The 2nd Verified Software Competition: Experience Report

Jean-Christophe Filliâtre[1,2], Andrei Paskevich[1,2], and Aaron Stump[3]

[1] LRI, Univ Paris-Sud, CNRS, Orsay F-91405
[2] INRIA Saclay-Île-de-France, ProVal, Orsay F-91893
[3] The University of Iowa, Iowa City, Iowa 52242

Abstract. We report on the second verified software competition. It was organized by the three authors on a 48 hours period on November 8–10, 2011. This paper describes the competition, presents the five problems that were proposed to the participants, and gives an overview of the solutions sent by the 29 teams that entered the competition.

1 Introduction

Competitions for high-performance logic-solving tools have played a prominent role in Computational Logic in the past decade or so. Well-established examples include CASC (the competition for first-order provers) [17], the SAT Competition and SAT Race [12], and SMT-COMP (the Satisfiability Modulo Theories Competition) [2]. Many other logic-solving fields have also developed competitions in recent years, with new ones coming online yearly. The reason for this interest is that competitions provide a high-profile opportunity for comparative evaluation of tools. Competitions are good for competitors, who have a chance to increase the visibility of their work by participation in the competition. This is true especially, but not at all exclusively, if they do well. Competitions are good for the field, since they attract attention from a broader audience than might otherwise be following research progress in that area. Finally, competitions are good for potential users of logic-solving tools, who have a chance to see which tools are available and which seem to be doing well on various classes of problems.

Competitions have recently been introduced in the field of deductive software verification. The 1st Verified Software Competition was held in 2010, affiliated with the "Verified Software: Theories Tools and Experiments" (VSTTE) conference [10]. A similar recent event was the COST IC0701 Verification Competition, held in 2011 [5]. These two competitions were quite alike in spirit and organization: The objective was to verify behavioral correctness of several algorithms in a limited amount of time (30–90 minutes per problem), and the participants were free to choose the language of specification and implementation. Another contest of verification tools was the SV-COMP 2012 competition [3], dedicated specifically to fully automated reachability analysis of C programs.

In this paper, we describe the 2nd Verified Software Competition, affiliated with VSTTE 2012. The purposes of this competition were: to help promote approaches and tools, to provide new verification benchmarks, to stimulate further development of verification techniques, and to have fun [1]. In the end, 29 teams competed to solve 5 problems designed by the organizers over a period of three days in early November, 2011. Their solutions, written using a total of 22 different tools, were then judged by the organizers, over the course of the following month. We chose not to make a total ranking list public, but instead identified 6 teams earning gold, silver, or bronze medals. The medalists were announced at VSTTE 2012 (Philadelphia, January 28, 2012) and the 2 gold-medalist teams were each given a 15-minute slot at the conference to present their solution.

The rest of this paper is organized as follows. Section 2 gives an overview of the competition. Then Section 3 describes the five problems and, for each, discusses the solutions that we received. Finally, Section 4 lists the winners and draws some lessons from this competition.

2 Competition Overview

2.1 Schedule

We have chosen to hold the competition over a 48 hours period in order to include more challenging problems than it was feasible for an on-site competition. One source of inspiration for us was the annual ICFP programming contest.

The competition was organized during Fall 2011, with the following schedule:

- Sep 30/Oct 7/Nov 1: the competition is announced on various mailing lists;
- Nov 8, 15:00 UTC: the competition starts (problems are put on the web);
- Nov 10, 15:00 UTC: the competition ends (solutions are sent by email);
- Dec 12: winners are notified privately;
- Jan 28: medalists are announced at VSTTE 2012 and other participants are sent their score and rank privately.

2.2 Technical Organization

The competition website [1] was hosted as a subpage of the VSTTE 2012 website. A Google group was created for the organizers to make announcements, and for the participants to ask questions prior and during the competition and to make their solutions public after the competition if they wish. An email address was created for the participants to submit their solutions.

