Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2013,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

VerifyThis Verification Competition 2012
- Organizer´s Report -

Marieke Huisman, Vladimir Klebanov,
and Rosemary Monahan

2013

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

VerifyThis Verification Competition 2012 —Organizer's Report—

Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan

fm2012.verifythis.org

Abstract. This is an informal report on the VerifyThis 2012 verification competition compiled by its organizers. It describes the general set up, the challenges and the results of the competition. More in-depth publications will follow in a special section of the STTT journal.

1 Introduction

VerifyThis was a two-day event taking place as part of the Symposium on Formal Methods (FM 2012) on August 30-31 in Paris, France. It was a successor of the program verification competition held at FoVeOOS 2011.

The aims of the competition are:

- to bring together those interested in formal verification, and to provide an engaging, hands-on, and fun opportunity for discussion
- to evaluate the usability of logic-based program verification tools in a controlled experiment that could be easily repeated by others.

The competition offered three challenges presented in natural language. Participants had to formalize the requirements, implement a solution, and formally verify the implementation for adherence to the specification. Submissions were judged for correctness, completeness and elegance by the organizers. As a first, the competition included a post-mortem session where participants answered the questions of the judges. In parallel, the participants used this session to discuss details of the problems and solutions among each other.

Teams of up to two people physically present on site could participate. Particularly encouraged was participation of:

- student teams (this includes PhD students)
- non-developer teams using a tool someone else developed
- several teams using the same tool.

The competition website can be found at `http://fm2012.verifythis.org/`. More background information on the competition format and the choices made is available in [4]. Reports from previous competitions of similar nature can be found in [6, 2, 3].

1.1 Participants

Participating teams in no particular order:

1. Bart Jacobs, Jan Smans (VeriFast)
2. Jean-Christophe Filliâtre, Andrei Paskevich (Why3)
3. Yannick Moy (GNATprove)
4. Wojciech Mostowski, Daniel Bruns (KeY)
5. Valentin Wüstholz, Maria Christakis (Dafny) (student, non-developer team)
6. Gidon Ernst, Jörg Pfähler (KIV) (student team)
7. Stefan Blom, Tom van Dijk (ESC/Java2) (non-developer team)
8. Zheng Cheng, Marie Farrell (Dafny) (student, non-developer team)
9. Claude Marché, François Bobot (Why3)
10. Ernie Cohen (VCC)
11. Nguyen Truong Khanh (Pat)

2 Challenge 1: Longest Common Prefix (LCP, 45 minutes)

2.1 Verification Task

Longest Common Prefix (LCP) is a problem in text querying [7]. In the following, we model text as an integer array, but it is perfectly admissible to use other representations (e.g., Java Strings), if a verification system supports them. LCP can be informally specified as follows:

- Input: an integer array `a`, and two indices `x` and `y` into this array
- Output: length of the longest common prefix of the subarrays of `a` starting at `x` and `y` respectively.

A reference implementation of LCP is given by the pseudocode below. Prove that your implementation complies with a formalized version of the above specification.

```
int lcp(int[] a, int x, int y) {
    int l = 0;
    while (x+l<a.length && y+l<a.length && a[x+l]==a[y+l]) {
        l++;
    }
    return l;
}
```

2.2 Results and Organizer Comments

As expected, the LCP challenge did not pose a difficulty. Eleven submissions were received, of which eight were judged as sufficiently correct and complete. Two submissions failed to specify the maximality of the result, while one submission had further adequacy problems.

We found the common prefix property was best expressed in Dafny syntax

```
a[x..x+l] == a[y..y+l]
```

which eliminated a lot of quantifier verbosity. The maximality was typically expressed by one of many variations of

```
x+l == a.length || y+l == a.length || a[x+l] != a[y+l]
```

Jean-Christophe Filliâtre and Andrei Paskevich (Why3 team) have also proved an explicit lemma that no greater result (i.e., longer common prefix) exists. This constituted the most general and close to the text specification.

2.3 Advanced Verification Tasks

For those who have completed the LCP challenge quickly, the description included a further challenge outlined below. No submissions attempting to solve the advanced challenge were received during the competition though.

