VerifyThis 2016

A Program Verification Competition

Marieke Huisman[1], **Rosemary Monahan**[2], **Peter Müller**[3], **Erik Poll**[4]

[1] University of Twente, The Netherlands, e-mail: `m.huisman@utwente.nl`
[2] Maynooth University, Ireland, e-mail: `Rosemary.Monahan@nuim.ie`
[3] ETH Zurich, Switzerland, e-mail: `peter.mueller@inf.ethz.ch`
[4] Radboud University Nijmegen, The Netherlands, e-mail: `erikpoll@cs.run.nl`

Abstract. VerifyThis 2016 was a one-day program verification competition which took place on April 2nd, 2016 in Eindhoven, The Netherlands as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2016). It was the fifth instalment in the VerifyThis competition series. This article provides an overview of the VerifyThis 2016 event, the challenges that were posed during the competition, and a high-level overview of the solutions to these challenges. It concludes with the results of the competition.

1 Introduction

VerifyThis 2016 took place on April 2nd, 2016 in Eindhoven, the Netherlands, as a one-day verification competition in the European Joint Conferences on Theory and Practice of Software (ETAPS 2016). It was the fifth edition in the VerifyThis series after the competitions held at FoVeOOS 2011, FM2012, Dagstuhl (Seminar 14171, April 2014), and ETAPS 2015 (April 2015).

The aims of the competition were:

- 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 formalise the requirements, implement a solution, and formally verify the implementation for adherence to the specification. There are no restrictions on the programming language and verification technology used. The time frame to solve each challenge is quite short (90 minutes) so that anyone can easily repeat the experiment. Examples of the verification challenges are available from the VerifyThis website http://www.verifythis.org/.

The correctness properties which the challenges present are typically expressive and focus on the input-output behaviour of programs. To tackle them to the full extent, some human guidance within a verification tool is usually required. At the same time, considering partial properties or simplified problems, if this suits the pragmatics of the tool, is encouraged. The competition welcomes participation of automatic tools as combining complementary strengths of different kinds of tools is a development that VerifyThis would like to advance.

Submissions are judged for correctness, completeness, and elegance. The focus includes the usability of the tools, their facilities for formalizing the properties and providing helpful output.

1.1 VerifyThis 2016

VerifyThis 2016 consisted of three verification challenges. Before the competition, an open call for challenge submissions was made. As a result, six challenges were submitted, of which one was used as inspiration and bonus challenge for the competition. The challenges (presented later) provided reference implementations at different levels of abstraction.

Fourteen teams participated (Table 1). Teams of up to two people were allowed and physical presence on site was required. We particularly encouraged participation of:

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

– several teams using the same tool

Teams using different tools for different challenges (or even for the same challenge) were welcome.

We started the competition day with an invited tutorial by Rustan Leino on the Dafny language and verifier [13]. Dafny has been one of the most popular tools during previous (and also the present) verification competitions. Therefore, we found it useful to expose the participants to the basic ideas behind it. The tutorial included a small verification challenge, which was not part of the competition: The participants developed a verified solution to the Dutch national flag problem [5]. The Dafny tutorial was open to all ETAPS participants.

As in the previous competitions, the day after the competition a post-mortem session was held, where participants explained their solutions and answered questions of the judges. In parallel, the participants used this session to discuss details of the problems and solutions among each other.

The website of the 2016 instalment of VerifyThis can be found at http://etaps2016.verifythis.org/. More background information on the competition format and the rationale behind it can be found in [9]. Reports from previous competitions of similar nature can be found in [12,3,8,11], and in the special issue of the International Journal on Software Tools for Technology Transfer (STTT) on the VerifyThis competition 2012 (see [10] for the introduction).

1.2 Post-mortem Sessions

Two concurrent post-mortem sessions were held the day after the competition. During one session, the judges asked the teams questions in order to better understand and appraise their solutions. Concurrently, all other participants presented their solutions to each other. These presentations were also attended by some non-participants.

1.3 Judging Criteria

Limiting the duration of each challenge assists the judging and comparison of each solution. However, this task is still quite subjective and hence, difficult. Discussion of the solution with the judges typically results in a ranking of solutions for each challenge.

Criteria that were used for judging were:

– Correctness: is the formalisation of the properties adequate and fully supported by proofs?
– Completeness: are all tasks solved, and are all required aspects covered?
– Readability: can the submission be understood easily, possibly even without a demo?
– Effort and time distribution: what is the relation between time expended on implementing the program vs. specifying properties vs. proving?
– Automation: how much manual interaction is required, and for what aspects?
– Novelty: does a submission apply novel techniques?