2.3 Rules

The full description of the competition is available at the competition website [1]. The main rules were the following:

- team work is allowed, but only teams up to 4 members are eligible for the first prize;

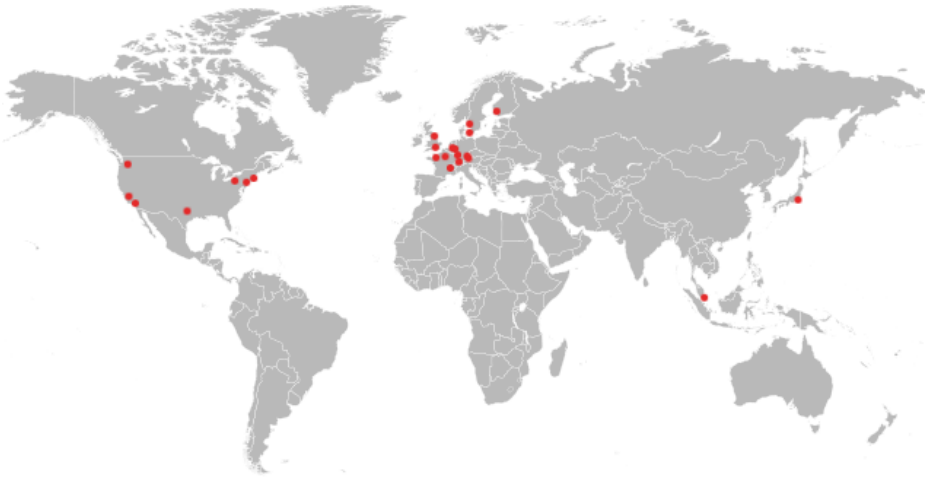

Fig. 1. Geographical distribution of the participants.

- any software used in the solutions should be freely available for noncommercial use to the public;
- software must be usable on x86 Linux or Windows;
- participants can modify their tools during the competition.

2.4 Participants

The competition gathered 79 participants, grouped in 29 teams as follows:

- 8 teams of size 1,
- 6 teams of size 2,
- 4 teams of size 3,
- 10 teams of size 4,
- 1 team of size 9.

Participants were from North America, Europe, and Asia, as depicted in Fig. 1. Participants used the following systems, in alphabetic order (the number between parentheses indicates how many teams used that system): ACL2 (1), Agda (3), ATS (1), B (2), BLAST (1), CBMC (1), Coq (7), Dafny (6), Escher (1), Guru (1), HIP (1), Holfoot (1), Isabelle (2), KeY (1), KIV (1), PAT (1), PML (1), PVS (3), Socos (1), VCC (2), VeriFast (1), Ynot (1). 9 teams used at least two different systems.

2.5 Evaluation Process

We approached the rather daunting task of hand-evaluating the solutions as follows. Our goal was to determine the medalists first, and then go back and provide

additional feedback as time permitted to all competitors. So we first triaged the solutions by determining their maximum possible score; that is, maximal points each could possibly earn, based on the parts of the problems it claimed to have solved. For example, only 5 teams claimed to have solved all parts of all the problems. We then heuristically started with the top 8 teams, and each of the organizers closely evaluated their solutions.

The evaluation process was as follows:

1. Read the submission to check that the formal specification conforms to the problem statement.
2. Check the formal proof:
   (a) Run the tool on the input files and check the output (proof found or correctly replayed).
   (b) Manually introduce errors in code or specification (for example, extend loop bounds or weaken a precondition) and rerun the tool to check that input files are not accepted anymore.

As a result of this evaluation, each task was attributed a fraction of its total "worth". We subtracted points based on our assessment that the solution for some part of a problem fell short in some way.

Despite some points being taken away, we were already able to determine our medalists from the first-chosen 8 teams, since no other team had a maximum possible score that could possibly beat the actual score we assessed for the bronze medalists. After we determined the winners, we then divided the remaining solutions among ourselves, to provide written feedback to each team on its solution. Each team was emailed the comments for its solution, once all had been inspected by at least one organizer.