Background Together with a suffix array, LCP can be used to solve interesting text problems, such as finding the longest repeated substring (LRS) in a text.

In its most basic form, a suffix array (for a given text) is an array of all suffixes of the text. For the text [7,8,8,6], the basic suffix array is

```
[[7,8,8,6],
  [8,8,6],
    [8,6],
      [6]]
```

Typically, the suffixes are not stored explicitly as above but represented as pointers into the original text. The suffixes in a suffix array are also sorted in lexicographical order. This way, occurrences of repeated substrings in the original text are neighbors in the suffix array.

For the above example (assuming pointers are 0-based integers), the sorted suffix array is: [3,0,2,1].

Verification Task The attached Java code[1] contains an implementation of a suffix array (SuffixArray.java), consisting essentially of a lexicographical comparison on arrays, a sorting routine, and LCP.

The client code (LRS.java) uses these to solve the LRS problem. Verify that it does so correctly.

Yet More Advanced Verification Tasks Those seeking a challenge might wish to verify one of the more advanced (w.r.t. performance) suffix array implementations, such as [5].

[1] Available as part of the original challenge description at `fm2012.verifythis.org` and in the appendix of this report.

3 Challenge 2: Prefix Sum (PrefixSum, 90 minutes)

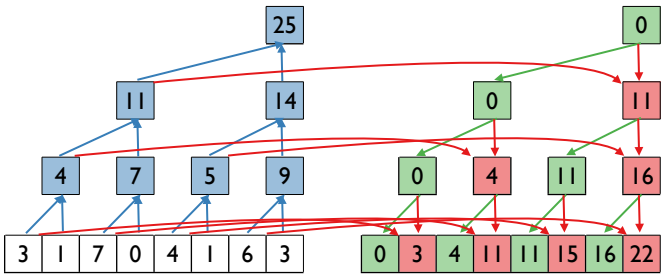

Fig. 1. Upsweep and downsweep phases of the prefix sum calculation. The organizers would like to thank Nathan Chong for the illustration.

3.1 Background

The concept of a prefix sum is very simple. Given an integer array `a`, store in each cell `a[i]` the value `a[0]+...+a[i-1]`.

Example 1. The prefix sum of the array [3, 1, 7, 0, 4, 1, 6, 3] is [0, 3, 4, 11, 11, 15, 16, 22].

Prefix sums have important applications in parallel vector programming, where the workload of calculating the sum is distributed over several processes. A detailed account of prefix sums and their applications is given in [1]. We will verify a sequentialized version of a prefix sum calculation algorithm.

3.2 Algorithm Description

We assume that the length of the array is a power of two. This allows us to identify the array initially with the leaves of a complete binary tree. The computation proceeds along this tree in two phases: upsweep and downsweep.

During the upsweep, which itself proceeds in phases, the sum of the children nodes is propagated to the parent nodes along the tree. A part of the array is overwritten with values stored in the inner nodes of the tree in this process (Figure 3, left[2]). After the upsweep, the rightmost array cell is identified with the root of the tree.

As preparation for the downsweep, a zero is inserted in the rightmost cell.

Then, in each step, each node at the current level passes to its left child its own value, and it passes to its right child, the sum of the left child from the upsweep phase and its own value (Figure 3, right).

[2] The original challenge description contained an illustrating excerpt from a slide deck on prefix sums. The organizers would like to thank Nathan Chong for it.

3.3 Verification Task

We provide an iterative and a recursive implementation of the algorithm. You may choose one of these to your liking.

1. Specify and verify the upsweep method. You can begin with a slightly simpler requirement that the last array cell contains the sum of the whole array in the post-state.
2. Verify both upsweep AND downsweep—prove that the array cells contain appropriate prefix sums in the post-state.

If a general specification is not possible with your tool, assume the length of array is 8.

3.4 Results and Organizer Comments

Eight submissions were received, of which one (by the VeriFast team) was judged as sufficiently correct and complete. Though the upsweep and downsweep algorithm were not complex, it was challenging to build a mental model of what is happening. The GNATprove team noted that the ability of the tool to test the requirement and auxiliary annotations by translating them to run-time checks was helpful in this challenge. We found this observation compelling.

