Noname manuscript No.
(will be inserted by the editor)

VerifyThis 2022 Program Verification Competition

Marie Farrell · Peter Lammich · Marieke Huismann · Rosemary Monahan · Peter Müller · Mattias Ullbrich

Received: date / Accepted: date

Abstract VerifyThis 2022 was a two-day program verification competition that was ran as part of the European Joint Conferences on Theory and Practice of Software (ETAPS) on the 2^{nd} and 3^{rd} of April 2022 in Munich, Germany. This paper provides an overview of the competition, challenges and results. We also examine the solutions to the challenges and reflect on lessons learned.

Keywords Verification · Formal Methods · Competition

1 Introduction

VerifyThis is a series of program verification competitions [12,13,15,14,8]. VerifyThis 2022 took place on the 2^{nd} and 3^{rd} of April, 2022 in Munich, Germany. It was the 10^{th} edition in the VerifyThis series and was held as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2022). This was the first in-person edition of the competition after the COVID-19 pandemic restrictions were lifted. The previous competition was held completely virtually in 2021[1].

The aims of the competition have been and continue to be [14,8,15]:

- 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.

Challenges in VerifyThis competitions are usually given as pseudocode with an accompanying list of verification tasks that typically focus on the input/output behaviour of programs. Competitors are provided with a specified time limit in which to complete each challenge. Within this predefined time limit, they must specify/implement the problem in their chosen verification tool and then try to verify as many of the verification tasks as possible for each challenge. There are no restrictions on the verification tools/techniques used and the challenges for each competition are available on the VerifyThis website[2]. Solutions are judged on the basis of correctness, completeness and elegance.

We began VerifyThis 2022 with a short *Team Introductions* session where each team introduced themselves, their background and the tools that they planned to use in the competition. After this, VerifyThis 2022 participants were provided with three challenges and given two hours to solve each challenge. Initially, we planned for the competition to follow a fully in-person format. However, a couple of days before the competition we allowed those already registered but unable to attend in-person (due to COVID-19 related issues) to attend virtually. There were roughly 3 virtual participants, one of which competed alongside an in-person team mate. Some were present and competed in the

We extend our sincere thanks to Amazon Web Services for their generous sponsorship of travel grants and prizes at VerifyThis 2022.

Marie Farrell
Department of Computer Science, The University of Manchester, UK
E-mail: marie.farrell@manchester.ac.uk

[1] `https://www.pm.inf.ethz.ch/research/verifythis/Archive/20191.html`

[2] `https://www.pm.inf.ethz.ch/research/verifythis.html`

competition, others joined for the tutorial and team presentation sessions.

At VerifyThis 2022, 17 teams submitted solutions to the challenges. There were approximately 3 other teams that participated but that did not submit any solutions. We summarise the teams and list the verification tools that they used in Table 1. Teams submitted their solutions at the end of each challenge timeslot. They also submitted a self assessment form[3] at the end of the competition that was used to judge the solutions.

Michael Sammler was invited to give a tutorial on the RefinedC tool [18] during Day 2 of VerifyThis whilst judging took place. Judging involved interviews between the organisers and each individual team where we used the information in the self assessment form as well as the conversations during interviews to assess the solutions. Once the tutorial was over, each team presented their solution(s) to the other teams again in parallel with judging.

The following sections describe each of the challenges and summarise the solutions that were submitted. Each challenge was accompanied by a problem description, pseudocode and a list of verification tasks.

