 4 \rangle, \\ 3 \mapsto \langle \mathit{first} \mapsto 5, \mathit{last} \mapsto 5 \rangle \end{array} \right\rangle, \text{ and}$$

F is $\langle 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 2, 5 \mapsto 3 \rangle$.

Verification Tasks

1. The algorithm always terminates returning an output state A', D', P', F'.
2. The output partition given by A', D', P', F' is a refinement of the input partition A, D, P, F so that
   1. For any $i < N$, $P'(F'(i))$ is an interval containing i.
   2. For any p' in P', there exists a p in P such that $[\![p']\!]_{A'} = [\![p]\!]_A \cap X$ or $[\![p']\!]_{A'} = [\![p]\!]_A \setminus X$

Problem 3: A Lock-free Log-based Set Algorithm

Given an implementation of lock-free set maintenance for `ThreadCnt` number of threads.
Data structures are

```
atomic<int>      log[size];    //log of operations
atomic<size_t> gc, tl, hd; //initially all zero
atomic<size_t> ht[threadCnt]; //initially all size
```

`log` maintains the set with entries `e` (for adding `e` to the set), 0 (for unused entry), `-e` (for deleting `e`).
`hd` is the latest entry visible to readers.
`gc` is the head of the entries that have been garbage collected.
`tl` is the head of entries that can be marked as redundant.
`ht[t]` is the index where thread `t` blocks the garbage collector.
`me()` is the identity of the calling thread.

Lock-free operations

```
#define doReturn(v)        { ht[me()] = size; return v; }
#define advance(loc,var)   { local = var; ht[me()] = local
#define grab(local, var)   { advance(local,var);
                             advance(local,var); }

void update(int val) {  //atomically add or remove
                        //abs(val) from the set
    size_t h;
    grab(h, hd)
    while (true) {
        bool success = !cmpXchg<int>(log+h, 0, val);
        cmpXchg<size_t>(&hd, h, h+1);
        if (success) doReturn();
        advance(h,hd);
    }
}
```

More Operations

```
bool lookup(int val) { // atomically check if positive
                       // int val is in the set
    size_t t,i;
    int x;
    grab(t,tl);
    for (i = hd; i != t && abs(x = h[i-1]) != val; i--);
    doReturn(i != t && 0 < x);
}

void collect() { // try to collect garbage; semantic no-
    size_t t = tl;
    for (size_t i = 0; i < threadCnt; i++)
        t = min(t,ht[i]);
    size_t g = gc;
    if (g < t) cmpXchg<size_t>(&gc,g,t);
}
```

Proof Obligations

- Prove the program is memory-safe, i.e. no thread accesses log entries below gc.
- Define, for every state, the abstract state of the set in that state. (You can do this using ghost variables or using an abstraction map, or any other reasonable method.) This abstract value should be consistent with the obvious intentions of the problem.
- Prove that lookup is linearizable, i.e. that if it returns true/false, val was/(was not) in the abstract set at some point between when the call started and when it returned.

Problem 4: Graph Cloning

Given graph G with nodes represented as heap-allocated objects, each storing a natural number.
Each node contains a map from labels l to references (pointers) to nodes.
G contains an edge from n_1 to n_2 with label l if and only if n_1's map maps l to n_2 as shown

```
public class Node {
    Map<Integer, Node> edges;
    int value;  // a natural number

    public Node(int value) {
        edges = new HashMap<Integer, Node>();
        this.value = value;
    }
    ...
}
```

Tasks

Implement a copy method that takes a node and copies the graph structure reachable from that node.
Prove the following properties of your copy method:

1. Provided that the argument node is non-null, the method will not cause a run-time errors including null-pointer dereferencing and precondition violations.
2. The method terminates for all non-null inputs.
3. The result of the method and all objects reachable from it are fresh, that is, are allocated and have not been allocated in the pre-state of the method.
4. The method has no observable side effects, that is, does not modify or de-allocate any object that was allocated in the pre-state.
5. For any valid call to n.copy(), the subgraph reachable from the n before the call is isomorphic to the subgraph reachable from the result after the call.

More tasks

Implement an eval method that takes a non-null node and a finite sequence of integers (labels) and returns an integer so that if there are suitable edges for all labels in the sequence, eval returns the natural number stored in the final node; otherwise, it returns -1. You should prove the following properties of your eval method:

1. Provided that the arguments are non-null, the method will not cause a run-time error (as defined in the previous task).
2. The method terminates for all non-null inputs (assuming the label sequence is finite).
3. The method satisfies the functional specification given above.
4. For all non-null nodes n and sequences path, n.eval(path) and n.copy().eval(path) yield the same result.

Problem 5: Binomial Heaps

Given classes `BinomialHeap` and `BinomialHeapNode` modeling binomial heaps and defining a faulty `extractMin` that does not preserve the class invariant

1. Find and describe the above-mentioned fault.
2. Provide an input binomial heap that exercises this fault, and describe the technique used to find the input (automated techniques are preferred to human inspection).
3. Provide a correct version of method `extractMin`.
4. Provide a suitable class invariant.
5. Verify that method `extractMin` indeed preserves the provided class invariant or at least a meaningful subset of properties from the class invariant.

Teams

- Cha-cha-cha (Maria Christakis, Rustan Leino, Dan Rosen + Dafny): 1, 2, 4
- IncreMentally Challenged (Nadia Polikarpova, Julian Tschannen, Scott West + Dafny): 1, 2, 4
- ProofInUse (David Mentre, Claude Marche, Yannick Moy + Why3 + Spark 2014): 1, 2
- SPARK 2014 (Johannes Kanig, Julien Thierry, Zhi Zhang + Coq + Spark 2014): 1, 2
- Team KIV (Gidon Ernst, Jrg Pfhler, Bogdan Tofan + KIF): 1, 2, 4
- VerCors (Stefan Blom, Saeed Darbari + Blom's verification tool): 1
- VeriFast (Bart Jacobs + VeriFast): 3, 4
- Viorel Prioteasa (Viorel Proteasa + Isabelle): 2, 4