3 Problems and Submissions

Prior to the competition, we prepared 8 problems of various difficulties. Each problem was solved using Why3 [4] and was independently tested by our colleagues Claude Marché and Duckki Oe. Finally, we picked up the following 5 problems:

1. Two-Way Sort (50 points) — sort an array of Boolean values
2. Combinators (100 points) — call-by-value reduction of SK-terms
3. Ring Buffer (150 points) — queue data structure in a circular array
4. Tree Reconstruction (150 points) — build a tree from a list of leaf depths
5. Breadth-First Search (150 points) — shortest path in a directed graph

We selected the problems and assigned the scores according to the time we ourselves and the testers spent devising solutions to them. The three problems we did not include were: Booth's multiplication algorithm, permutation inverse in place [11, p. 176], and counting sort. The former two problems seemed too difficult for a 48-hour competition and the third one was too similar to problems

	2-way sort	SK comb.	ring buffer	tree rec.	BFS
at least one task attempted	25	21	20	28	19
all tasks attempted	20	19	19	17	12
perfect solution	19	10	15	12	9

Fig. 2. Solutions overview (over a total of 29 solutions).

```
two_way_sort(a: array of boolean) :=
  i <- 0;
  j <- length(a) - 1;
  while i <= j do
    if not a[i] then
      i <- i+1
    elseif a[j] then
      j <- j-1
    else
      swap(a, i, j);
      i <- i+1;
      j <- j-1
    endif
  endwhile
```

Fig. 3. Problem 1: Two-Way Sort.

1 and 3. In hindsight, we could have chosen more difficult problems, as the top-ranked participants completed all tasks without much trouble.

Each problem consists of a program to be verified, according to a set of verification tasks ranging from mere safety and termination to full behavioral correctness. A pseudo-code with an imperative flavor is used to describe the programs, but the participants were free to turn them into the language of their choice, including purely applicative languages.

Figure 2 gives an overview of the solutions we received, on a per-problem basis. The remainder of this section describes the problems in detail and provides some feedback on the solutions proposed by the participants.

3.1 Problem 1: Two-Way Sort

The first problem we considered to be easy, and was supposed to be a warm-up exercise for the participants. A pseudo-code to sort an array of Boolean values is given (see Fig. 3). It simply scans the array from left to right with index `i` and from right to left with index `j`, swapping values `a[i]` and `a[j]` when necessary. The verification task are the following:

1. *Safety:* Verify that every array access is made within bounds.
2. *Termination:* Prove that function `two_way_sort` always terminates.
3. *Behavior:* Verify that after execution of function `two_way_sort`, the following properties hold:

$$
\begin{array}{rl}
\textit{terms} & t ::= \mathsf{S} \mid \mathsf{K} \mid (t\ t) \\
\textit{CBV contexts} & C ::= \Box \mid (C\ t) \mid (v\ C) \\
\textit{values} & v ::= \mathsf{K} \mid \mathsf{S} \mid (\mathsf{K}\ v) \mid (\mathsf{S}\ v) \mid ((\mathsf{S}\ v)\ v)
\end{array}
$$

$$
\begin{array}{rr}
\Box[t] = t & C[((\mathsf{K}\ v_1)\ v_2)] \to C[v_1] \\
(C\ t_1)[t] = (C[t]\ t_1) & C[(((\mathsf{S}\ v_1)\ v_2)\ v_3)] \to C[((v_1\ v_3)\ (v_2\ v_3))] \\
(v\ C)[t] = (v\ C[t]) &
\end{array}
$$

Fig. 4. Problem 2: Combinators.

(a) array `a` is sorted in increasing order;
(b) array `a` is a permutation of its initial contents.

Fig. 2 confirms that Problem 1 was the easiest one. Yet we were surprised to see a considerable number of quite laborious solutions, in particular regarding the definition of a permutation (task 3b). We were expecting the participants to pick up such a definition from a standard library, but almost no one did so. Various definitions for the permutation property were used: explicit lists of transpositions together with an interpretation function, bijective mapping of indices, equality of the multisets of elements, etc.

One team used the BLAST model checker to perform tasks 1 (safety) and 2 (termination). Some participants spotted that the loop test `i <= j` can be safely replaced by `i < j` (and proved it).

