Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

VerifyThis 2012

A Program Verification Competition

Marieke Huisman[1], **Vladimir Klebanov**[2], **Rosemary Monahan**[3]

[1] University of Twente, The Netherlands, e-mail: m.huisman@utwente.nl
[2] Karlsruhe Institute of Technology, Germany, e-mail: klebanov@kit.edu
[3] Maynooth University, Maynooth, Co. Kildare, Ireland, e-mail: Rosemary.Monahan@nuim.ie

Received: date / Revised version: date

Abstract. VerifyThis 2012 was a two-day verification competition taking place as part of the International Symposium on Formal Methods (FM 2012) on August 30-31, 2012 in Paris, France. It was the second installment in the VerifyThis series. After the competition, an open call solicited contributions related to the VerifyThis 2012 challenges and overall goals. As a result, seven papers were submitted and, after review and revision, included in this special issue.

In this introduction to the special issue, we provide an overview of the VerifyThis competition series, an account of related activities in the area, and an overview of solutions submitted to the organizers both during and after the 2012 competition. We conclude with a summary of results and some remarks concerning future installments of VerifyThis.

1 Introduction

Software is vital for modern society. The efficient development of correct and reliable software is of ever growing importance. An important technique for achieving this goal is *formal verification:* demonstrating in a mathematically rigorous and machine-checked way that a program satisfies a given formal specification of what is considered correct behavior. In the last decade, technologies for the formal verification of software—mostly based on logics and formal reasoning—have been rapidly maturing and are on the way to complement and partly replace traditional software engineering methods.

However, to achieve a major uptake of formal verification techniques in industrial practice, realistic demonstrations of their capabilities are needed. This major challenge for formal verification was identified 20 years ago, as illustrated by the following quote from [LL95]:

> A recent questionnaire [Formal Methods: A Survey, 1993] of the British National Physical Laboratory (NPL) showed that one of the major impediments of formal methods to gain broader acceptance in industry is the **lack of realistic, comparative surveys**.

Surprisingly this observation is still accurate and relevant.

One way to improve this situation is to systematically encourage comparative evaluation of formal verification techniques. It has become generally accepted wisdom that regular evaluation helps focus research, identify relevant problems, bolster development, and advance the field in general. Benchmark libraries and competitions are two popular approaches. Competitions are widely acknowledged as a means of improving the available tools, increasing the visibility of their strengths, and establishing a publicly available set of benchmark problems. In the formal methods community (loosely interpreted), competitions include those on SAT, SMT, Planning, Quantified Boolean Formulas, Hardware Model Checking, Software Model Checking, and Automated Theorem Proving[1]. These events had a significant positive impact on the development speed and the quality of the participating tools as theoretical results are transferred to practical tools almost instantly.

This special issue of Software Tools for Technology Transfer (STTT) reports on the VerifyThis 2012 competition, which posed program verification challenges concerned with expressive data-centric properties. In this introduction, we present the competition challenges along with a high-level overview of the solutions, report on

[1] http://www.satcompetition.org, http://www.smtcomp.org, http://ipc.icaps-conference.org, http://www.qbflib.org/competition.html, http://fmv.jku.at/hwmcc11, http://sv-comp.sosy-lab.org, and http://www.cs.miami.edu/~tptp/CASC.

the results of the competition and conclude with some suggestions for future installments.

1.1 About VerifyThis

VerifyThis 2012 was a two-day event taking place as part of the Symposium on Formal Methods (FM 2012) on August 30-31, 2012 in Paris, France. It was the second installment in the VerifyThis series (though the first one explicitly branded as such) after the program verification competition held at FoVeOOS 2011.

The aims of the VerifyThis competition series 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.

Typical challenges in the VerifyThis competitions are small but intricate algorithms given in pseudo-code with an informal specification in natural language. Participants have to formalize the requirements, implement a solution, and formally verify the implementation for adherence to the specification. The time frame to solve each challenge is quite short (between 45 and 90 minutes) so that anyone can easily repeat the experiment.

Correctness properties are typically expressive and concerned with data. To tackle them to the full extent, some human guidance for the verification tool is usually required. At the same time, the competition welcomes participation of automatic tools. Considering partial properties or simplified problems, if this suits the pragmatics of the tool, is encouraged. Combining complementary strengths of different kinds of tools is a development that VerifyThis would like to advance in the future. Submissions are judged by the organizers for

- correctness,
- completeness, and
- elegance.