2 Challenge 1: Downsampling a Point Cloud

This challenge was directly derived from experience in verifying a real-world use case of autonomous grasping for active debris removal in space [11,10]. This algorithm is a classical algorithm that is used for reducing the size of an input image to ease further processing. This type of functionality is employed in many real-world/industrial applications of robotics and vision systems. As such, it is widely available in libraries for main stream programming languages such as Python. We provided competitors with a simplified version which they were free to further simplify as required to suit their chosen verification tool(s).

```
datatype Point = Point(x:real,y:real,z:real)
method downSample(p:list<Point>, voxel_size:real)
  returns (pd:list<Point>)
{
  // Get max and min x, y and z of point cloud
  x_max := max{pt.x | pt ∈ p};
  x_min := min{pt.x | pt ∈ p};
  y_max := max{pt.y | pt ∈ p};
  y_min := min{pt.y | pt ∈ p};
  z_max := max{pt.z | pt ∈ p};
  z_min := min{pt.z | pt ∈ p};

  // Find the number of voxels in all 3 dimensions.
  //A voxel is a cube with edge length voxel_size.
  //Note that we round up in the division.
  var num_vox_x := (|x_max-x_min|/voxel_size).Ceiling;
  var num_vox_y := (|y_max-y_min|/voxel_size).Ceiling;
  var num_vox_z := (|z_max-z_min|/voxel_size).Ceiling;

  // Array of voxels, each element
  // e.g. at voxel_array[i,j,k] is a Point which is
  //initialised to (0.0, 0.0, 0.0)
  voxel_array
      := new Point[num_vox_x,num_vox_y,num_vox_z];

  // Array of counts in all dimensions (useful for
  // averaging), each element should be set to 0 at
  // initialisation, at the end the sum of counts
  // should be equal to the number of points in
  //the input point cloud.
  count_array
      := new int[num_vox_x,num_vox_y,num_vox_z];

  // Objective is to calculate:
  // E.g: voxel_array[0,2,1]
  //     -> (((0.23, 2.45, 1.89) + (0.13, 2.87, 1.35)
  //      + ..())/count_array[0,2,1])

  forall pt in p {
    // take the floor to collect points that are in
    // the same region
    var x_floored := ((pt.x-x_min)/voxel_size).Floor;
    var y_floored := ((pt.y-y_min)/voxel_size).Floor;
    var z_floored := ((pt.z-z_min)/voxel_size).Floor;

    voxel_array[x_floored,y_floored,z_floored]
      := voxel_array[x_floored,y_floored,z_floored]+pt;
    count_array[x_floored,y_floored,z_floored]
      := count_array[x_floored,y_floored,z_floored]+1;
  }

  // Average the voxelised (bucketed) points to get
  // the final point cloud
  i, j, k := 0, 0, 0;
  pd := [];
  for 0 ≤ i < num_vox_x
    for 0 ≤ j < num_vox_y
      for 0 ≤ k < num_vox_z
        if(count_array[i,j,k] ≠0)
          pd.append(voxel_array[i,j,k]
                    /count_array[i,j,k]);
  return pd;
}
```

Fig. 1 Pseudocode for Challenge 1: Downsampling a Point Cloud.

2.1 Problem Description

The algorithm shown in Fig. 1 describes the functionality of downsampling an input point cloud. Downsampling is used to reduce the size of an input image before it is processed further. The resulting point cloud retains the overall geometric structure but has a reduced number of points. Techniques such as this are common in domains such as signal/image processing and robotics. Our pseudocode uses square voxel downsampling with a predefined voxel size. That is, the space is tiled into cube-shaped voxels, and the average point in the voxel becomes the new point to replace the others. The pseudocode (Fig. 1) uses real numbers but participants may simplify and use integers if their tool does not have adequate support for real numbers. Other potential simplifications include requiring that the least point is already at $(0, 0, 0)$ and assuming fixed parameters for `x_max`, `y_max` and `z_max`.

[3] `https://mariefarrell.github.io/verifythis/self_assessment.txt`

Team Name	Team Members	Student Team	Tool(s) Used
VerCors Yellow	Robert Rubbens & Philip Tashce	Yes	VerCors [4]
VerCors Blue	Lukas Armborst & Pieter Bos	No	VerCors [4]
VerCors Red	Omer Sakar & Raul Monti	No	VerCors [4]
Frama-C Veterans	Lionel Blatter & Jean-Christophe Lechenet	No	Frama-C [6] and Coq [5]
Team KIV	Stefan Bodenmuller & Martin Bitterlich	Yes	KIV [3]
Acid Jazz	Jonas Fiala & Thibault Dardinier	Yes	Viper [17]
Team 3	Vytautas Astrauskas and Marco Eilers	Yes	Viper [17] and Nagini [9]
Jeroen	Jeroen Dijkhuizen	No	Prusti [2]
rc::trust_me	Michael Sammler & Paul Zhu	Yes	Refined C [18]
SUPAERO	Baptiste Pollien & Lelio Brun	No	Frama-C [6]
The Superconders	Linard Arquint & Joao Pereira	Yes	Gobra [21]
Soundproof	Aurel Bily & Felix Wolf	Yes	Viper [17]
The Blacksmiths	Xavier Denis	Yes	Creusot [7]
KeY Team 2	Julian Wiesler & Alicia Appelhagen	Yes	KeY [1]
assert(false)	Noe De Santo & Mario Bucev	No	Stainless [16]
Team CIVL	Stephen Siegel	No	CIVL [20]
KeY Team 1	Wolfram Pfeifer	Yes	KeY [1]