3.2 Problem 2: Combinators

Problem 2 is slightly different from the other ones. Instead of providing pseudo-code to be verified, this problem simply gives a specification and requires the participants to first implement a function, and then to perform some verification tasks. Namely, the problem defines call-by-value reduction of SK-terms (see Fig. 4) and then proposes one implementation task:

1. Implement a unary function `reduction` which, when given a combinator term t as input, returns a term t' such that $t \to^* t'$ and $t' \not\to$, or loops if there is no such term.

and three verification tasks:

1. Prove that if `reduction`(t) returns t', then $t \to^* t'$ and $t' \not\to$.
2. Prove that `reduction` terminates on any term that does not contain S.
3. Consider the meta-language function ks defined by

$$
\begin{array}{ll}
ks\ 0 & = \mathsf{K}, \\
ks\ (n+1) & = ((ks\ n)\ \mathsf{K}).
\end{array}
$$

   Prove that `reduction` applied to the term $(ks\ n)$ returns K when n is even, and $(\mathsf{K}\ \mathsf{K})$ when n is odd.

One subtlety regarding this problem is that function `reduction` may not terminate. Hence implementing it is already challenging in some systems; that is why implementation is a task in itself. Another consequence is that verification task 1 is a partial correctness result. Tasks 2 and 3 were slightly easier, in particular because it is possible to exhibit a termination argument for `reduction` when applied on S-free terms.

A common error in submissions was forgetting to require a value on the left hand-side of a context ($v\ C$). Indeed, in absence of a dependent type to impose this side condition in the definition of the data type for contexts, one has to resort to an extra well-formedness predicate and it was sometimes accidentally omitted from specifications. Regarding verification task 1, another error was to prove that the returned term was a value without proving that values are irreducible.

One team provided an improved result for verification task 2, showing that any S-free term necessarily reduces to a term of the shape K (K (K ...)). Another team proved as a bonus that reduction of SII(SII) diverges (with I being defined as SKK).

3.3 Problem 3: Ring Buffer

With problem 3, we are back with traditional imperative programming using arrays. A bounded queue data structure is implemented using a circular array (see Fig. 5) with operations to create a new queue, to clear it, to get or remove the first element, and to add a new element. The verification tasks are the following:

1. *Safety.* Verify that every array access is made within bounds.
2. *Behavior.* Verify the correctness of the implementation w.r.t. the first-in first-out semantics of a queue.
3. *Harness.* The following test harness should be verified.

```
test (x: int, y: int, z: int) :=
  b <- create(2);
  push(b, x);
  push(b, y);
  h <- pop(b); assert h = x;
  push(b, z);
  h <- pop(b); assert h = y;
  h <- pop(b); assert h = z;
```

The challenge of this problem is to come up with a nice specification of the first-in first-out semantics of the queue (task 2). Most participants defined the model of the queue as a list, either as a ghost field in the `ring_buffer` record itself, or as a separate logical function. Some participants, however, opted for algebraic specifications (*i.e.* showing `head`(`push`$(b, x)) = x$ and so on). Note that verification task 3 does not require a modular proof using the model defined for the purpose of verification task 2. Thus a mere calculation is accepted as a valid answer for task 3; that solution was used by several participants.

```
type ring_buffer = record
  data : array of int; // buffer contents
  size : int;          // buffer capacity
  first: int;          // queue head, if any
  len  : int;          // queue length
end

create(n: int): ring_buffer :=
  return new ring_buffer(
      data = new array[n] of int;
      size = n; first = 0; len = 0)

clear(b: ring_buffer) :=
  b.len <- 0

head(b: ring_buffer): int :=
  return b.data[b.first]

push(b: ring_buffer, x: int) :=
  b.data[(b.first + b.len) mod b.size] <- x;
  b.len <- b.len + 1

pop(b: ring_buffer): int :=
  r <- b.data[b.first];
  b.first <- (b.first + 1) mod b.size;
  b.len <- b.len - 1;
  return r
```

Fig. 5. Problem 3: Ring Buffer.