The focus is primarily on the usability of the tools, their facilities for formalizing the properties to be specified and the helpfulness of their output.

For the first time, the 2012 competition included a post-mortem session where participants explained their solutions and answered questions of the judges. In parallel, the participants used this session to discuss details of their solutions amongst each other.

In another first, challenges were solicited from the public in advance of the competition, and eight suggestions for challenges were received. Even though we decided not to use the submitted challenges directly,[2] the call for challenge submissions was useful, as it provided:

- additional challenges that formal verification technique developers can try their tools upon;
- insight into what people in the community consider interesting, challenging and relevant problems; and
- inspiration for further challenges.

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

- student teams (including PhD students),
- non-developer teams using a tool someone else developed, and
- several teams using the same tool.

Note that teams were welcome to use different tools for different challenges (or even for the same challenge).

The competition website can be found at `http://fm2012.verifythis.org/`. More background information on the competition format and the choices made can be found in [HKM12]. Reports from previous competitions of similar nature can be found in [KMS+11, BBD+12, FPS12].

1.2 VerifyThis 2012 Participants and Tools Used

Participating teams and the tool which they used in the competition follow in no particular order:

1. Bart Jacobs, Jan Smans (VeriFast [PMP+14])
2. Jean-Christophe Filliâtre, Andrei Paskevich (Why3 [FP13])
3. Yannick Moy (GNATprove [DEL+14])
4. Wojciech Mostowski, Daniel Bruns (KeY [ABB+14])
5. Valentin Wüstholz, Maria Christakis (Dafny [Lei10]) (student, non-developer team)
6. Gidon Ernst, Jörg Pfähler (KIV [RSSB98]) (student team)
7. Stefan Blom, Tom van Dijk (ESC/Java2 [CK05]) (non-developer team)
8. Zheng Cheng, Marie Farrell (Dafny) (student, non-developer team)
9. Claude Marché, François Bobot (Why3)
10. Ernie Cohen (VCC [CDH+09])
11. Nguyen Truong Khanh (PAT [LSD10])

1.3 Papers presented in this Special Issue

After the competition, an open call for this issue of STTT solicited contributions related to the VerifyThis 2012 challenges and overall goals. This call targeted not only competition participants, but also anyone interested in tackling the challenges, using them as a benchmark for novel techniques, or advancing the agenda of VerifyThis in general. Contributors were encouraged to share their experience of the competition challenges containing topics such as (but not limited to) the following:

- details of the tool/approach used
- material emphasizing usability of the tool

[2] In particular, because the author of the best challenge submission was participating in the competition.

- discussion of completed challenges
- details of additional related challenges
- a reflection on what was learned from completing the competition challenges (advancements that are necessary for tools, usability issues, comparison with other tools etc.)
- a report on the experience of participating in the competition

As a result, seven papers were submitted and, after review and revision, included in this issue. The first paper in this issue is contributed by the VeriFast team, who won the prize for the best team [JSP15]. They provide an introduction to the VeriFast tool, and then describe their solutions to the competition's challenges, including several post-competition alternative and improved solutions. The next paper in this issue is contributed by the KIV team, who won the prize for the best student team [EPS+15]. They introduce the KIV tool, and describe their solutions to the competitions challenges, including a comparison to other solutions. Next, this special issue continues with the contribution of the GNATprove team, who won the prize for the best user-assistance tool feature [HMWC15]. This paper introduces GNATprove, and discusses who it is used on the first two challenges. The special issue then continues with a contribution by the combined Why3 teams [BFMP15], who introduce Why3, and describe the solutions to the challenges that were developed post-competition, combining and polishing the competition solutions of the two Why3 teams. The last contribution of a competition participation is provided by the KeY team [BMU15], who introduce the KeY verifier, and discuss their solutions to the challenges, as developed during the competition, and completed afterwards. The special issue then continues with a contribution by the developers of the AutoProof verifier [TFN15]. They did not participate in the competition, but describe how their tool has been tried on the challenges afterwards. They do not provide full solutions to all challenges; in some cases they only verify a single use case. Finally, this special issue concludes with a slightly different contribution: Blom (from the ESC/Java team) and Huisman discuss how they extended their VerCors tool set to support reasoning about magic wands, and used this extension to solve the third challenge [BH15].

