VerifyThis 2019: a program verification competition

International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-021-00619-x

COMPETITIONS AND CHALLENGES

Special Issue: TOOLympics 2019

VerifyThis 2019: a program verification competition

Claire Dross[1] · Carlo A Furia[2] · Marieke Huisman[3] · Rosemary Monahan[4] · Peter Müller[5]

Accepted: 6 May 2021

Abstract
VerifyThis is a series of program verification competitions that emphasize the human aspect: participants tackle the verification of detailed behavioral properties—something that lies beyond the capabilities of fully automatic verification and requires instead human expertise to suitably encode programs, specifications, and invariants. This paper describes the 8th edition of VerifyThis, which took place at ETAPS 2019 in Prague. Thirteen teams entered the competition, which consisted of three verification challenges and spanned 2 days of work. This report analyzes how the participating teams fared on these challenges, reflects on what makes a verification challenge more or less suitable for the typical VerifyThis participants, and outlines the difficulties of comparing the work of teams using wildly different verification approaches in a competition focused on the human aspect.

Keywords functional correctness · correctness proofs · program verification · verification competition

1 The VerifyThis 2019 verification competition

VerifyThis is a series of *program verification competitions* where participants prove expressive input/output properties of small programs with complex behavior. This report describes VerifyThis 2019, which took place on April 6–7, 2019, in Prague, Czech Republic, as a 2-day event of the European Joint Conferences on Theory and Practice of Software (ETAPS 2019). It was the eighth event in the series, after the VerifyThis competitions held at FoVeOOS 2011, FM 2012, the Dagstuhl Seminar 14171 (in 2014), and ETAPS 2015–2018. The organizers of VerifyThis 2019 were also the authors of this paper—henceforth referred to as "we."

VerifyThis aims to bring together researchers and practitioners interested in formal verification, providing them with an opportunity for engaging, hands-on, and fun discussion. The results of the competition help the research community evaluate progress and assess the usability of formal verification tools in a controlled environment—which still represents, on a smaller scale, important practical aspects of the verification process.

Unlike other verification competitions that belong to the same TOOLympics (Competitions in Formal Methods) track of ETAPS, VerifyThis emphasizes verification problems that go beyond what can be proved fully automatically, and require instead human experts "in the loop." During a VerifyThis event, participating teams are given a number of verification challenges that they have to solve on-site during the time they have available using their favorite verification tools. A challenge is typically given as a natural-language description—possibly complemented with some pseudo-code or lightweight formalization—of an algorithm and its specification. Participants have to implement the algorithm in the input language of their tool of choice, formal-

✉ Carlo A Furia
furiac@usi.ch

Claire Dross
dross@adacore.com

Marieke Huisman
m.huisman@utwente.nl

Rosemary Monahan
Rosemary.Monahan@mu.ie

Peter Müller
Peter.Mueller@inf.ethz.ch

[1] AdaCore, Paris, France

[2] USI Università della Svizzera italiana, Lugano, Switzerland

[3] University of Twente, Enschede, The Netherlands

[4] Maynooth University, Maynooth, Ireland

[5] ETH Zurich, Zurich, Switzerland

Published online: 19 May 2021

ize the specification, and formally prove the correctness of the implementation against the specification. The challenge descriptions leave a lot of details open, so that participants can come up with the formalization that best fits the capabilities of their verification tool of choice. Correctness proofs usually require participants to supply additional information, such as invariants or interactive proof commands.

Following a format that consolidated over the years, VerifyThis 2019 proposed three verification challenges. During the first day of the competition, participants worked during three 90-minute slots—one for each challenge. Judging of the submitted solutions took place during the second day of the competition, when we assessed the level of correctness, completeness, and elegance of the submitted solutions. Based on this assessment, we awarded prizes to the best teams in different categories (such as overall best team, and best student teams) The awards were announced during the ETAPS lunch on Monday, April 8, 2019.

The online archive of VerifyThis

http://verifythis.ethz.ch

includes the text of all verification challenges, and the solutions submitted by the teams (typically revised and improved after the competition). Reports about previous editions of VerifyThis are also available [3,6,13,16,18–20]. The motivation and initial experiences of organizing verification competitions in the style of VerifyThis are discussed elsewhere [17,23]; a recent publication [11] draws lessons from the history of VerifyThis competitions. Finally, an extended version of the present article [10] includes more details about the challenges that we had to omit for space constraints.

1.1 Challenges

A few months before the competition, we sent out a public "Call for Problems" asking for suggestions of verification challenges that could be used during the competition. Two people submitted by the recommended deadline proposals for three problems; and one more problem proposal arrived later, close to the competition date.

We combined these proposals with other ideas in order to design three challenges suitable for the competition. Following our experience, and the suggestions of organizers of previous VerifyThis events, we looked for problems that were suitable for a 90-minute slot, and that were not too biased toward a certain kind of verification language or tool. A good challenge problem should be presented as a series of specification and verification steps of increasing difficulty; even inexperienced participants should be able to approach the first steps, whereas the last steps are reserved for those with advanced experience in the problem's domain, or that find it particularly congenial to the tools they're using. Typically, the first challenge involves an algorithm that operates on arrays or even simpler data types; the second challenge targets more complex data structures in the heap (such as trees or linked lists); and the third challenge involves concurrency.

In the end, we used one suggestion[1] collected through the "Call for Problems" as the basis of the first challenge, which involves algorithms on arrays (see Sect. 2). Another problem suggestion[2] was the basis of the second challenge, which targets the construction of binary trees from a sequence of integers (see Sect. 3). For the third challenge, we took a variant of the matrix multiplication problem (which was already used, in a different form, during VerifyThis 2016) that lends itself to a parallel implementation (see Sect. 4).