Table 1 This table summarises the teams that competed in VerifyThis 2022 and lists the verification tool(s) that each team used. We also distinguish teams that were fully composed of students (10 in total) and those that were not (7 in total).

2.2 Verification tasks

We provided five verification tasks for competitors to attempt. Specifically, we asked them to verify the following properties:

1. Memory Safety.
2. Termination.
3. The output point cloud is smaller or equal to the input point cloud. For example:
$$size(pd) <= size(p)$$
4. The output point cloud is within the same range as the input point cloud. For example:
$$boundingbox(pd) \ \ inside \ \ boundingbox(p)$$
5. The output point cloud is a correct downsampled version of the input point cloud.

2.3 Solutions

This challenge had 5 verification tasks, the highest score received on this challenge was 2/5 and was achieved by three teams (The Superconders, KeY Team 2 and Frama-C Veterans). These teams primarily focused on verifying memory safety and termination for this challenge, some of them managed to specify but not verify some of the other properties. No team was able to verify the correctness tasks 4 and 5 for this challenge. The scoring scheme is discussed in detail in Section 5.1.

3 Challenge 2: Mergesort With Runs

The second challenge focused on a sorting algorithm that is based on mergesort. This algorithm is recursive and is shown in Fig. 2.

3.1 Problem Description

The following mergesort-based algorithm sorts the elements of a list or array, and, at the same time, computes the indexes where runs of equal elements end. The program operates on a type T with a *weak partial ordering* $\leq$, i.e., there exists a mapping $f : T \to L$, such that L is linearly ordered and $t_1 \leq t_2$ iff $f(t_1) \leq f(t_2)$. Moreover, we assume an unsigned integer type `size_t` that is large enough to hold array indexes, and a type `array` that supports indexing `a[i]` and pushing elements to the back `a.push_back`, extending the array's size. Arrays are always initialized as empty with length zero. The pseudocode corresponding to Challenge 2 is shown in Fig. 2.

We asked the competitors to implement the `merge` and `msort` functions. They were also asked to implement the `array`, `T`, and `size_t` types in any way that fits the tool that they were using. If their tool could not handle fixed bit-width types, they could use arbitrary precision integers for `size_t`. If their tool could not handle generic types or type-classes, they could fix `T` to be some concrete type, e.g., integer. An example of the input and output of this algorithm is shown below.

```
msort([5,4,5,3,9,3],0,6) = {
  // end indexes or runs (exclusive)
  runs: [    2,3,  5,6]
  //sorted input
  data: [3,3,4,5,5,9]
}
```