1.4 Related Efforts and Activities

There are a number of related accounts and activities that we would like to mention before presenting the VerifyThis 2012 details.

A historically interesting qualitative overview of the state of program verification tools was compiled in 1987 by Craigen [Cra87]. There are also several larger comparative case studies in formal development and verification, treated by a number of different methods and tools. Here we name the RPC-memory specification case study, resulting from a 1994 Dagstuhl meeting [BMS96], the "production cell" case study [LL95] from 1995, and the Mondex case study [Woo06].

Recently we have seen a resurgence of interest in benchmarking program verification tools. In particular, several papers appeared during the last years presenting specific challenges for program verification tools and techniques [LLM07, WSH+08, LM10]. In addition, the recent COST Action IC0701 maintains an online repository[3] of verification challenges and solutions (which focuses mainly on object-oriented programs).

Of note are the following competitions closely related to ours:

- The first "modern" competition and an inspiration for VerifyThis was the Verified Software Competition (VSComp[4]), organized by the Verified Software Initiative (VSI). Its first installment took place on site at VSTTE 2010. Subsequent VSComp installments included several 48-hour online competitions, and a larger verification challenge, running over a period of several months. In general, the problems tackled during VSComp are larger than those in VerifyThis, as time restrictions are less strict.
- Since 2012, the SV-COMP[5] software verification competition takes place in affiliation with the TACAS conference. This competition focuses on fully automatic verification, and is off-line, i.e., participants submit their tools by a particular date, and the organizers check whether they accurately handle the challenges. We have regular contact with the SV-COMP organizers, and in particular, we monitor the (shrinking) gap between the expressive properties tackled in VerifyThis and the automation achieved by tools evaluated in SV-COMP.
- The RERS Challenge[6] taking place since 2010 is dedicated to rigorous examination of reactive systems. The Challenge aims to bring together researchers from all areas of software verification and validation, including theorem proving, model checking, program analysis, symbolic execution, and testing, and discuss the specific strengths and weaknesses of the different technologies.

In contrast, the unique proposition of the VerifyThis competition series is that it assesses the user-tool interaction and emphasizes the repeatability of the evaluation within modest time requirements.

In April 2014, we organized (together with Dirk Beyer of SV-COMP) a Dagstuhl seminar on "Evaluating Software-Verification Systems: Benchmarks and Competitions" [BHKM14], where we gathered participants and organizers of different verification-related competitions.

[3] `http://www.verifythis.org`.
[4] `http://www.vscomp.org`
[5] `http://sv-comp.sosy-lab.org`
[6] `http://rers-challenge.org`

The event was concluded with a joint VerifyThis/SV-COMP competition session. The verification challenge chosen was based on a bug encountered in the Linux kernel.[7]

Participants were encouraged to build teams of up to three people, in particular mixing attendees and tools from different communities. The applied automatic verifiers (typical of tools used in SV-COMP) could detect the assertion violation easily, though interpreting the error path and locating the bug cause required not negligible effort. Unsurprisingly, proving the program correct after fixing the bug was not easy for most automatic verifiers (with the notable exception of the Predator tool). With deductive verifiers typically used in VerifyThis the situation was more varied. Several teams succeeded in verifying parts of the code respective to a subset of assertions. Success factors were support for verifying C programs (as otherwise time was lost translating the subject program into the language supported by the verifier) and having found the bug first either by testing or by using automatic verification as an auxiliary technique. An interesting question that arose for future investigation is, whether and how the automatically synthesized safety invariants provided by some automatic verifiers can be used in a deductive verifier.