The only "technical" problem in this challenge was reasoning about powers of two. KIV and Why3 already included a formalization of the exponentiation operator. The GNATprove team was the only one to make use of the bounded array simplification proposed in the challenge description. It was also the only team that has chosen to verify the iterative version of the algorithm and not the recursive one during the competition.

3.5 Advanced Verification Tasks

A verification system supporting concurrency could be used to verify a parallel algorithm for prefix sum computation [1].

4 Challenge 3: Iterative Deletion in a Binary Search Tree (TreeDel, 90 minutes)

4.1 Verification Task

Given: a pointer `t` to the root of a non-empty binary search tree (not necessarily balanced). Verify that the following procedure removes the node with the minimal key from the tree. After removal, the data structure should again be a binary search tree.

```
(Tree, int) search_tree_delete_min (Tree t) {
   Tree tt, pp, p;
   int m;
   p = t->left;
   if (p == NULL) {
       m = t->data; tt = t->right; dispose(t); t = tt;
   } else {
       pp = t; tt = p->left;
       while (tt != NULL) {
           pp = p; p = tt; tt = p->left;
       }
       m = p->data; tt = p->right; dispose(p); pp->left= tt;
   }
   return (t,m);
}
```

Note: When implementing in a garbage-collected language, the call to `dispose()` is superfluous.

4.2 Organizer Comments

This problem has appeared in [8] as an example of an iterative algorithm that becomes much easier to reason about when reimplemented recursively.

The difficulty stems from the fact that the loop invariant has to talk about a complicated "tree with a hole" data structure, while the recursion-based specification can concentrate on the data structure still to be traversed, which in this case is also a tree. A solution proposed in [8] is that of a *block contract*, i.e., a pre/post-style contract for arbitrary code blocks. A block contract enables recursion-style forward reasoning about loops and other code without explicit code transformation.

Only the VeriFast team submitted a working solution to this challenge within the allotted time. The KIV team submitted a working solution about 20 minutes after the deadline.

Incidentally, the VeriFast tool implements block contracts, though only for loops and not for arbitrary code blocks as would be necessary here. During the competition, the VeriFast team instead built a solution based on (an encoding of) a "magic wand" operator of separation logic. The block contracts are presented in [8] as a much simpler alternative to this kind of approach.

The solution of the KIV team involved induction over the number of loop iterations, which is a form of "forward reasoning" in essence very similar to block contracts. In KIV's case there is no explicit block contract annotation though, but the user interactively supplies a corresponding induction hypothesis during the proof process. Unsurprisingly, the induction encompasses the while loop and the following code line that restores the data structure to its tree form.

5 Results, Statistics, and Remarks

5.1 Awarded Prizes and Statistics

The main results of the competition are as follows:

- Best team: Bart Jacobs, Jan Smans (VeriFast)
- Best student team: Gidon Ernst, Jörg Pfähler (KIV)
- Distinguished user-assistance tool feature: integration of proving and run-time assertion checking in GNATprove (team member: Yannick Moy)
- Tool used by most teams: prize shared between Dafny and Why3
- Best (pre-competition) problem submission: "Optimal Replay" by Ernie Cohen

Statistics per challenge:

- LCP: 11 submissions were received, of which 8 were judged as correct and complete and two as correct but partial solutions.
- PrefixSum: 8 submissions were received, of which one was judged correct and complete.
- TreeDel: 7 submissions were received, of which one was judged correct and complete.

5.2 Post-mortem Session

The post-mortem session the day after the competition was much appreciated both by the judges and by the participants. It was very helpful for the judges to be able to ask the teams questions in order to better understand and appreciate their submissions. At the same time, the other participants were having a lively discussion about the challenges, presenting their solutions to each other and exchanging ideas and comments with great enthusiasm. We would recommend such a post-mortem session for any on-site verification competition.

5.3 Session Recording

The artifacts produced and submitted by the teams during the competition only tell half of the story. The *process* of arriving at a solution is just as important. The organizers have for some time already planned to record and analyze this process (on voluntary basis). The recording would give insight into the pragmatics of different verification systems and allow the participants to learn more from the experience of others.