1.2 Participants

Table 1 lists the 13 teams that participated in VerifyThis 2019. Four teams consisted of a single person, whereas the majority of teams included two persons (the maximum allowed).

As it is often the case during verification competitions, the majority of participants used a tool they know very well because they have contributed to its development. However, four teams identified themselves as non-developers, as they did not directly contribute to the development of the verification tools they used during the competition.

Out of 21 participants, 11 were graduate students. Some participated with a senior colleague, while some others worked alone or with other students, making up a total of three all-student teams.

1.3 Judging

Judging took place on the competition's second day. Each team sat for a 20–30-minute interview with us, during which they went through their solutions, pointing out what they did and didn't manage to verify, and which aspects they found the most challenging.

Following the suggestions of previous organizers [11], we asked teams to fill in a questionnaire about their submitted solutions in preparation for the interview. The questionnaire asked them to explain the most important features of the implementation, specification, and verification in their solutions, such as whether the implementation diverged from the pseudo-code given in the challenge description, whether the specification included properties such as memory safety, and whether verification relied on any simplifying assumptions. The questionnaire also asked participants to reflect on the process they followed (How much human effort was involved? How long would it take to complete your solution?), and on the strengths and weaknesses of the tools they used. With the bulk of the information needed for judging

[1] Sent by Nadia Polikarpova

[2] Sent by Gidon Ernst.

Table 1 Teams participating in VerifyThis 2019, listed in order of registration

	Team name	Members	Tool	
1	Mergesort	*Quentin Garchery*	Why3	[5,14]
2	VerCors T(w/o)o	Marieke Huisman, Sebastiaan Joosten	VerCors	[1,4]
3	Bashers	Mohammad Abdulaziz, *Maximilian P L Haslbeck*	Isabelle	[26]
4	Jourdan-Mével	Jacques-Henri Jourdan, *Glen Mével*	Coq	[2,21]
5	OpenJML	David Cok	OpenJML	[8]
6	YVeTTe	Virgile Prevosto, *Virgile Robles*	Frama-C	[22]
7	The Refiners	Peter Lammich, *Simon Wimmer*	Isabelle	[24,26]
8	KIV	*Stefan Bodenmüller*, Gerhard Schellhorn	KIV	[12]
9	Sophie & Wytse	*Sophie Lathouwers*, *Wytse Oortwijn*	VerCors	[4]
10	Coinductive Sorcery	*Jasper Hugunin*	Coq	[2]
11	Heja mig	*Christian Lidström*	Frama-C	[22]
12	Eindhoven UoT	Jan Friso Groote, *Thomas Neele*	mCRL2	[7,9]
13	Viper	Alexander J. Summers	Viper	[25]

For each TEAM, the table reports its NAME, its MEMBERS, and the verification TOOL they used. A member name is in *italic* if the member is a student; and it is underlined if the member is also a developer of the tool or of some extension used in the competition

available in the questionnaire, we could focus the interviews on the aspects that the participants found the most relevant while still having basic information about all teams.

At the same time as judging was going on, participants not being interviewed were giving short presentations of their solutions to the other teams. This is another time-honored tradition of VerifyThis, which contributes more value to the event and makes it an effective forum to exchange ideas about how to do verification in practice. We briefly considered the option of merging interviews (with organizers) and presentation (to other participants), but in the end we decided that having separate sessions makes judging more effective and lets participants discuss freely with others without the pressure of the competition—although the atmosphere was generally quite relaxed!

Once the interviews were over, we discussed privately to choose the awardees. We structured our discussion around the questionnaires' information, and supplemented it with the notes taken during the interviews. Nevertheless, we did not use any fixed quantitative scoring, since VerifyThis's judging requires us to compare very different approaches and solutions to the same problems. Even criteria that are objectively defined in principle may not be directly comparable between teams; for example, correctness is relative to a specification, and hence, different ways of formalizing a specification drastically change the hardness of establishing correctness. We tried to keep an open mind toward solutions that pursued an approach very different from the one we had in mind when writing the challenges, provided the final outcome was convincing. Still, inevitably, our background, knowledge, and expectations somewhat may have biased the judging process. In the end, we were pleased by all submissions, which showed a high level of effort, and results that were often impressive—especially considering the limited available time to prepare a solution.

We awarded six prizes in four categories:

- *Best Overall Team* went to Team *refiners*
- *Best Student Teams* went to Team *mergesort* and Team *sw*
- *Most Distinguished Tool Feature* went to Team *bashers*—for a library to model concurrency in Isabelle, which they developed specifically in preparation for the competition—and to Team *vercors*—for their usage of ghost method parameters to model sparse matrices
- *Tool Used by Most Teams* went to Viper—used directly or indirectly[3] by three different teams—represented by Alexander J. Summers.

2 Challenge 1: Monotonic segments and GHC Sort

The first challenge was based on the generic sorting algorithm used in Haskell's GHC compiler.[4] The algorithm is a form of *patience sorting*.[5]

[3] VerCors uses Viper as back-end; hence, Team *viper* used it directly, and Team *vercors* and Team *sw* used it indirectly.

[4] https://hackage.haskell.org/package/base-4.12.0.0/docs/src/Data.OldList.html#sort.