2 VerifyThis 2012 Challenge 1: Longest Common Prefix (LCP, 45 minutes)

2.1 Verification Task

Longest Common Prefix (LCP) is a problem in text querying [SW11]. 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;
}
```

[7] The challenge can be found in the SV-COMP database at `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/heap-manipulation/bubble_sort_linux_false-unreach-call.c`

2.2 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 (i.e., the "longest" qualifier in LCP), 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 much of the quantifier verbosity. The maximality was typically expressed by a variation of the following expression:

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

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

2.3 Advanced Verification Tasks

For those who have completed the LCP challenge quickly, the description included a further challenge, named LRS, outlined below. No submissions attempting to solve the advanced challenge were received during the competition. Three solutions developed later are presented in the papers in this special issue.

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].

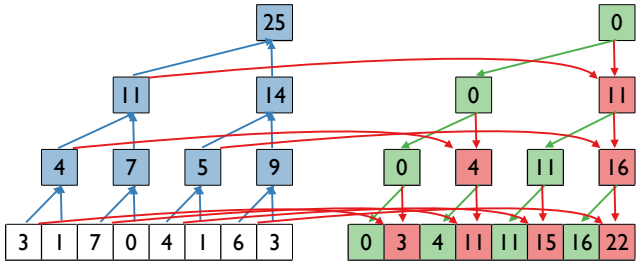

Fig. 1. Upsweep and downsweep phases of the prefix sum calculation, picture taken from [Cho14].

Verification task. The attached Java code[8] 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.

Results. This special issue contains contributions from the KeY, KIV, and the (joint) Why3 teams with solutions to the LRS challenge. The effort needed to develop them is reported at a couple of days rather than hours. The difficult part of the challenge is to prove maximality of the computed solution.

Future verification tasks. Together with the call for contributions to this special issue, we put forth a challenge to verify one of the advanced suffix array implementations optimized for performance, such as, e.g., [KS03]. So far, this challenge remains unmet. An interesting potential approach would be to verify that a complex implementation equals or corresponds in its functional behavior to a simple one. This technique known as *regression verification* does not require a functional correctness specification and in many cases features a favorable pragmatics.

3 VerifyThis 2012 Challenge 2: Prefix Sum (PrefixSum, 90 minutes)

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 [Ble93]. 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[9]). 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).

3.3 Verification Task

We provide an iterative and a recursive implementation of the algorithm (shown in Appendix B). 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 Organizer Comments

Eight submissions were received at the competition. Though the upsweep and downsweep algorithm were not complex, it was challenging to build a mental model of what is happening. The VeriFast solution was the only one judged as sufficiently correct and complete.

In this recursive solution, upsweep and downsweep are specified in terms of recursive separation logic predicates, allowing the proofs to consist of explicit unfolding and folding of the relevant predicates. A simple final lemma was proved by induction. After the competition, the KIV and the combined Why3 teams also

[8] Available as part of the original challenge description at `fm2012.verifythis.org` and in Appendix A.

[9] The original challenge description contained an illustrating excerpt from a slide deck on prefix sums

provided complete versions of both upsweep and downsweep. These solutions are presented in details in the papers corresponding to each tool within this special issue.

The main "technical" problem in this challenge was reasoning about powers of two. The GNATprove team was the only team to make use of the bounded array simplification proposed in the challenge description. It was also the only team that attempted to verify the iterative version of the algorithm and not the recursive one during the competition (the KIV team developed an iterative solution in the aftermath). In this issue, the GNATprove team report that, as a follow up to the competition, they also generalized the specification in both SPARK 2005 and SPARK 2014 as a useful exercise in comparing the new and old version of the tools.

The ability of the GNATprove tool to test the requirement and auxiliary annotations by translating them to run-time checks was helpful in this challenge. This feature won the distinguished prize of user-assistance tool feature awarded by the jury of the VerifyThis competition. The Why3 paper makes an observation that a facility to "debug the specifications" would have assisted greatly in developing a solution. The KIV team states that "inspecting (explicit) proof trees of failed proof attempts" was an invaluable help in finding out which corrections were necessary during the iterative development process.

The AutoProof team's main difficulty with this challenge was expressing how the original array is modified at each iteration. In this issue, they explain how this would have been overcome if **old** expressions could be used in loop invariants (in iterative solutions) or in postconditions within the scope of bounded **across** quantifier (in recursive solutions). Using workarounds, such as making copies of the initial arrays for future reasoning, or defining specific predicates for each property, resulted in a verification that was too difficult for AutoProof in its early stage of development.[10] A full report is provided in this issues' AutoProof paper.

While modular verification remains the main goal of tool development, the advantages of the possibility to fall back to non-modular verification are now gaining wider recognition. In absence of specifications, tools like KIV, KeY, and AutoProof can verify concrete clients by inlining the bodies of functions called in the client code or exhaustively unrolling bounded loops. This establishes that the implementation is correct for the given client. Although, a generalized proof is not obtained at first, this "two-step verification" process helps speedup the debugging of failed verification attempts and guides the generalization of partial verification attempts.

After the competition, the KeY team provided a partial solution to this challenge, with a recursive implementation and a partial specification concerned only with the rightmost element after the upsweep phase. A complete specification for upsweep is also provided in their solution presented in this issue, although its proof is not completed. Challenges were reasoning about sums and the exponential function. The KIV and Why3 teams benefited in this challenge as their libraries already included a formalization of the exponentiation operator. The Why3 team also imported the sum function, and its associated proofs, from their tool's standard library.

Another hot topic of the past is the ability to check absence of integer overflow. Currently, all the participating tools have the capabilities to do so. Now, the flexibility to enable or disable such checks (potentially in a fine-grained way) has become an important property. The support of ghost variables proved useful for many teams when expressing loop invariants and passing arrays to the downsweep procedure. The KeY team also reported that using frames with KeY's built-in data type of location sets added structure to the proof.

This challenge demonstrated the requirement for user interaction during proof construction. This interaction comes via both textual and non-textual (point-and-click) interaction styles, with some tools, e.g., KeY and KIV combining both styles. While the textual interaction paradigm has advantages w.r.t. effort reuse across proof iterations, the point-and-click style can at times offer greater flexibility.

3.5 Future Verification Tasks

A verification system supporting concurrency could be used to verify a parallel algorithm for prefix sum computation [Ble93].

4 VerifyThis 2012 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.

[10] AutoProof has been significantly improved since its 2013 version used here.