This problem appeared to be the second easiest one, judging by the number of perfect solutions, see Fig. 2. Though it was not explicitly required, some solutions are generic w.r.t. the type of elements (and then instantiated on type `int` for task 3). Some participants replaced the `mod` operation by a test and a subtraction, since `b.first + b.len` and `b.first + 1` cannot be greater than 2`b.size` $- 1$. This was accepted.

3.4 Problem 4: Tree Reconstruction

There is a little story behind the fourth problem. It is the last step in Garsia-Wachs algorithm for minimum cost binary trees [7]. It can be stated as follows: given a list l of integers, you have to reconstruct a binary tree, if it exists, such that its leaf depths, when traversed in order, form exactly l. For instance, from the list $1, 3, 3, 2$ one reconstructs the binary tree

```
type tree
Leaf(): tree
Node(l:tree, r:tree): tree

type list
is_empty(s: list): boolean
head(s: list): int
pop(s: list)

build_rec(d: int, s: list): tree :=
  if is_empty(s) then fail; endif
  h <- head(s);
  if h < d then fail; endif
  if h = d then pop(s); return Leaf(); endif
  l <- build_rec(d+1, s);
  r <- build_rec(d+1, s);
  return Node(l, r)

build(s: list): tree :=
  t <- build_rec(0, s);
  if not is_empty(s) then fail; endif
  return t
```

Fig. 6. Problem 4: Tree Reconstruction.

but there is no tree corresponding to the list $1, 3, 2, 2$. A recursive function to perform this reconstruction is given (see Fig. 6; R.E. Tarjan is credited for this code [7, p. 638]). Then the verification tasks are the following:

1. *Soundness.* Verify that whenever function `build` successfully returns a tree, the depths of its leaves are exactly those passed in the argument list.
2. *Completeness.* Verify that whenever function `build` reports failure, there is no tree that corresponds to the argument list.
3. *Termination.* Prove that function `build` always terminates.
4. *Harness.* The following test harness should be verified:
   - Verify that `build` applied to the list $1, 3, 3, 2$ returns the tree `Node(Leaf, Node(Node(Leaf, Leaf), Leaf))`.
   - Verify that `build` applied to the list $1, 3, 2, 2$ reports failure.

One difficulty here is to prove completeness (task 2). Another difficulty is to prove termination (task 3), as it is not obvious to figure out a variant for function `build_rec`. On the contrary, verification task 4 (harness) turns out to be easy as soon as one can execute the code of `build`, as in harness for problem 3.

A delightful bonus from the ACL2 team is worth pointing out: To demonstrate that function `build` is reasonably efficient, they applied it to the LISP code of function `build_rec` itself, as each S-expression can be seen as a binary tree.

```
bfs(source: vertex, dest: vertex): int :=
  V <- {source}; C <- {source}; N <- {};
  d <- 0;
  while C is not empty do
    remove one vertex v from C;
    if v = dest then return d; endif
    for each w in succ(v) do
      if w is not in V then
        add w to V;
        add w to N;
      endif
    endfor
    if C is empty then
      C <- N;
      N <- {};
      d <- d+1;
    endif
  endwhile
  fail "no path"
```

Fig. 7. Problem 5: Breadth-First Search.

3.5 Problem 5: Breadth-First Search

The last problem is a traditional breadth-first search algorithm to find out the shortest path in a directed graph, given a source and a target vertex. The graph is introduced as two abstract data types, respectively for vertices and finite sets of vertices, and a function `succ` to return the successors of a given vertex:

```
type vertex
type vertex_set
succ(v: vertex): vertex_set
```

The code for the breadth-first search is given in Fig. 7. Then the verification tasks are the following:

1. *Soundness.* Verify that whenever function `bfs` returns an integer n this is indeed the length of the shortest path from `source` to `dest`. A partial score is attributed if it is only proved that there exists a path of length n from `source` to `dest`.
2. *Completeness.* Verify that whenever function `bfs` reports failure there is no path from `source` to `dest`.