[5] Named after the patience card game https://en.wikipedia.org/wiki/Patience_sorting.

```
cut := [0]  # singleton sequence with element 0
x, y := 0, 1
while y < n:   # n is the length of sequence s
   increasing := s[x] < s[y]  # in increasing segment?
   while y < n and (s[y-1] < s[y]) == increasing:
      y := y + 1
   cut.extend(y)  # extend cut by adding y to its end
   x := y
   y := x + 1
if x < n:
   cut.extend(n)
```

Fig. 1 Algorithm to compute the maximal cutpoints `cut` of sequence `s`

2.1 Challenge description

2.1.1 Part A: Monotonic segments

Given a sequence s

$$s \quad = \quad s[0]\, s[1] \ldots s[n-1] \qquad n \geq 0$$

of elements over a totally sorted domain (for example, the integers), we call **monotonic cutpoints** any indexes that cut s into segments that are monotonic: each segment's elements are all increasing or all decreasing.[6] Here are some examples of sequences with monotonic cutpoints:

SEQUENCE s	MONOTONIC CUTPOINTS	MONOTONIC SEGMENTS
1 2 3 4 5 7	0 6	1 2 3 4 5 7
1 4 7 3 3 5 9	0 3 5 7	1 4 7 | 3 3 | 5 9
6 3 4 2 5 3 7	0 2 4 6 7	6 3 | 4 2 | 5 3 | 7

In this challenge, we focus on **maximal** monotonic cutpoints, that is such that, if we extend any segment by one element, the extended segment is not monotonic anymore.

Given a sequence s, for example, stored in an array, maximal monotonic cutpoints can be computed by scanning s once while storing every index that corresponds to a change in monotonicity (from increasing to decreasing, or vice versa), as shown by the algorithm in Fig. 1.

To solve Challenge 1.A, we asked participants to carry out the following tasks.

Implementation task: Implement the algorithm in Fig. 1 to compute monotonic cutpoints of an input sequence.

Verification tasks:

1. Verify that the output sequence consists of the input sequence's *cutpoints*.
2. Verify that the cutpoints in the output are a *monotonic* sequence.
3. Strengthen the definition of monotonic cutpoints so that it requires *maximal* monotonic cutpoints, and prove that your algorithm implementation computes maximal cutpoints according to the strengthened definition.

2.1.2 Part B: GHC Sort

To sort a sequence s, **GHC Sort** works as follows:

1. Split s into monotonic segments $\sigma_1, \sigma_2, \ldots, \sigma_{m-1}$
2. Reverse every segment that is decreasing
3. Merge the segments *pairwise* in a way that preserves the order
4. If all segments have been merged into one, that is an ordered copy of s; then terminate. Otherwise, go to step 3

Merging in step 3 works like merging in Merge Sort.

To solve Challenge 1.B, we asked participants to carry out the following tasks.

Implementation task: Implement GHC Sort in your programming language of choice.

Verification tasks:

1. Write functional specifications of all procedures/functions/main steps of your implementation.
2. Verify that the implementation of *merge* returns a sequence `merged` that is **sorted**.
3. Verify that the overall sorting algorithm returns an output that is sorted.
4. Verify that the overall sorting algorithm returns an output that is a permutation of the input.

2.2 Submitted solutions

Team *openjml* and Team *refiners* submitted solutions of challenge 1 that were complete and correct. Another team got close but missed a few crucial invariants. Five teams made substantial progress but introduced some simplifying assumptions or skipped verification of maximality. And another five teams' progress was more limited, often due to a mismatch between their tools' capabilities and what was required by the challenge.

Teams did not find the definition of monotonicity hard to work with because it is asymmetric: as far as we could see, most of them encoded the property as we suggested and made it work effectively.

[6] More precisely, all strictly increasing, or nonincreasing (decreasing or equal).

However, a couple of teams were confused by mistakenly assuming a property of monotonic segments: since the condition for "decreasing" is the complement of the condition for "increasing," they concluded that increasing and decreasing segments must strictly alternate (after a decreasing segment comes an increasing one, and vice versa). This is not true in general, as shown by the example of sequence 6 3 4 2 5 3 7, which is made of 4 monotonic segments 6 3 | 4 2 | 5 3 | 7, all of them decreasing.

While we did not give a formal definition of maximality, the teams that managed to deal with this advanced property did not have trouble formalizing it. Since "extending" a segment can be generally done both on its right and on its left endpoint, teams typically expressed maximality as two separate properties: to the right and to the left. While it may be possible to prove that one follows from the other (and the definition of monotonic cutpoints), explicitly dealing with both variants was found to be preferable in practice since the invariants to prove one variant are clearly similar to those to prove the other.

3 Challenge 2: Cartesian trees

The second challenge involved the notion of Cartesian trees[7] of a sequence of integers and, in particular, dwelt on how such trees can be constructed in linear time from the sequence of all nearest smaller values[8] of the input sequence.

3.1 Challenge description

3.1.1 Part A: All nearest smaller values

For each index in a sequence of values, we define the nearest smaller value to the left, or left neighbor, as the last index among the previous indexes that contains a smaller value. There are indexes that do not have a left neighbor; for example, the first value, or the smallest value in a sequence.

We consider here an algorithm that constructs the sequence of left neighbors of all values of a sequence s. This algorithm is given in pseudo-code in Fig. 2.

As an example, consider sequence $s = 4\,7\,8\,1\,2\,3\,9\,5\,6$. The sequence of the left neighbors of s (using indexes that start from 1) is: $\texttt{left} = 0\,1\,2\,0\,4\,5\,6\,6\,8$. The left neighbor of the first value of s is 0 (denoting no valid index), since the first value in a list has no values at its left. The fourth value of s (value 1) is also 0, since 1 is the smallest value of the list.