3.2 Verification tasks

There were 8 verification tasks posed for this challenge. Specifically, participants should verify:

```
// Structure to store sorted array and end
// indexes of runs
struct sr {
  // End indexes of runs (exclusive)
  array<size_t> runs;
  array<T> data; // Data
}

// Merge r_1 and r_2
sr merge(sr r_1, sr r_2) {
  sr res;

  // Current positions in data arrays
  size_t di_1=0; size_t di_2=0;
  // Current positions in runs
  size_t ri_1=0; size_t ri_2=0;

  while (ri_1 < r_1.runs.length
         or ri_2 < r_2.runs.length) {
    // Check if we have to take data from first
    // and/or second input array
    bool t_1 = ri_1 < r_1.runs.length
               and (ri_2 = r_2.runs.length
                 or r_1.data[di_1] ≤ r_2.data[di_2]);
    bool t_2 = ri_2 < r_2.runs.length
               and (ri_1 = r_1.runs.length
                 or r_2.data[di_2] ≤ r_1.data[di_1]);

    if (t_1) { // Copy data from first input array
      for (;di_1 < r_1.runs[ri_1]; ++di_1)
        res.data.push_back(r_1.data[di_1]);
      ++ri_1;
    }

    if (t_2) { // Copy data from second input array
      for (;di_2 < r_2.runs[ri_2]; ++di_2)
        res.data.push_back(r_2.data[di_2]);
      ++ri_2;
    }

    // Add new segment boundary
    res.runs.push_back(res.data.size());
  }
  return res;
}

// Mergesort array in between l and h. assumes l≤h
sr msort(array<T> a, size_t l, size_t h) {
  // Corner cases
  if (l = h) return res;
  if (h-l = 1) {
    res.data.push_back(a[l]);
    res.runs.push_back(res.data.size());
    return res;
  }

  size_t m = l + (h-l) / 2; // Compute middle index

  sr res_1 = msort(a,l,m);   // Sort left side
  sr res_2 = msort(a,m,h);   // Sort right side
  return merge(res_1,res_2);   // Merge
}
```

Fig. 2 Pseudocode for Challenge 2: Mergesort With Runs.

1. Memory safety.
2. Termination.
3. `merge` merges correctly (permutation and sortedness).
4. `merge` returns the correct run indexes.
5. `msort` sorts the input (permutation and sortedness).
6. `msort` returns the correct run indexes.
7. `msort` is a stable sorting algorithm.
8. `msort` runs in $O(n \log n)$ time and $O(n \log n)$ space.

3.3 Solutions

This challenge had 8 verification tasks, the highest score received on this challenge was 4/8 (Team CIVL) and only one team received the next highest score of 3/8 (Acid Jazz). Both of these teams verified memory safety (partial for Acid Jazz) and termination. Team CIVL verified some correctness properties while Acid Jazz verified tasks 3 and 5 (partially).

4 Challenge 3: The World's Simplest Lock-Free Hash

This challenge is inspired by Jeff Preshing's blog entry: *The World's Simplest Lock-Free Hash Table*[4]. The corresponding pseudocode is shown in Fig. 3.

4.1 Problem Description

This simple hash-set has a fixed capacity, only supports insert and membership query operations, and uses linear probing for collision handling. However, it is thread-safe and implemented lock-free. At the core, the insert operation uses a compare-and-swap (CAS) operation to find a free spot for adding the key to be inserted. An optimization replaces expensive CAS operations by cheaper load operations when the table fills up.

The hash table works with a key type K. It has a special value

`key_invalid :: K`

that is used as placeholder for empty slots, and cannot be used as key. Moreover, there is a function

`get_hash(size_t,K) :: size_t`

such that `get_hash(`n`,`k`)` returns a hash-code for k, in the range $[0, n)$.

The compare and swap operation that we use returns the value that is stored in target after the operation (whether swapped or not). Competitors were free to replace it with whatever similar operation their tool supports. For example:

```
T compare_and_swap(T &target, T oldv, T newv) {
  T result;
  atomic {
    if (target = oldv) target=newv;
    result=target;
  }
  return result;
}
```

The competitors were informed that if their tool does not support concurrency, and they also cannot model concurrency on a more abstract level, then they