One difficulty is that the graph is not necessarily finite, thus the code may diverge when there is no path from `source` to `dest`. This was done purposely, to introduce another partial correctness task (as in problem 2). However, some participants asked during the competition if they may assume the graph to be finite, in particular to assume vertices to be the integers $0, 1, \ldots, n-1$, and

sometimes even to use an explicit adjacency matrix for the graph. We answered positively to that request. We also received solutions for this problem that did not rely on this assumption.

Another request was the possibility to rewrite the inner loop of the code (which updates sets `V` and `N`) using set operations (union, difference, etc.). We also agreed. On second thought, the problem would have been nicer if stated this way, that is with the inner loop replaced by the following two assignments:

```
N <- union(N, diff(succ(v), V));
V <- union(V, succ(v));
```

This fifth problem has the lowest number of perfect solutions: 9 out of 29 submissions.

4 Competition Outcome

4.1 And the Winners Are...

A group of 6 excellent submissions with tied scores emerged from our evaluation. Thus we opted for 6 medalists (2 bronze, 2 silver, 2 gold) to avoid discriminating between solutions that were too close. The medalists are:

Gold medal (600 points):

- Jared Davis, Matt Kaufmann, J Strother Moore, and Sol Swords with ACL2 [8,9].
- Gidon Ernst, Gerhard Schellhorn, Kurt Stenzel, and Bogdan Tofan with KIV [16].

Silver medal (595 points):

- K. Rustan M. Leino and Peter Müller with Dafny [13].
- Sam Owre and Natarajan Shankar with PVS [15,14].

Bronze medal (590 points):

- Ernie Cohen and Michał Moskal with VCC [6].
- Jason Koenig and Nadia Polikarpova with Dafny [13].

4.2 Lessons Learned

Evaluation. Probably, the most important conclusion we arrived at during the competition was that evaluation of submitted solutions is a non-trivial and, to some extent, subjective process. This is what sets deductive verification competitions apart from programming contests (where the success can be judged using series of tests) and prover competitions (where a proof trace can be mechanically verified by a trusted certification procedure). In our case, a solution consists of a *formal specification* of the problem and a *program* to solve it, both written in some system-specific language. It is the responsibility of a verification system to check that the program satisfies the requirements posed by the specification, and, to a first approximation, we can trust the verification software to do its job

correctly. Even then a user (and a judge is nothing but an exigent user) must have a good knowledge of the system in question and be aware of the hidden assumptions and turned-off-by-default checks — an issue we stumbled on a couple of times during evaluation.

What is more difficult, error-prone, and time-consuming is checking that a submitted specification corresponds to our intuitive understanding of the problem and its informal description written by humans and for humans. We found no other way to do it than to carefully read the submissions, separating specification from proofs and program code, and evaluating its conformance to our requirements. In this respect, it is not unlike peer-reviewing of scientific publications. Additionally, every system proposes its own language: sometimes quite verbose, sometimes employing a rather exotic syntax, sometimes with specification parts thinly spread among hundreds of lines of auxiliary information. Here we must commend the solutions written for PVS and Dafny for being among the easiest ones to read.

Advice to future organizers. To help improve the evaluation process, we would recommend to organizers of future competitions to gather a kind of "program committee" to which reviewing of submissions could be delegated. Such a committee would benefit from having experts in as many different tools as possible, to help provide more expert reviews of submitted solutions. While a common specification language is out of the question (as the main interest of the contest lies in comparing diverse approaches to formalization), we can strive for more rigid problem descriptions, down to first-order formulations of desired properties. We, as a community, should also push verification system developers towards well-structured and easily readable languages, with a good separation of specification from implementation and proof. Indeed, future organizers might consider requiring the specification of each part of a problem to be clearly indicated, either in comments or even better, by placing the specifications in separate files.