To solve Challenge 2.A, we asked participants to carry out the following tasks:

[7] https://en.wikipedia.org/wiki/Cartesian_tree.

[8] https://en.wikipedia.org/wiki/All_nearest_smaller_values.

```
stack := []  # empty stack
for every index x in s:
    # pop values greater or equal to s[x]
    while not stack.is_empty
          and s[stack.top] >= s[x]:
        stack.pop

    if stack.is_empty:
        # x doesn't have a left neighbor
        left[x] := 0
    else:
        left[x] := stack.top

    stack.push (x)
```

Fig. 2 Algorithm to compute the sequence `left` of all left nearest smaller values of input sequence `s`. The algorithm assumes that **indexes start from** 1, and hence, it uses 0 to denote that an index has no left neighbor

Implementation task. Implement the algorithm to compute the sequence of left neighbors from an input sequence.

Verification tasks.

1. *Index*: verify that, for each index i in the input sequence s, the left neighbor of i in s is smaller than i, that is $\texttt{left}[i] < i$.
2. *Value*: verify that, for each index i in the input sequence s, if i has a left neighbor in s, then the value stored in s at the index of the left neighbor is smaller than the value stored at index i, namely, if $\texttt{left}[i]$ is a valid index of s then $s[\texttt{left}[i]] < s[i]$.
3. *Smallest*: verify that, for each index i in the input sequence s, there are no values smaller than $s[i]$ between $\texttt{left}[i]+1$ and i (included).

3.1.2 Part B: Construction of a Cartesian tree

Given a sequence s of *distinct* numbers, its unique *Cartesian tree* $CT(s)$ is the tree such that:

1. $CT(s)$ contains exactly one node per value of s.
2. When traversing $CT(s)$ in-order—that is, using a symmetric traversal: first visit the left subtree, then the node itself, and finally the right subtree—elements are encountered in the same order as s.
3. Tree $CT(s)$ has the heap property—that is, each node in the tree contains a value (not an index) bigger than its parent's.

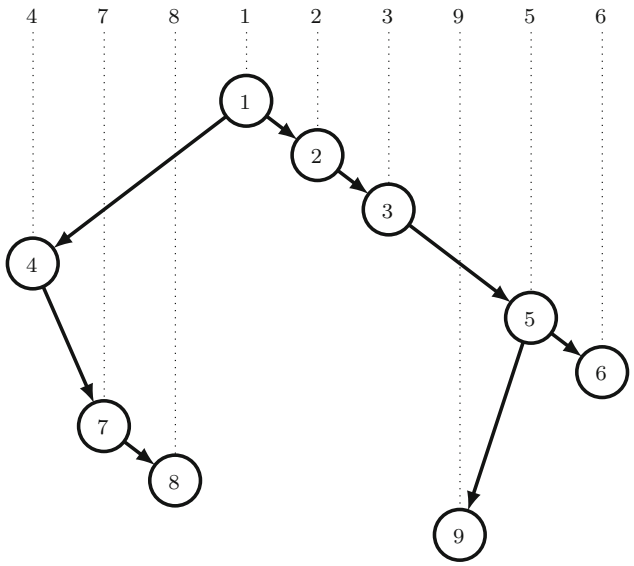

Fig. 3 Cartesian tree of sequence 4 7 8 1 2 3 9 5 6

The Cartesian tree of sequence $s = 4\,7\,8\,1\,2\,3\,9\,5\,6$ is given in Fig. 3.

To construct a Cartesian tree in linear time, we first construct the sequence of left neighbors for the value of s using the algorithm in Fig. 2. Then, we construct the sequence of right neighbors using the same algorithm, but starting from the end of the list. Thus, for every index x in sequence s, the parent of x in $CT(s)$ is either:

- The left neighbor of x if x has no right neighbor.
- The right neighbor of x if x has no left neighbor.
- If x has both a left neighbor and a right neighbor, then x's parent is the larger one.
- If x has no neighbors, then x is the root node.

To solve Challenge 2.B, we asked participants to carry out the following tasks:

Implementation task. Implement the algorithm for the construction of the Cartesian tree.
Verification tasks.

1. *Binary:* verify that the algorithm returns a well-formed binary tree, with one node per value (or per index) in the input sequence.
2. *Heap:* verify that the resulting tree has the heap property, that is, each non-root node contains a value larger than its parent.
3. *Traversal:* verify that an in-order traversal of the tree traverses values in the same order as in the input sequence.

3.2 Submitted solutions

Two teams submitted solutions to challenge 2 that were both correct and complete: Team *openjml* worked on part A of the challenge, and Team *vercors* on part B. The latter team even managed to verify a partial specification of part B's task *traversal*—which was marked "optional." Another four teams completed the first two verification tasks of part A, one of them coming close to finishing the proof of the third, with only a small part of the necessary invariant missing. Another team completed all three verification tasks of part A but with simplifying assumptions (on the finite range of inputs). Another two teams completed part A's verification task 1 only. The remaining four teams didn't go further than implementing the algorithm of the same part and writing partial specifications of the properties that were to be verified.

We presented challenge 2 under the assumption that its part A was somewhat easier and more widely feasible than part B. The fact that most teams worked on part A may seem to confirm our assumption about its relatively lower difficulty.[9] At the same time, one out of only two teams who submitted a complete and correct solution to challenge 2 tackled part B. This may just be survival bias but another plausible explanation is that the difficulties of the two parts are not so different (even though part B looks more involved).