[4] `https://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/`

```
typedef hset = array<K>

hset empty(size_t n) {
  hset t = new K[n];
  for (size_t i=0; i<n; {+\¬+}\:i)
    t[i]=key_invalid;
  return t;
}

// Assumes k ≠ key_invalid
// returns true if key was inserted,
// false if table is full
bool insert(K k, hset t){
  size_t n = t.length;
  size_t i_0 = get_hash(n,k);
  size_t i = i_0;

  do {
    {
    // Optimization: probe for potentially free spot
      key kk = atomic_load(t[i]);

      // Key already in table
      if (kk = k) return true;

      // Spot taken, try next index
      if (kk ≠ key_invalid) {
        i = (i+1) mod n; continue;
      }
    }

    // Maybe i is still free when we
    // try to put our key there
    key k' = compare_and_swap(t[i],key_invalid,k);

    // We (or someone else) stored our key
    if (k' = k) return true;

   // Someone interfered with us, try next index
    i=(i+1) mod n;
  } while (i ≠ i_0);
  // Stop if we went one full round
  return false; // Table is full
}

// Assumes k ≠ key_invalid
bool member(hset t, key k) {
  size_t n = t.length;
  size_t i_0 = get_hash(n,k);
  size_t i = i_0;

  do {
    key k' = atomic_load(t[i]);
    if (k' = k) return true; // found the key
    if (k' = key_invalid) return false;
    // found empty entry. Key not in.
    i=(i+1) \mod n;
  } while (i ≠ i_0);
  return false;
  // Table full, our key is not in
}
```

Fig. 3 Pseudocode for Challenge 3: The World's Simplest Lock-Free Hash.

should implement and verify the sequential version of this hash-set as last resort.

Competitors were instructed to implement the above hash-table, using whatever atomic operations their tool supports, but try to stay lock-free. If they have to use locks, use a fine-grained lock around `compare_and_swap`, rather than locking the whole table for the duration of an `insert`/`member` operation.

4.2 Verification tasks:

There were 6 verification tasks for this challenge. Specifically, participants were asked to verify the following properties:

1. `empty(`n`)` creates an empty set with capacity n.
2. `member(`k`) == true`, if `insert(`k`)` has been executed before (and returned true).
3. `member(`k`) == false`, if no `insert(`k`)` that returned true can have been executed.
4. Termination.
5. Every key is contained in the table at most once.
6. If insert returns false, the table is full.

Competitors were permitted to re-phrase properties 2 and 3, but they should ensure that they could prove properties like:

```
insert(1) ∨ insert(2) ∨ insert(3);
{ assert(member(1) and ¬member(42)) }
  ∨ insert(4) ∨ insert(5);
```

4.3 Solutions

This challenge had 6 verification tasks, the highest score received on this challenge was 6/6 (Team CIVL) with Acid Jazz again second with (3/6). The bounded model-checker, CIVL, was very effective at this challenge but verification was restricted to 2 threads, each performing 2 arbitrary inserts, due to state space explosion. Acid Jazz verified tasks 1 and 4 completely, partially verifying tasks 2 and 6 using the Viper tool.

After the competition, Team KIV [19] published a paper that presents a refinement-based proof technique for concurrent systems. In their paper, they used this challenge as a running example to motivate their work. They remarked that they could not complete the challenge during the competition due to the time constraints imposed but provided a more detailed solution in their paper. Since the focus of their paper was not on the challenge itself, rather serving the purpose of presenting their novel approach, it is difficult to say exactly to what degree their updated solution addresses the challenge tasks.

5 Judging and Results

5.1 Judging Criteria

Judging at VerifyThis typically involves a private in-person discussion between the organisers and the individual teams. The purpose of this is to give the competitors an opportunity to explain their solutions to the organisers who may not be overly familiar with the specific tool that were used. As part of the submission process, teams were required to provide a completed self-assessment form[5]. This self assessment form included questions about the solutions that they provided and gave the participants a place to record any simplifications that were made as well as the good and bad points of their chosen verification tool in the context of each specific challenge.

We used this form to guide our discussion with the teams and we also assembled a marking sheet. Each task could receive 1 mark, with partial solutions receiving 0.5/1 (we did not reduce the granularity further than this). Using this marking scheme we were able to determine which teams solved the most tasks and then we more closely considered the nature of the solutions and how they compared to one another. For example, the CIVL model-checking tool verified a lot of tasks but only up to some size of input space whereas Viper could verify tasks completely but on an abstract model, rather than full implementation of the algorithm. Both approaches have positives and negatives and these needed to be considered to really understand the depth of the solutions that were provided.

In parallel with these discussions, teams were encouraged to present their solutions to the other teams that participated in the competition. This serves the purpose of allowing teams to see how their competitors solved the problems and increases awareness in the community about the specific verification tools that were used.