2 Challenge 1: Matrix Multiplication

This problem was inspired by the challenge submitted by Daniel Grahl, Karlsruhe Institute of Technology. He suggested to verify Strassen's algorithm for matrix multiplication. However, we felt that this would be too involved for a 90 minutes slot. Therefore we added two tasks related to the naive matrix multiplication algorithm, while the verification of Strassen's algorithm was left as a bonus exercise. The actual challenge was phrased in collaboration with Wojciech Mostowski, Halmstad University.

2.1 Verification Task

Consider the following pseudocode algorithm, which is naive implementation of matrix multiplication. For simplicity we assume that the matrices are square.

```
int[][] matrixMultiply(int[][] A, int[][] B) {
    int n = A.length;

    // initialise C
    int[][] C = new int[n][n];

    for (int i = 0; i < n; i++) {
           for (int k = 0; k < n; k++) {
                  for (int j = 0; j < n; j++) {
                      C[i][j] += A[i][k] * B[k][j];
                  }
           }
    }
    return C;
}
```

Tasks.

1. Provide a specification to describe the behaviour of this algorithm, and prove that it correctly implements its specification.
2. Show that matrix multiplication is associative, i.e., the order in which matrices are multiplied can be disregarded: $A(BC) = (AB)C$. To show this, you should write a program that performs the two different computations, and then prove that the result of the two computations is always the same.
3. [Optional, if time permits] In the literature, there exist many proposals for more efficient matrix multiplication algorithms. Strassen's algorithm was one of the first. The key idea of the algorithm is to use a recurisive algorithm that reduces the number of multiplications on submatrices (from 8 to 7), see https://en.wikipedia.org/wiki/Strassen_algorithm for an explanation. A relatively clean Java implementation (and Python and C++) can

Table 1. Teams participating in VerifyThis 2016 (alphabetically by tool).

#	Team members	Tool		Team attributes
1	Stephen Siegel	CIVL	[17]	
2	Rustan Leino	Dafny	[13]	
3	Christiaan Dirkx, Luca Weibel	Dafny	— ” —	student, non-developer
4	Paul Gainer, Maryam Kamali	Dafny	— ” —	non-developer
5	Gudmund Grov, Yuhui Lin	Dafny	— ” —	non-developer
6	Henning Günther, Alfons Laarman	Dafny	— ” —	non-developer
7	Mihai Herda, Michael Kirsten	KeY	[1]	student
8	Gidon Ernst	KIV	[6]	student
9	Jan Friso Groote	mCRL2	[4]	
10	Malte Schwerhoff, Alexander J. Summers	Viper	[15]	
11	Stefan Blom, Wytse Oortwijn	VerCors	[2]	
12	Bart Jacobs	VeriFast	[16]	
13	Martin Clochard	Why3	[7]	student
14	Léon Gondelman, Mário Pereira	Why3	— ” —	student

be found here: https://martin-thoma.com/strassen-algorithm-in-python-java-cpp/.
Prove that the naive algorithm above has the same behaviour as Strassen's algorithm. Proving it for a restricted case, like a 2x2 matrix should be straightforward, the challenge is to prove it for arbitrary matrices with size 2^n.

2.2 Comments on Solutions

12 teams submitted a (partial) solution to the first task of this challenge, 6 addressed the second task, and only one team attempted the optional third task during the competition.

One difficulty in the first task was the unusual order of the nested loops in the provided code. Several teams sidestepped this difficulty be rewriting the code. Two teams (Rustan Leino with Dafny; Léon Gondelman and Mário Pereira with Why3) proved functional correctness and safety of the original code. Two further teams (Bart Jacobs with VeriFast; Martin Clochard with Why3) verified the altered code, where the inner loops were swapped. Two teams (Stephen Siegel with CIVL; Jan Friso Groote with mCRL2) applied bounded model checking to verify the code for matrices up to size 16 and 1[1], resp. The remaining teams submitted partial solutions, typically consisting of a specification and incomplete proofs.

Different representations for matrices were used. Gidon Ernst (KIV) used functions to represent matrices, and did not consider array bounds. Most teams used matrices, or arrays of arrays, but Malte Schwerhoff and Alexander J. Summers (Viper) used a single-dimensional encoding of arrays. The verification effort was significantly influenced by the level of support the verification tool provided for arrays; if this is not sufficiently automated, this leads to a significant overhead in the verification process. It was also remarkable that the use of access permissions in the specification language (as in VerCors, Viper and VeriFast) often resulted in a significant verification overhead. At the end of the allocated time, both the VerCors and Viper team had more-or-less finished to prove that all required access permissions were available, but had not looked at the functional properties yet.