Indeed, part A revealed some difficulties that were not obvious when we designed it. First, the algorithm in Fig. 2 follows an imperative style, and hence, it is not obvious how to encode it using functional style; various teams introduced subtle mistakes while trying to do so. Part B is easier in this respect, as the Cartesian tree construction algorithm consists of a simple iteration over the input, which manipulates data that can all be encoded indifferently using sequences, arrays, or lists. Part A, in contrast, requires a stack data structure with its operations. In the end, what really makes part B harder than part A is probably its third, optional, verification task *traversal*. Specifying it is not overly complicated, but proving it requires a complex "big" invariant, which was understandably not easy to devise in the limited time available during the competition.

[9] After the competition, Team *vercors* explained that they missed our hint that part A was simpler, and chose part B only because it looked like a different kind of challenge (as opposed to part A, which they felt was similar in kind to challenge 1's part A). In the heat of the competition, participants may miss details of the challenges that may have helped them; this is another factor that should be considered when designing a challenge.

```
y := (0, ..., 0)
for every element (r, c, v) in m:
   y (c) := y (c) + x (r) * v
```

Fig. 4 Algorithm to multiply an input vector `x` with a sparse matrix `m` and store the result in the output vector `y`. Input matrix `m` is represented in the COO format as a list of triplets

4 Challenge 3: Sparse matrix multiplication

The third challenge targeted the *parallelization* of a basic algorithm to multiply *sparse* matrices (where most values are zero).

4.1 Challenge description

We represent *sparse matrices* using the coordinate list (COO) format. In this format, nonzero values of a matrix are stored in a sequence of triplets, each containing row, column, and corresponding value. The sequence is sorted, first by row index and then by column index, for faster lookup. For example, the matrix:

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \end{pmatrix}$$

is encoded into the following sequence (using row and column indexes that start from 1):

(1, 3, 1) (2, 1, 5) (2, 2, 8) (4, 2, 3)

In this challenge, we consider an algorithm that computes the multiplication of a vector of values (encoded as a sequence) with a sparse matrix. It iterates over the values present inside the matrix, multiplies each of them by the appropriate element in the input vector, and stores the result at the appropriate index in the output vector. Figure 4 presents the algorithm in pseudo-code.

To solve challenge 3, we asked participants to carry out the following tasks:

Implementation tasks.

1. Implement the algorithm to multiply a vector x with a sparse matrix m.
2. We want to execute this algorithm in parallel, so that each computation is done by a different process, thread, or task. Add the necessary synchronization steps in your sequential program, using the synchronisation feature of your choice (lock, atomic block, …). You can choose how to allocate work to processes. For example:
   - each process computes exactly one iteration of the for loop;
   - there is a fixed number of processes, each taking an equal share of the total number of for loop iterations;
   - work is assigned to processes dynamically (for example using a work stealing algorithm).

Verification tasks.

1. Verify that the sequential multiplication algorithm indeed performs standard matrix multiplication (that is, it computes the output vector y with values $y_i = \sum_k x_k \times m_{k,i}$).
2. Verify that the concurrent algorithm does not exhibit concurrency issues (data races, deadlocks, …).
3. Verify that the concurrent algorithm still performs the same computation as the sequential algorithm. If time permits, you can also experiment with different work allocation policies and verify that they all behave correctly.

4.2 Submitted solutions

No teams solved challenge 3 completely. Six teams, out of the 12 teams[10] that took part in VerifyThis's third and final session, attempted the verification of the sequential algorithm only—usually because their tools had little or no support for concurrency; out of these six teams, one completed verification task 1. Another six teams introduced concurrency in their implementation and tried to verify the absence of concurrency issues (verification task 2). Some of these teams used tools with built-in support for the verification of concurrent algorithms, while others added concurrency to their mostly sequential tools via custom libraries. Three teams out of the six that tackled task 2 completed the verification task in time during the competition; all of them happened to use a tool with built-in support for concurrency. Finally, five teams attempted verification task 3 (proving that the sequential and concurrent algorithms compute the same output). Two of them achieved substantial progress on the proofs of task 3: Team *eindhoven* used a model checker with native support for concurrency; Team *refiners* used Isabelle—a tool without built-in support for concurrency—and hence modeled the concurrent implementation as a sequential algorithm that goes over the sparse matrix's elements in nondeterministic order.

Regardless of whether their verification tools supported concurrency, all teams had plenty of work to do in challenge 3. We wanted a challenge that was approachable by everybody, and it seems that challenge 3 achieved this goal.

[10] That is, one team skipped the last session.

On the other hand, the challenge turned out to be more time-consuming than we anticipated. The sequential and the concurrent part *alone* were enough to fill all 90 minutes of the competition session, and no team could complete the whole challenge.[11] When we designed the challenge, we did not realize how time-consuming it would be.

The multiplication algorithm is conceptually simple, but verifying it requires to fill in a lot of details, such as associativity and commutativity properties of arithmetic operations, that are not central to the algorithm's behavior but are necessary to complete the proofs. In most cases, it was these details that prevented participants from completing the challenge. Another feature that is often missing from verification tools but was required for challenge 3 is the ability of expressing sums over sets and sequences; while this can always be specified and verified, doing so takes time and distracts from the main goal of the challenge.

In all, verification challenges involving concurrency are not only harder to verify but also to design! There are so many widely different concurrency models and verification frameworks that calibrating a challenge so that it suits most of them is itself a challenge. A possible suggestion to come up with concurrency challenges in the future is to write problems with different parts that are suitable for different verification approaches. This strategy worked to ensure that tools without support for concurrency still had work to do in this challenge, and it may be generalizable to encompass different styles of concurrent programming and reasoning.