```
(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 [Tue10] 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 by Thomas Tuerk in [Tue10] 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. After the competition, the combined Why3 teams, the KeY team, and the ESC/Java2 team also developed a solution for this challenge. These solutions are discussed in details within the correspondng papers in this issue.

During the competition, the VeriFast team developed a solution based on (an encoding of) a "magic wand" operator of separation logic, which describes how one property can be *exchanged* or *traded* for a different property. In this challenge, the magic wand operator is used to describe the loop outcome[11], which captures the "tree with a hole" property: if the magic wand is combined with the subtree starting at `pp`, then a full tree is re-established.

In VeriFast, the magic wand operator is encoded by a predicate-parameterized lemma describing the transformation that is done by the magic wand. A similar solution was developed by the ESC/Java2 team. In fact, during the competition, the team worked out this solution on paper, but as ESC/Java2 did not provide sufficient support for reasoning about pointer programs, they did not attempt any tool-based verification. After the competition, the team extended their VerCors tool set for the verification of concurrent software using permission-based separation logic [BH14], with support for magic wand reasoning.

The VerCors tool set translates annotated Java programs into annotated Chalice, which is a a small, class-based language that supports concurrency via threads, monitors, and messages, and then uses Chalice's dedicated program verifier. The translation encodes complex aspects of the Java semantics and annotation language. The paper that Blom and Huisman contributed to this special issue shows how parametrized abstract predicates and magic wands are encoded into Chalice, by building witness objects that enable manipulation of the encoded assertions. They illustrate their encoding by verifying the tree delete challenge, using a loop invariant that is similar to the loop specification used by VeriFast. However, the difference is that in their approach, the user is directly manipulating a magic wand, and the encoding is done by the tool, while in VeriFast, the user has to encode the magic wand themselves.

The VeriFast team also developed an alternative post-competition solution, which does not use the magic wand operator but instead defines a recursive tree-with-a-hole predicate coupling the concrete data structure and two abstract trees. Using this predicate, a loop invariant maintains that the original tree can be decomposed into a tree with "a hole at `pp`", and another complete tree, starting at `pp`. When the loop finishes, and the left-most element is removed, this decomposition is used to create the final tree. The VeriFast team's contribution to this special issue describes both solutions.

The KIV team is the only team that applied "forward reasoning", which is the most efficient solution to this challenge, to the full extent. In KIV, the forward argument was not shaped as a block contract annotation and rule but as induction over the number of loop iterations during the proof. While a loop invariant can only talk about the loop body, the induction hypothesis can cover both the loop and the following tree modification. It can thus be easily expressed using the standard tree definition only. The correctness proof in KIV is furthermore structured in two parts: at first, a correspondence between the iterative pointer program and a recursive functional program operating on abstract trees is proved (here, the above-mentioned induction is performed). Then, the functional program is proved correct w.r.t. the requirement (removing the minimal element).