5.2 Results

The results of VerifyThis 2022 were as follows:

Best Overall Team:
Acid Jazz (Jonas Fiala and Thibault Dardinier)

Joint 2nd Place:
VerCors Yellow (Robert Rubbens and Philip Tasche)
Team CIVL (Stephen Siegel)

Most Distinguished Tool Feature:
Stainless (used counterexamples in a nice way)

6 Discussion

This section reflects on our experience of organising VerifyThis 2022. We compare our challenges to those from previous competitions and briefly summarise points related to diversity.

6.1 Lessons Learned

As with previous editions of VerifyThis [8], we found it difficult to assess how long a given challenge would take. However, by providing a list of verification tasks for each challenge we hoped to provide tasks that could be done relatively quickly as well as those that were more complex. Our challenge time slots were 120 minutes which is longer than at previous editions of VerifyThis. This was due to scheduling at ETAPS 2022. As such, we felt that we had the opportunity to pose some particularly challenging verification tasks. However, teams still struggled to tackle all of the tasks for every problem. In hindsight, rather than adding extra and more complex verification tasks to fill the time, we should have only added additional tasks that were no more difficult than the standard. For example, we were aware that task 5 of Challenge 1 was especially difficult (one of the organisers had verified the other tasks in prior work [11]).

It is difficult to assess the level of expertise amongst participants as well as the verification capabilities of the many tools that might be used. By providing multiple verification tasks, we hoped that we could include tasks that were accessible for all tools. As a result, we included some cross-cutting verification tasks such as termination (all challenges) and memory safety (Challenge 1 and 2). These tasks were generally solvable (even partially) for most of the participants.

As is traditional at VerifyThis, our third challenge was related to concurrency. We echo the point made by previous organisers that concurrency related challenges are difficult to design, especially whilst being conscious that not all tools/teams will have the capacity to model and verify such problems [8]. Additionally, and in general, it is difficult to ensure that no solution can be easily found online to these problems. With the advancement of technologies like ChatGPT this will become even more difficult and future organisers should keep this in mind when designing challenges.

ETAPS 2022 was the first in-person conference for many attendees since the COVID-19 pandemic and the organisers endeavoured to make it an in-person only event. However, this proved difficult for both VerifyThis and ETAPS. Since COVID was still prevalent, we had to facilitate some remote participation at the last minute. This would have been better if included in the planning from the beginning and some potential participants complained about us not having remote attendance. Remote is harder to facilitate though because

[5] `https://mariefarrell.github.io/verifythis/self_assessment.txt`

Challenge	Sequential/Concurrent	Input	Algorithm	Partial	Complete
1	Sequential	Matrix	Iterative	65	0
2	Sequential	Array	Recursive	88	0
3	Concurrent	Array	Iterative	76	0.06

Table 2 Percentage of partial and complete results for the challenges.

it is impossible to determine whether or not the competitors are following the rules. Perhaps a more hybrid format could be considered in the future where the competition, like the IEEE Xtreme[6] programming competition which has multiple competition venues geographically. This would also help to encourage wider participation in and awareness of VerifyThis.

6.2 Analysis of the Solutions

The submissions used a variety of verification tools, as summarised in Table 1. Of the 17 teams that competed, 10 were student-only teams. KeY Team 2 was the only team that had an undergraduate team member. All teams submitted solutions to Challenge 1 and 2, 15 teams provided a solution to Challenge 3.

We compare our results with those from previous editions of VerifyThis using the values from Table 2 in [8]. We illustrate the percentage of partial and complete solutions to our challenges in Table 2.

Our Challenge 1 had the lowest ever percentage of completions across all of the previous VerifyThis Challenge 1 problems. This challenge proved especially difficult for competitors to solve. The majority of teams submitting partial solutions could verify memory safety and termination. One team was able to specify and partially verify task 3 (that the output point cloud is smaller than the input point cloud). This is interesting because this challenge was drawn from a real-life problem. Its difficulty likely stems from the teams having very limited time to provide a complete solution and the tools themselves needing improvement to efficiently deal with realistic examples.