5 Discussion

We organize the discussion around three themes. Section 5.1 outlines how teams revised their solutions for publication in the months after the competition. Section 5.2 analyzes the features of the verification challenges offered over the years, and how they affect the teams' success rate. Section 5.3 mentions some lessons we learned during this year's VerifyThis, which we would like to pass on to future organizers.

5.1 Revised solutions

A couple of weeks after VerifyThis was over, we contacted all participants again, asking them permission to publish their solutions online. Teams who consented had the choice of either publishing the solutions they submitted during the competition or supplying revised solutions—which they could prepare with substantially more time and the benefit of hindsight. Nine teams submitted revised solutions—from light revisions to significant extensions. Among the former, Team *jm* and Team *openjml* cleaned up their code, added comments, and improved a few aspects of the implementation or specification to make them more readable. Team *yvette* thoroughly revised their solutions and filled in missing parts of specification and proofs, so as to complete parts A of challenges 1 and 2, and the sequential part of challenge 3. Team *kiv* and Team *viper* went further, as they also completed the concurrent part of challenge 3. So did Team *vercors*, Team *sw*,[12] and Team *refiners* who also provided partial solutions for part B of challenge 2. Team *mergesort* submitted extensively revised solutions, including the only complete solution to challenge 2's part B—relying on a Coq proof of task *traversal*[13]—and the sequential part of challenge 3.

The process of revising and extending solutions *after* the competition is very different from that of developing them from scratch *during* it. With virtually unlimited time at their disposal, and the freedom to explore different approaches even if they may not pan out in the end, every team could in principle come up with a complete and correct solution. At the same time, comparing the post-competition work of different teams is not very meaningful since some may simply not have additional time to devote to improving solutions—after all, we made it clear that revising solutions was something entirely optional that they did not commit to when they signed up for the competition.

5.2 What makes a challenge difficult?

We used various criteria to classify the 21 challenges used at VerifyThis to date—three in each edition excluding VerifyThis 2014, which was run a bit differently among participants to a Dagstuhl Seminar. We classified each challenge according to which VerifyThis *competition* it belongs to, whether it appeared first, second, or third in *order* of competition, how much *time* was given to solve it, whether it targets a *sequential* or concurrent algorithm, what kind of *input* data it processes (array, tree, linked list, and so on), whether the main algorithm's *output* involves the same kind of data structure as the input, whether the challenge's main *algorithm* is iterative or recursive (or if the algorithm is only outlined), and whether the input data structure is mutable or immutable. For each challenge, we also record what percentage of participating teams managed to submit a partial or complete correct solution. Table 2 shows the results of this classification.

[11] Using a model checker, Team *eindhoven* covered all verification tasks but relied on simplifying assumptions on input size and number of processes.

[12] Team *vercors* and Team *sw* worked together to prepare one revised solution that merged both teams' independent work during the competition.

[13] The proof obligation was generated automatically by Why3, but the Coq proof steps were supplied manually.

Table 2 For each challenge PROBLEM used at VerifyThis: the COMPETITION when it was used; the ORDER in which it appeared; how much TIME (in minutes) was given to participants to solve it; whether the main algorithm is SEQUENTIAL or concurrent; the main INPUT data type; whether the OUTPUT data type is of the *same* kind as the input, *simpler*, or more *complex*; when the ALGORITHM was given in pseudo-code, whether it was *iterative* or *recursive* (if it was not given, whether it was *outlined* or participants had to *find* it based on the requirements); whether the input is MUTABLE or immutable; and the percentages of participating teams that were able to submit a partial or COMPLETE solution

Problem	Competition	Order	Time	Sequential	Input	Output	Algorithm	Mutable	Partial	Complete
Maximum by elimination	VT11	1	60	Sequential	Array	Simple	Iterative	Immutable	83	67
Tree maximum	VT11	2	90	Sequential	Tree	Simple	Outlined	Immutable	100	17
Find duplets in array	VT11	3	90	Sequential	Array	Simple	Find	Immutable	83	50
Longest common prefix	VT12	1	45	Sequential	Array	simple	Iterative	Immutable	100	73
Prefix sum	VT12	2	90	Sequential	Array	Complex	Outlined	Immutable	73	9
Delete min node in binary search tree	VT12	3	90	Sequential	Tree	Same	Recursive	Mutable	18	0
Relaxed prefix	VT15	1	60	Sequential	Array	Same	Iterative	Immutable	79	7
Parallel GCD by subtraction	VT15	2	60	Concurrent	Scalar	Same	iterative	mutable	79	14
Doubly linked lists	VT15	3	90	Sequential	Linked list	Same	Outlined	Mutable	71	7
Matrix multiplication	VT16	1	90	Sequential	Matrix	Same	Iterative	Immutable	86	43
Binary tree traversal	VT16	2	90	Sequential	Tree	Simple	Iterative	Mutable	79	7
Static tree barriers	VT16	3	90	concurrent	Tree	Simple	Iterative	Mutable	79	7
Pair insertion sort	VT17	1	90	Sequential	Array	Same	Iterative	Mutable	100	10
Odd–even transposition sort	VT17	3	90	Concurrent	Array	Same	Iterative	Mutable	60	0
Tree buffer	VT17	4	90	Sequential	Tree	Same	Recursive	Immutable	40	20
Gap buffer	VT18	1	60	Sequential	Array	Same	Iterative	Mutable	91	36
Count colored tiles	VT18	2	90	Sequential	Array	Same	Recursive	Immutable	55	18
Array-based queue lock	VT18	3	90	Concurrent	Array	Simple	Iterative	Mutable	27	9
Monotonic segments and GCG sort	VT19	1	90	Sequential	Array	Same	iterative	Immutable	85	15
Cartesian trees	VT19	2	90	Sequential	Array	Complex	Iterative	Immutable	69	15
Sparse matrix multiplication	VT19	3	90	Concurrent	Matrix	Same	Iterative	Immutable	85	0