In parellel to work on the first task, Stefan Blom and Wytse Oortwijn used Dafny to prove associativity (subtask 2). Stephen Siegel was able to verify the property for matrices up to size 16 using CIVL, essentially by writing an explicit program test, and providing this to the symbolic execution engine. He was the only participant to address the third task, using the same approach.

After the competition, the two Why3 team's combined forces, and proved correctness of the Strassen algorithm.

3 Challenge 2: Binary Tree Traversal

This challenge was prepared by Radu Grigore, University of Kent.

3.1 Verification Task

Consider a binary tree:

```
class Tree {
   Tree left, right, parent;
   bool mark;
 }
```

[1] Using mCRL2 on larger matrices did not work, due to a bug discovered during the competition.

We are given a binary tree with the following properties:

– It is well formed, in the sense that following a child pointer (`left` or `right`) and then following a `parent` pointer brings us to the original node. Moreover, the `parent` pointer of the root is null.
– It has at least one node, and each node has 0 or 2 children.

We do not know the initial value of the `mark` fields.

Our goal is to set all `mark` fields to true. The algorithm below (Morris 1979) works in time linear in the number of nodes, as usual, but uses only a constant amount of extra space.

```
void markTree(Tree root) {
    Tree x, y;
    x = root;
    do {
          x.mark = true;
          if (x.left == null && x.right == null) {
                  y = x.parent;
          } else {
                y = x.left;
                x.left = x.right;
                x.right = x.parent;
                x.parent = y;
          }
          x = y;
    } while (x != null);
  }
```

Tasks. Prove that:

1. upon termination of the algorithm, all `mark` fields are set
2. the tree shape does not change
3. the code does not crash, and
4. the code terminates.

As a bonus, prove that the nodes are visited in depth-first order.

3.2 Comments on Solutions

11 teams submitted a solution to this challenge, but none of the solutions was complete. The most comprehensive solution by far was developed by Bart Jacobs with VeriFast; he proved all required properties except termination, which he proved a little later. The solution provides a good insight into the workings of the algorithm by making the conceptual stack, which is encoded implicitly via the flipped pointers, explicit in the specification.

Other noteworthy solutions include Malte Schwerhoff and Alexander Summers's solution in Viper, which proves the preservation of the tree and memory safety (tasks 2 and 3), as well as Mihai Herda and Michael Kirsten's solution in KeY and Léon Gondelman and Mário Pereira's solution with Why3, both of which prove memory safety. Jan Friso Groote encoded the `markTree` procedure in mCRL2, such that the tool can generate the state space for every finite tree, which implicitly proves termination. He did not define properties to capture memory safety yet.

We were not able to identify a particular reason for the fact that no solution was complete, although it seems that proving termination for most teams was complicated. The challenge was certainly very ambitious for a 90-minute slot. Besides time, the different teams struggled with very different problems and limitations of the tools they used.

4 Challenge 3: Static Tree Barriers

This challenge is inspired by a paper by Banerjee and Malkis [14].

4.1 Verification Task

This challenge focuses on multi-threaded executions and targets verifiers for concurrent programs. However, the problem can also be encoded as a transition system and verified by a verifier for sequential code.

Consider a multi-threaded program execution. In each state with N threads, we assume we have given a binary tree with N nodes. Each node corresponds to exactly one thread. In the context of this challenge, we do not consider thread creation and termination; the number of nodes and their position in the tree is assumed to be immutable. Each node has references to at most two children, and each node except for the root has a reference to its parent. Moreover, each node stores a boolean value sense and an integer version. Hence, we have the following data structure:

```
  class Node {
      final Node left, right;
      final Node parent;

      boolean sense;
      int version;

      // methods and constructors omitted
  }
```

The tree described above represents a tree barrier that can be used for thread synchronization. This synchronization is performed by the following method of class Node:

```
void barrier()
    requires !sense
    ensures  !sense
  {
    // synchronization phase
    if(left != null)
      while(!left.sense) { }

    if(right != null)
      while(!right.sense) { }


    // assume that the following two
    // statements are executed atomically
    sense = true;
    version++;

    // wake-up phase
    if(parent == null)
      sense = false;

    while(sense) { }

    if(left != null)
      left.sense = false

    if(right != null)
      right.sense = false
}
```

Synchronization is performed in two phases. In the synchronization phase, each thread that calls `barrier()` on its node waits until all threads have called barrier. The `sense` field is used to propagate information about waiting threads up in the tree. When all threads have called `barrier()`, the propagation reaches the root of the tree and initiates the wake-up phase, which proceeds top-down.

Assume a state in which `sense` is false and `version` is zero in all nodes. Assume further that no thread is currently executing `barrier()` and that threads invoke `barrier()` only on their nodes. The number of threads (and, thus, the number of nodes in the tree) is constant, but unknown.