In contrast, our partial completion values for Challenge 2 was the highest amongst all previous recursive challenges. Specifically, we had 88% of teams produce partial solutions. This is in contrast with 55% (VerifyThis 2018), 40% (VerifyThis 2017) and 18% (VerifyThis 2012). Almost all teams were able to verify some degree of memory safety and termination for this challenge. Team Soundproof were ranked highest on this challenge using Viper. They verified memory safety, partial termination and partially that the algorithm sorts the input.

[6] `https://ieeextreme.org/`

We had 76% of teams produce partial solutions to the concurrent Challenge 3 which seems to be in line with previous concurrency challenges. These include 85% (VerifyThis 2019), 60% (VerifyThis 2017) and 79% (VerifyThis 2016). Team CIVL was the only team to produce a complete solution to Challenge 3 using bounded model checking.

6.3 Equality, Diversity & Inclusion

We included a Diversity questionnaire that participants were asked to complete (though it was not compulsory). Since we have a relatively small set of participants in the competition we do not specifically cite the numeric results here. However, the vast majority of participants were white, male Europeans. In order to diversify participants at VerifyThis we suggest that future organisers make a conscious decision to advertise the competition more widely and encourage more non-white, non-male, non-Europeans to participate in future editions of VerifyThis.

7 Conclusion

VerifyThis gives participants the opportunity to compete as well as interact with others interested in program verification. We thoroughly enjoyed the competition and we extend our heartfelt thanks to those who were brave enough to attempt our challenges. We hope that you continue to participate in future editions of VerifyThis.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, et al. The KeY tool: integrating object oriented design and formal verification. *Software & Systems Modeling*, 4:32–54, 2005.
2. V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods Symposium*, pages 88–108. Springer, 2022.
3. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal System Development with KIV. In *International Conference on Fundamental Approaches to Software Engineering*, pages 363–366. Springer, 2000.

4. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors tool set: verification of parallel and concurrent software. In *International Conference on Integrated Formal Methods*, pages 102–110. Springer, 2017.
5. A. Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant.* MIT Press, 2022.
6. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.
7. X. Denis, J.-H. Jourdan, and C. Marché. *The Creusot Environment for the Deductive Verification of Rust Programs.* PhD thesis, Inria Saclay-Île de France, 2021.
8. C. Dross, C. A. Furia, M. Huisman, R. Monahan, and P. Müller. VerifyThis 2019: a program verification competition. *International journal on software tools for technology transfer*, 23(6):883–893, 2021.
9. M. Eilers and P. Müller. Nagini: a static verifier for Python. In *International Conference on Computer Aided Verification*, pages 596–603. Springer, 2018.
10. M. Farrell, N. Mavrakis, C. Dixon, and Y. Gao. Formal verification of an autonomous grasping algorithm. In *International symposium on artificial intelligence, robotics and automation in space. European Space Agency*, 2020.
11. M. Farrell, N. Mavrakis, A. Ferrando, C. Dixon, and Y. Gao. Formal modelling and runtime verification of autonomous grasping for active debris removal. *Frontiers in Robotics and AI*, 8, 2021.
12. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis 2012. *International journal on software tools for technology transfer*, 17(6):647–657, 2015.
13. M. Huisman, V. Klebanov, R. Monahan, and M. Tautschnig. VerifyThis 2015. *International journal on software tools for technology transfer*, 19(6):763–771, 2017.
14. M. Huisman, R. Monahan, P. Müller, A. Paskevich, and G. Ernst. VerifyThis 2018: A program verification competition. Technical report, Université Paris-Saclay, 2019.
15. M. Huisman, R. Monahan, P. Muller, and E. Poll. VerifyThis 2016: A program verification competition. 2016.
16. V. Kuncak and J. Hamza. Stainless verification system tutorial. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 2–7. IEEE, 2021.
17. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.
18. M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 158–174, 2021.
19. G. Schellhorn, S. Bodenmüller, and W. Reif. Thread-local, step-local proof obligations for refinement of state-based concurrent systems. In *International Conference on Rigorous State-Based Methods*, pages 70–87. Springer, 2023.
20. S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: the concurrency intermediate verification language. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.
21. F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In *International Conference on Computer Aided Verification*, pages 367–379. Springer, 2021.