To help us understand which factors affect the complexity of a verification problem, we fit a linear regression model (with normal error function) that uses *competition*, *order*, *time*, *sequential*, *input*, *output*, *algorithm*, and *mutable* as predictors, and the percentage of *complete* solutions as outcome.[14] Using standard practices [15], categorical predictors that can take n different values are encoded as $n - 1$ binary indicator variables—each selecting a possible discrete value for the predictor. Fitting a linear regression model provides, for each predictor, a regression coefficient estimate and a standard error of the estimate; the value of the predictor has a definite effect on the outcome if the corresponding coefficient estimate differs from zero by at least two standard errors.

Our analysis suggests that the competition challenges were somewhat simpler in the early editions compared to the recent editions (starting from VerifyThis 2015): the coefficients for indicator variables related to predictor *competition* for the years 2015–2017 and 2019 are clearly negative, indicating that belonging to one of these editions tends to decrease the number of correct solutions. Similarly, the later a challenge problem appears in a competition the fewer teams manage to solve it correctly. This is to be expected, as the first challenge is normally the simpler and more widely accessible one, and participants get tired as a competition stretches over several hours.

When a challenge's main algorithm is only outlined, or is given in pseudo-code but is recursive, and when the input is a mutable data structure, participants found it harder to complete a correct solution. While the difficulty of dealing with mutable input is well known—and a key challenge of formal verification—different reasons may be behind the impact of dealing with naturally recursive algorithms. One interpretation is that verification tools are still primarily geared toward iterative algorithms; a different, but related, interpretation is that VerifyThis organizers are better at gauging the complexity of verifying iterative algorithms, as opposed to that of recursive algorithms that may be easy to present but hard to prove correct.

Sequential algorithms, as opposed to concurrent ones, are associated with harder problems. Since the association is not

[14] We could also perform a similar analysis using the percentage of *partial* solutions as outcome. However, what counts as "partially correct" is a matter of degree and depends on a more subjective judgment—which would risk making the analysis invalid.

very strong, it is possible that this is only a fluke of the analysis: sequential algorithms are the vast majority (76%) of challenges and thus span a wide range of difficulties; the few challenges involving concurrent algorithms have often been presented in a way that they offer a simpler, sequential variant (see for example challenge 3 in Sect. 4)—which may be what most teams go for.

The input data structure also correlates with the ease of verification. Unsurprisingly, when the algorithm's input is a scalar more teams are successful; but, somewhat unexpectedly, success increases also when the input is a linked list or a tree. It is possible that the organizers are well aware of the difficulty of dealing with heap-allocated data structures, and hence stick to relatively simple algorithms when using them in a verification challenge. Another possibility is that linked lists and trees just featured a few times (compared to the ubiquitous arrays), and hence, their impact is more of a statistical fluke. Input in matrix form is associated with harder problems too; this is probably because most verification tools have no built-in matrix data types, and representing bidimensional data using arrays or lists is possible in principle but may be cumbersome.

5.3 Lessons learned for future competitions

Most verification tools are somewhat specialized in the kinds of properties and programs they mainly target; such specialization normally comes with a cost in terms of less flexibility when tackling challenges outside their purview. VerifyThis organizers try to select challenges that target different domains and properties, so that no participants will be exclusively advantaged. However, this may also indicate that it may be interesting to see the participation of teams using *different approaches*. While only one team has used two different verification tools in the history of VerifyThis, teams using verification frameworks that integrate different libraries and features effectively have at their disposal a variety of approaches. For instance, Team *refiners* used a refinement library for Isabelle only in one challenge, whereas they stuck to Isabelle's mainstream features for the rest of the competition. In order to promote eclectic approaches to verification, organizers of future events may introduce a new award category that rewards the teams that displayed the widest variety of approaches during the competition.

VerifyThis challenges are made publicly available after the competition every year, and several team members took part in more than one competition. Therefore, the most competitive and ambitious teams are aware of the kinds of problems that will be presented, and may be better prepared to solve them in the limited time at their disposal. We have evidence of at least one team that went one step further preparing for the competition this year: Team *bashers* created an Isabelle library to reason about concurrency, expecting a challenge of the same flavor as those given in recent years. These observations may give new ideas to organizers of future events to design verification challenges that are interesting but also feasible. For example, they could announce before the competition (in the call for participation) some topics that will appear in the verification challenges, or some program features that participants will be expected to deal with—but without mentioning specific algorithms or problems. Researchers and practitioners interested in participating may then use this information to focus their preparation.

Following the recurring suggestions of previous organizers, we used a questionnaire to help compare solutions and judge them. This was of great help and we hope future organizers can improve this practice even further. While our questionnaire was primarily made of open questions and collected qualitative data, it may be interesting to complement it with *quantitative* information about the challenges and the solutions—such as the size of specifications, implementation, and other tool-specific annotations. Collecting such information consistently year after year could also pave the way for more insightful analyses of the trends in the evolution of verification technology as seen through the lens of verification competitions (perhaps along the lines of what we did in Sect. 5.2).