Tasks. Prove that:

1. the following invariant holds in all states: If `n.sense` is true for any node `n` then `m.sense` is true for all nodes `m` in the subtree rooted in `n`.
2. for any call `n.barrier()`, if the call terminates then there was a state during the execution of the method where all nodes had the same version.

Hint: The problem can be solved with a verifier for sequential programs by encoding it as a non-deterministic transition system that represents threads and their states explicitly in each state of the transition system.

4.2 Comments on Solutions

The third challenge also turned out to be difficult to solve for most participants. We received 11 solutions, most of which were rather incomplete. Again, the most comprehensive solution was developed by Bart Jacobs with VeriFast, which was completed the day after the competition. Besides the two solutions employing model checkers, the VeriFast solution was the only one to directly handle threads. The VerCors tool also can handle threads, but the VerCors team only provided a very partial solution, specifying the access permissions. All other solutions encoded the problem as transition system and employed sequential reasoning, as suggested by the problem statement.

The problem stated that was initially given to the participants did not state the necessary assumption that two assignments are executed atomically. Two teams (Stephen Siegel; Bart Jacobs) reported the issues, and the problem statement was then fixed.

5 Results and Closing Remarks

5.1 Awarded Prizes

Prizes were awarded in the following categories:

- Best team: Bart Jacobs (VeriFast)
- Best student team—awarded to two teams:
  - Martin Clochard (Why3)
  - Léon Gondelman and Mário Pereira (Why3)
- Distinguished user-assistance tool feature: Alexander J. Summers and Malte Schwerhoff (Viper) for their support of quantified permissions
- Best challenge submission: Daniel Grahl, for suggesting "Strassen's algorithm", which inspired Challenge 1: Matrix Multiplication.

The best student teams each received a 150 Euro cash prize donated by our sponsors while the best overall team received 100 Euros. Smaller prizes were also awarded for the best problem submission and the distinguished user-assistance tool feature.

5.2 Final Remarks

The VerifyThis 2016 challenges have offered a substantial degree of complexity and difficulty. We noted that most state-of-the-art verification tools provide reasonable support to reason about functional properties and arrays, but that reasoning about pointers and concurrent computations is still a challenge. We believe that built-in support for verification of pointer programs and concurrent computations such as access permissions is useful, but it does cause overhead when the access permissions are irrelevant for the problem at hand. A next challenge is to make the verification of trivial access permissions implicit, so one does not have to verify them explicitly when this is irrelevant. For program verification tools that do not provide such built-in support, the next challenge is to provide adequate libraries, in order to efficiently handle such problems.

A new edition of the VerifyThis competition will be held as part of ETAPS 2017. We are curious to see the progress on the verification tools, given the experiences of the VerifyThis 2016 competition.

Acknowledgments

The organisers would like to thank Wojciech Mostowski, Radu Grigore and Erik Poll for their feedback and support prior to and during the competition. The organisers also thank the competition's sponsors: Galois, Inc., and Microsoft Research. Their contributions helped us to support participants with travel grants, and to finance the various prizes.

References

1. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In D. Giannakopoulou and D. Kroening, editors, *6th International Conference on Verified Software: Theories, Tools and Experiments (VSTTE 2014)*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014.
2. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *19th International Symposium on Formal Methods (FM 2014)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
3. 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)*, volume 7421 of *LNCS*, pages 3–21. Springer, 2011.
4. S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mCRL2 toolset and its recent advances. In N. Piterman and S. A. Smolka, editors, *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.
5. E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
6. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2014.
7. J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
8. 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, *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, volume 873 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2012.
9. 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, *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, volume 873 of *CEUR Workshop Proceedings.* CEUR-WS.org, 2012.
10. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis 2012. *Int. J. Softw. Tools Technol. Transf.*, 17(6):647–657, Nov. 2015.
11. M. Huisman, V. Klebanov, R. Monahan, and M. Tautschnig. VerifyThis 2015. A program verification competition. *Int. J. Softw. Tools Technol. Transf.*, 2016. Accepted for publication.
12. 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, *17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
13. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
14. A. Malkis and A. Banerjee. On automation in the verification of software barriers: Experience report. *J. Autom. Reasoning*, 52(3):275–329, 2014.
15. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.
16. W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In J. Vitek, editor, *24th European Symposium on Programming (ESOP 2015)*, volume 9032 of *LNCS*, pages 158–182. Springer, 2015.
17. S. F. Siegel, M. B. Dwyer, G. Gopalakrishnan, Z. Luo, Z. Rakamaric, R. Thakur, M. Zheng, and T. K. Zirkel. CIVL: The concurrency intermediate verification language. Technical Report UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware, 2014.