Finding the right technical solution to capture a video or a sequence of screenshots of the user's screen has turned out to be a challenge. The requirements for the recording software are:

- availability on different platforms, ease of installation, robustness
- reasonable quality of the recording and (it's trade-off) space requirements. The recorded session can be several hours long

– low load on the CPU of the recorded machine.

The majority of tools that the organizers have screened are targeted towards producing demonstrations or screencasts and require CPU-intensive video encoding.

A promising approach was to install a VNC server on the target machine and offload the recording to another machine over the network. Limited tests of this setup have been unsuccessful though, as the wireless network at the conference venue did not allow direct connections between machines of the participants. This is apparently not uncommon.

The organizers are welcoming suggestions in this matter.

5.4 Final Remarks

The VerifyThis 2012 challenges have offered a substantial degree of complexity and difficulty. The competition has also demonstrated the importance of strategy. Starting with a simplified version of the challenge and adding complexity gradually is often more efficient than attacking the full challenge at once.

Acknowledgments The organizers would like to thank Rustan Leino, Nadia Polikarpova, and Mattias Ulbrich for their feedback and support prior to the competition.

References

1. G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
2. T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J.-C. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The COST IC0701 verification competition 2011. In B. Beckert, F. Damiani, and D. Gurov, editors, *International Conference on Formal Verification of Object-Oriented Systems (FoVeOOS 2011)*, LNCS. Springer, 2012.
3. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd Verified Software Competition: Experience report. In V. Klebanov, A. Biere, B. Beckert, and G. Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, 2012.
4. M. Huisman, V. Klebanov, and R. Monahan. On the organisation of program verification competitions. In V. Klebanov, B. Beckert, A. Biere, and G. Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE), Manchester, UK, June 30, 2012*, volume 873 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.
5. J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

6. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st Verified Software Competition: Experience report. In M. Butler and W. Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *LNCS*. Springer, 2011.
7. R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.
8. T. Tuerk. Local reasoning about while-loops. In P. Müller, D. Naumann, and H. Yang, editors, *Proceedings, VS-Theory Workshop of VSTTE 2010*, pages 29–39, 2010.

A LCP/LRS Source Code

```
1  public class SuffixArray {
2
3      private final int[] a;
4      private final int[] suffixes;
5      private final int N;
6
7      public SuffixArray(int[] a) {
8          this.a = a;
9          N = a.length;
10         suffixes = new int[N];
11         for (int i = 0; i < N; i++) suffixes[i] = i;
12         sort(suffixes);
13     }
14
15
16     public int select(int i) {
17         return suffixes[i];
18     }
19
20
21     private int lcp(int x, int y) {
22         int l = 0;
23         while (x+l<N && y+l<N && a[x+l]==a[y+l]) {
24             l++;
25         }
26         return l;
27     }
28
29
30     public int lcp(int i) {
31         return lcp(suffixes[i], suffixes[i-1]);
32     }
33
34
35     public int compare(int x, int y) {
36         if (x == y) return 0;
37         int l = 0;
38
39         while (x+l<N && y+l<N && a[x+l] == a[y+l]) {
40             l++;
41         }
42
43         if (x+l == N) return -1;
44         if (y+l == N) return 1;
45         if (a[x+l] < a[y+l]) return -1;
46         if (a[x+l] > a[y+l]) return 1;
47
48         throw new RuntimeException();
49     }
```

```
50
51
52      public void sort(final int[] data) {
53          for(int i = 0; i < data.length + 0; i++) {
54              for(int j = i;
55                  j > 0 && compare(data[j-1], data[j]) > 0; j--) {
56                  final int b = j - 1;
57                  final int t = data[j];
58                  data[j] = data[b];
59                  data[b] = t;
60              }
61          }
62      }
63
64
65      public static void main(String[] argv) {
66          int[] arr = {1,2,2,5};
67          SuffixArray sa = new SuffixArray(arr);
68          System.out.println(sa.lcp(1,2));
69          int[] brr = {1,2,3,5};
70          sa = new SuffixArray(brr);
71          System.out.println(sa.lcp(1,2));
72          int[] crr = {1,2,3,5};
73          sa = new SuffixArray(crr);
74          System.out.println(sa.lcp(2,3));
75          int[] drr = {1,2,3,3};
76          sa = new SuffixArray(drr);
77          System.out.println(sa.lcp(2,3));
78      }
79
80
81  }
82
83  //Based on code by Robert Sedgewick and Kevin Wayne.
```

```
1   public class LRS {
2
3       private static int solStart = 0;
4       private static int solLength = 0;
5       private static int[] a;
6
7       public static void main(String[] args) {
8           a = new int[args.length];
9           for (int i=0; i<args.length; i++) {
10              a[i]=Integer.parseInt(args[i]);
11          }
12          doLRS();
13          System.out.println(solStart+"->"+solLength);
14      }
15
16
17
18      public static void doLRS() {
19          SuffixArray sa = new SuffixArray(a);
20
21          for (int i=1; i < a.length; i++) {
22              int length = sa.lcp(i);
23              if (length > solLength) {
24                  solStart = sa.select(i);
25                  solLength = length;
26              }
27          }
28      }
29
30  }
31
32  //Based on code by Robert Sedgewick and Kevin Wayne.
```