We are always pleased to see how committed participants are, and how much effort they put into their work during and after the competition. One sign of this commitment is that most teams (see Sect. 5.1 for details) were available to substantially *revise* their solutions during the weeks and months after the competition, so that we could publish a complete solution that shows the full extent of the capabilities of their tools. It may be interesting to find ways to give more visibility to such additional work—for example, publishing post-proceedings where teams can describe in detail their work and how it was perfected. Since not all participants may be able to commit to such an extra amount of work, this may be organized only occasionally, and contributing to it should be on a voluntary basis.

Acknowledgements We thank Amazon Web Services for sponsoring travel grants and monetary prizes; Gidon Ernst, Nadia Polikarpova, and Stephen F. Siegel for submitting ideas for competition problems; and Virgile Prevosto for accepting, on a short notice, our invitation to give a tutorial. The extensive support by local organizers of ETAPS was fundamental for the practical success of VerifyThis. Being part of TOOLympics further enhanced the interest and variety of the participants' experience, and made VerifyThis more widely visible. Finally, our biggest "thank you" goes to the participants for their enthusiastic participation, for their effort (often continued after the competition when preparing revised solutions), and for contributing to interesting discussions in an amiable atmosphere.

Funding Open Access funding provided by Università della Svizzera italiana.

References

1. Amighi, A., Blom, S., Huisman, M.: VerCors: A layered approach to practical verification of concurrent software. In: IEEE Computer Society 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 495–503. (2016). https://ieeexplore.ieee.org/abstract/document/7445381
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer, New York (2004)
3. Beyer, D., Huisman, M., Klebanov, V., Monahan, R.: Evaluating software verification systems: benchmarks and competitions (Dagstuhl Reports 14171). Dagstuhl Rep. **4**(4), 1–19 (2014)
4. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: 13th International Conference on Integrated Formal Methods (IFM), Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017)
5. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, pp. 53–64. Wrocław, Poland (2011). https://hal.inria.fr/hal-00790310
6. Bormer, T., Brockschmidt, M., Distefano, D., Ernst, G., Filliâtre, J., Grigore, R., Huisman, M., Klebanov, V., Marché, C., Monahan, R., Mostowski, W., Polikarpova, N., Scheben, C., Schellhorn, G., Tofan, B., Tschannen, J., Ulbrich, M.: The COST IC0701 verification competition 2011. In: Formal Verification of Object-Oriented Software (FoVeOOS), Lecture Notes in Computer Science, vol. 7421, pp. 3–21. Springer (2012)
7. Bunte, O., Groote, J.F., Keiren, J.J.A., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.C.: The mCRL2 toolset for analysing concurrent systems – improvements in expressivity and usability. In: Tools and Algorithms for the Construction and Analysis of Systems (Part II), Lecture Notes in Computer Science, vol. 11428, pp. 21–39. Springer (2019)
8. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: Proceedings 1st Workshop on Formal Integrated Development Environment (F-IDE), EPTCS, vol. 149, pp. 79–92 (2014)
9. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7795, pp. 199–213. Springer (2013)
10. Dross, C., Furia, C.A., Huisman, M., Monahan, R., Müller, P.: VerifyThis 2019: A program verification competition (extended report). arXiv:2008.13610 (2020)
11. Ernst, G., Huisman, M., Mostowski, W., Ulbrich, M.: VerifyThis – verification competition with a human factor. In: Tools and Algorithms for the Construction and Analysis of Systems – 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Lecture Notes in Computer Science, vol. 11429, pp. 176–195. Springer (2019)
12. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: Overview and VerifyThis competition. STTT **17**(6), 677–694 (2015)
13. Filliâtre, J., Paskevich, A., Stump, A.: The 2nd verified software competition: Experience report. In: Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, CEUR Workshop Proceedings, vol. 873, pp. 36–49. CEUR-WS.org (2012)
14. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming (ESOP), Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (2013)
15. Gelman, A., Hill, J.: Data Analysis Using Regression and Multilevel/hierarchical Models. Cambridge University Press, Cambridge (2007)
16. Huisman, M., Klebanov, V., Monahan, R., Tautschnig, M.: VerifyThis 2015: a program verification competition. Int. J. Softw. Tools Technol. Transfer **19**(6), 763–771 (2017)
17. Huisman, M., Klebanov, V., Monahan, R.: On the organisation of program verification competitions. In: Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, CEUR Workshop Proceedings, vol. 873, pp. 50–59. CEUR-WS.org (2012)
18. Huisman, M., Monahan, R., Müller, P., Paskevich, A., Ernst, G.: VerifyThis 2018: A program verification competition. https://hal.inria.fr/hal-01981937/ (2018). HAL Id: hal-01981937, version 1
19. Huisman, M., Monahan, R., Müller, P., Poll, E.: VerifyThis 2016: A program verification competition. http://hdl.handle.net/2066/161349 (2016). CTIT technical report 1381-3625
20. Huisman, M., Monahan, R., Müller, P., Mostowski, W., Ulbrich, M.: VerifyThis 2017: A program verification competition. Tech. Rep. 10, Karlsruher Institut für Technologie (KIT) (2017)
21. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28**, (2018)
22. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015)
23. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M.A., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st verified software competition: Experience report. In: FM 2011: Formal Methods, Lecture Notes in Computer Science, vol. 6664, pp. 154–168. Springer (2011)
24. Lammich, P.: Refinement to imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2019)
25. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation (VMCAI), Lecture Notes in Computer Science, vol. 9583, pp. 41–62. Springer (2016)
26. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle framework. In: 21st International Conference on Theorem Proving in Higher Order Logics (TPHOL), Lecture Notes in Computer Science, vol. 5170, pp. 33–38. Springer (2008)