Advice to future competitors. The single biggest issue we had in evaluating solutions was just understanding the specifications of the theorems. One has little choice but to trust the verification tool which claims the solution is correct, but we cannot escape the need to judge whether that solution solves the stated problem or falls short in some way (for example, by adding an assumption that was not explicitly allowed, or by incorrectly formulating some property). So anything a competitor can do to make it as clear and comprehensible as possible what the specification is will help making judging easier and more likely less error-prone. Sometimes even determining which parts of a set of proof scripts or files constitute the specification was challenging. And of course, any special notation or syntax for a tool should be carefully (but briefly) introduced in the README for the submission (and/or possibly inline in the submission itself, where it is first used). Finally, a few submissions we evaluated were not based on plain text files, but rather required a specialized viewer even to look at the solutions. This posed problems for us, particularly when the viewer was only available on one platform (e.g., Windows). We recommend that all tools based on specialized

viewers have some kind of export feature to produce plain text, at least for the statements of theorems proved.

Problem difficulty. When several independent problems are proposed, it is not easy to estimate their relative difficulty in an unbiased manner. The solution that you devise for your problem is not necessarily the best or the simplest one: we were pleasantly surprised to see some participants find more elegant and concise formulations of our algorithms and specifications than those we came up with ourselves (cf. the inner loop in problem 5). Also, what is hard to do in your system of choice might be easy with some other tool. It is always better to draw in several independent and competent testers, preferably using different systems, before the competition.

Verification tasks. An interesting class of verification problems is related to termination issues. Even for systems that admit diverging programs it is not always possible to specify and prove non-termination on a certain input (and we did not include any such task in our problems). Somewhat paradoxically, the systems that are based on logics with total functions (such as Coq) are better suited for this task, as some indirection is required anyway to describe a diverging computation (for example, a supplementary "fuel" parameter).

5 Conclusion

Organizing this competition was a lot of fun — and it seems it was so for the participants as well, which was one of our goals. But it was also a lot of work for us to evaluate the solutions. Obviously this format cannot be kept for future competitions, especially if we anticipate on an even greater number of participants. Alternatives include on-site competitions in limited time (to limit the number of participants), peer-reviewing of the solutions (to limit the workload), and servers with pre-installed verification software (to avoid the installation burden).

Acknowledgments. A large number of people contributed to the success of this competition. We would like to thank: our beta-testers Claude Marché and Duckki Oe; VSTTE 2012 chairs Ernie Cohen, Rajeev Joshi, Peter Müller, and Andreas Podelski; VSTTE 2012 publicity chair Gudmund Grov; LRI's technical staff. Finally, we are grateful to Vladimir Klebanov for encouraging us to write this paper.

References

1. The 2nd Verified Software Competition, 2011. `https://sites.google.com/site/vstte2012/compet`.
2. Clark Barrett, Morgan Deters, Leonardo Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012. to appear in print, available from Springer Online.

3. Dirk Beyer. Competition on Software Verification (SV-COMP). In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012*, volume 7214 of *LNCS*, page 504–524. Springer, 2012. Materials available at `http://sv-comp.sosy-lab.org/2012/index.php`.
4. François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, 2010. `http://why3.lri.fr/`.
5. Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. The COST IC0701 Verification Competition 2011. In F. Damiani and D. Gurov, editors, *Formal Verification of Object-Oriented Software, Revised Selected Papers Presented at the International Conference, FoVeOOS 2011*. Springer Verlag, 2012. Materials available at `http://foveoos2011.cost-ic0701.org/verification-competition`.
6. Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430, 2009.
7. Adriano M. Garsia and Michelle L. Wachs. A new algorithm for minimum cost binary trees. *SIAM J. on Computing*, 6(4):622–642, 1977.
8. Matt Kaufmann and J. Strother Moore. *ACL2 Version 4.3*. 2011. `http://www.cs.utexas.edu/users/moore/acl2`.
9. Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
10. Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st Verified Software Competition: Experience report. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*. Springer, 2011. Materials available at `www.vscomp.org`.
11. Donald E. Knuth. *The Art of Computer Programming, volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.
12. D. Le Berre and L Simon, editors. *Special Issue on the SAT 2005 Competitions and Evaluations*, volume 2, 2006.
13. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.
14. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
15. The PVS system. `http://pvs.csl.sri.com/`.
16. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 69–72, Townsville, North Queensland, Australia, july 1997.
17. G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.