The Why3 teams developed a solution to the challenge after the competition, which is based on the notion of Huet's Zipper [Hue97]. A zipper is a special data structure that can be used to encode arbitrary paths (and updates) in aggregate data structures. Since in the tree delete algorithm, always the left branch of a tree

[11] This specification uses a loop contract. If the tool supported contracts for arbitrary code blocks, then the modification after the loop could be included and a simpler solution as proposed by Tuerk would have been possible.

is chosen, the Why3 team used a simplified version of the zipper. From a zipper and a subtree, the complete tree can be recovered. The zipper is maintained in the program as a ghost variable, which makes it thus an operational and constructive encoding of the "tree with a hole".

Finally, the KeY team describe a post-competition solution to the problem in this issue. They use a quite different approach to handle this challenge. Their specifications are written in terms of an "abstract flat representation of the tree". In addition, they use the notion of footprint to capture that the tree is indeed a tree, and that the tree-structure is preserved throughout the iteration. To prove that the minimal element is removed by the algorithm, they maintain a loop invariant on the abstract flat representation of the tree, using the currently visited node to separate the upper part, i.e., the nodes that do not have to be examined anymore, and the lower part, i.e., the nodes that still may be changed by the deletion operator. The key property is that the footprint of the upper part is strictly disjoint from the footprint of the lower part, thus changes in the lower part will not affect the upper part.

5 Prizes, 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 (both tools had 2 user teams)
- 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.

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.

5.2 Post-mortem Session

The post-mortem session, on 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 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 a 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.

Acknowledgments

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

References

[ABB+14] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou, Daniel Kroening, Elizabeth Polgreen, and Natarajan Shankar, editors, *Proceedings, 6th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE), Vienna, July 2014*, LNCS. Springer, 2014.

[BBD+12] 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 Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *International Conference on Formal Verification of Object-Oriented Systems (FoVeOOS 2011)*, LNCS. Springer, 2012.

[BFMP15] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[BH14] S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[BH15] Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[BHKM14] Dirk Beyer, Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. Evaluating Software Verification Systems: Benchmarks and Competitions (Dagstuhl Reports 14171). *Dagstuhl Reports*, 4(4):1–19, 2014.

[Ble93] Guy E. Blelloch. Prefix sums and their applications. In John H. Reif, editor, *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[BMS96] M. Broy, S. Merz, and K. Spies, editors. *Formal Systems Specification. The RPC-Memory Specification Case Study*, volume 1169 of *LNCS*. Springer, 1996.

[BMU15] Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42. Springer-Verlag, 2009.

[Cho14] N. Chong. *Scalable Verification Techniques for Data-Parallel Programs*. PhD thesis, Imperial College London, 2014.

[CK05] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 108–128. Springer-Verlag, 2005.

[Cra87] Dan Craigen. Strengths and weaknesses of program verification systems. In *Proc. of the 1st European Software Engineering Conference on ESEC '87*, pages 396–404. Springer-Verlag, 1987.

[DEL+14] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentré, and Yannick Moy. Rail, space, security: Three case studies for SPARK 2014. In *7th Europen Congress on Embedded Real Time Software and Systems (ERTS^2 2014)*, 2014.

[EPS+15] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. KIV: overview and VerifyThis competition. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3: Where programs meet provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pages 125–128. Springer-Verlag, 2013.

[FPS12] Jean-Christophe Filliâtre, Andrei Paskevich, and Aaron Stump. The 2nd Verified Software Competition: Experience report. In Vladimir Klebanov, Armin Biere, Bernhard Beckert, and Geoff Sutcliffe, editors, *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, 2012.

[HKM12] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. On the organisation of program verification competitions. In Vladimir Klebanov, Bernhard Beckert, Armin Biere, and Geoff 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.

[HMWC15] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove. A competition report from builders of an industrial-strength verifying compiler. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[Hue97] G. Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 1997.