B PrefixSum Source Code

Recursive Version

```
1  import java.util.Arrays;
2
3  class PrefixSumRec {
4
5      private int[] a;
6
7      PrefixSumRec(int[] a) {
8          this.a = a;
9      }
10
11
12     public void upsweep(int left, int right) {
13         if (right > left+1) {
14             int space = right - left;
15             upsweep(left-space/2,left);
16             upsweep(right-space/2,right);
17         }
18         a[right] = a[left]+a[right];
19     }
20
21
22     public void downsweep(int left, int right) {
23         int tmp = a[right];
24         a[right] = a[right] + a[left];
25         a[left] = tmp;
26         if (right > left+1) {
27             int space = right - left;
28             downsweep(left-space/2,left);
29             downsweep(right-space/2,right);
30         }
31
32     }
33
34
35     public static void main (String[] args) {
36         int[] a = {3,1,7,0,4,1,6,3};
37         PrefixSumRec p = new PrefixSumRec(a);
38         System.out.println(Arrays.toString(a));
39         p.upsweep(3,7);
40         System.out.println(Arrays.toString(a));
41         a[7] = 0;
42         p.downsweep(3,7);
43         System.out.println(Arrays.toString(a));
44     }
45
46 }
47
48
49 /*
50 [3, 1, 7, 0, 4, 1, 6, 3]
51 [3, 4, 7, 11, 4, 5, 6, 25]
52 [0, 3, 4, 11, 11, 15, 16, 22]
53 */
```

Iterative Version

```
1  import java.util.Arrays;
2
3  class PrefixSumIter {
4
5      private int[] a;
6
7      PrefixSumIter(int[] a) {
8          this.a = a;
9      }
10
11
12     public int upsweep() {
13         int space = 1;
14         for (; space < a.length; space=space*2) {
15             int left = space - 1;
16             while (left < a.length) {
17                 int right = left + space;
18                 a[right] = a[left] + a[right];
19                 left = left + space*2;
20             }
21         }
22         return space;
23     }
24
25
26     public void downsweep(int space) {
27         a[a.length - 1] = 0;
28         space = space/2;
29         for (; space > 0; space=space/2) {
30             int right = space*2 - 1;
31             while (right < a.length) {
32                 int left = right - space;
33                 int temp = a[right];
34                 a[right] = a[left] + a[right];
35                 a[left] = temp;
36                 right = right + space*2;
37             }
38         }
39     }
40
41
42     public static void main (String[] args) {
43         int[] a = {3,1,7,0,4,1,6,3};
44         PrefixSumIter p = new PrefixSumIter(a);
45         System.out.println(Arrays.toString(a));
46         int space = p.upsweep();
47         System.out.println(space);
48         System.out.println(Arrays.toString(a));
49         p.downsweep(space);
50         System.out.println(Arrays.toString(a));
51     }
52
53 }
54
55
56 /*
57 [3, 1, 7, 0, 4, 1, 6, 3]
58 [3, 4, 7, 11, 4, 5, 6, 25]
59 [0, 3, 4, 11, 11, 15, 16, 22]
60 */
```