[JSP15] Bart Jacobs, Jan Smans, and Frank Piessens. Solving the VerifyThis 2012 challenges with VeriFast. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[KMS+11] 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)*, volume 6664 of *LNCS*. Springer, 2011.

[KS03] Juha Kärkkäinen and Peter 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.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370. Springer-Verlag, 2010.

[LL95] Claus Lewerentz and Thomas Lindner. Case study "production cell": A comparative study in formal specification and verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, pages 388–416. Springer, 1995.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Form. Asp. Comput.*, 19:159–189, June 2007.

[LM10] K. Rustan M. Leino and Michał Moskal. VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.

[LSD10] Yang Liu, Jun Sun, and Jin Song Dong. Developing model checkers using PAT. In *Proceedings of the 8th International Conference on Automated Technology for Verification and Analysis*, ATVA'10, pages 371–377. Springer-Verlag, 2010.

[PMP+14] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with verifast: Industrial case studies. *Sci. Comput. Program.*, 82:77–97, 2014.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In Wolfgang Bibel and PeterH. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume 9 of *Applied Logic Series*, pages 13–39. Springer Netherlands, 1998.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[TFN15] Julian Tschannen, Carlo A. Furia, and Martin Nordio. AutoProof meets some verification challenges. *Int. J. Softw. Tools Technol. Transfer, in this issue*, 2015.

[Tue10] Thomas Tuerk. Local reasoning about while-loops. In Peter Müller, David Naumann, and Hongseok Yang, editors, *Proceedings, VS-Theory Workshop of VSTTE 2010*, pages 29–39, 2010.

[Woo06] J. Woodcock. First steps in the verified software grand challenge. *Computer*, 39(10):57–64, 2006.

[WSH+08] Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce M. 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, *Proceedings, Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 5295 of *LNCS*, pages 84–98. Springer, 2008.

A LCP/LRS Source Code

```
public class SuffixArray {

    private final int[] a;
    private final int[] suffixes;
    private final int N;

    public SuffixArray(int[] a) {
        this.a = a;
        N = a.length;
        suffixes = new int[N];
        for (int i = 0; i < N; i++) suffixes[i] = i;
        sort(suffixes);
    }


    public int select(int i) {
        return suffixes[i];
    }


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


    public int lcp(int i) {
        return lcp(suffixes[i], suffixes[i-1]);
    }


    public int compare(int x, int y) {
        if (x == y) return 0;
        int l = 0;

        while (x+l<N && y+l<N && a[x+l] == a[y+l]) {
            l++;
        }

        if (x+l == N) return -1;
        if (y+l == N) return 1;
        if (a[x+l] < a[y+l]) return -1;
        if (a[x+l] > a[y+l]) return 1;

        throw new RuntimeException();
    }


    public void sort(final int[] data) {
        for(int i = 0; i < data.length + 0; i++) {
            for(int j = i;
                j > 0 && compare(data[j-1], data[j]) > 0; j--) {
                final int b = j - 1;
                final int t = data[j];
                data[j] = data[b];
                data[b] = t;
            }
        }
    }


    public static void main(String[] argv) {
        int[] arr = {1,2,2,5};
        SuffixArray sa = new SuffixArray(arr);
        System.out.println(sa.lcp(1,2));
        int[] brr = {1,2,3,5};
        sa = new SuffixArray(brr);
        System.out.println(sa.lcp(1,2));
        int[] crr = {1,2,3,5};
        sa = new SuffixArray(crr);
        System.out.println(sa.lcp(2,3));
        int[] drr = {1,2,3,3};
        sa = new SuffixArray(drr);
        System.out.println(sa.lcp(2,3));
    }

}
//Based on code by Robert Sedgewick and Kevin Wayne.
```

```
public class LRS {

    private static int solStart = 0;
    private static int solLength = 0;
    private static int[] a;

    public static void main(String[] args) {
        a = new int[args.length];
        for (int i=0; i<args.length; i++) {
            a[i]=Integer.parseInt(args[i]);
        }
        doLRS();
        System.out.println(solStart+"->"+solLength);
    }


    public static void doLRS() {
        SuffixArray sa = new SuffixArray(a);

        for (int i=1; i < a.length; i++) {
            int length = sa.lcp(i);
            if (length > solLength) {
                solStart = sa.select(i);
                solLength = length;
            }
        }
    }

}
//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 */
```