VerifyThis 2017

A Program Verification Competition

Marieke Huisman[1], **Rosemary Monahan**[2], **Peter Müller**[3], **Wojciech Mostowski**[4], **Mattias Ulbrich**[5]

[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] Halmstad University, Sweden e-mail: Wojciech.Mostowski@hh.se
[5] Karlsruhe Institute of Technology, Germany e-mail: ulbrich@kit.edu

Abstract. VerifyThis 2017 was a two-day program verification competition which took place from April 22-23rd, 2017 in Uppsala, Sweden as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2017). It was the sixth instalment in the VerifyThis competition series. This article provides an overview of the VerifyThis 2017 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 2017 took place from April 22-23rd, 2017 in Uppsala, Sweden, as a two-day verification competition in the European Joint Conferences on Theory and Practice of Software (ETAPS 2017). It was the sixth edition in the VerifyThis series after the competitions held at FoVeOOS 2011, FM2012, Dagstuhl (Seminar 14171, April 2014), ETAPS 2015 (April 2015), and ETAPS 2016 (April 2016).

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. The verification challenges are available from the VerifyThis website http://www.pm.inf.ethz.ch/research/verifythis.html.

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 2017

VerifyThis 2017 consisted of three verification challenges. Before the competition, an open call for challenge submissions was made. The two submitted were used as inspiration for the competition. The challenges (presented later) provided reference implementations at different levels of abstraction.

Ten teams participated (Table 1) in this edition of the competition. Teams of up to two people were allowed and physical presence on site was required. We particularly encouraged participation of:

- student teams (including 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 also welcome.

We started the competition day with an invited tutorial by Jean-Christophe Filliâtre on the Why3 software verification tool [6]. This tutorial was open to all ETAPS participants and included a small verification challenge, where participants developed a verified solution to Kadanes Algorithm [2]. The tools used where those available from http://why3.lri.fr/try/. Why3 has been one of the most popular tools during previous (and also the present) verification competitions so the topic was of interest to all 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 2017 instalment of VerifyThis can be found at http://www.pm.inf.ethz.ch/research/verifythis/Archive/2017.html. 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 [14,4,7,11,12] 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. These sessions provided an excellent opportunity to explore the strengths and weaknesses of the solutions presented by each team, while acquiring more detailed knowledge of the verification tools used. Concurrently, all other participants presented their solutions, leading to lively discussion and exchange about tool developments. 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? Where any bugs found in the code?
- Completeness: are all tasks solved, and are all required aspects covered? Are any assumptions made? Is termination verified?
- 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? Does the solution make use of information from libraries?
- Novelty: does a submission apply novel techniques? What special features of the tool are used in the solution?

2 Challenge 1: Pair Insertion Sort

Although it is an algorithm with $O(n^2)$ complexity, the pair insertion sorting algorithm is used in modern library implementations. When dealing with smaller numbers of elements, insertion sorts performs better than, e.g., quicksort due to a lower overhead. It can be implemented more efficiently if the array traversal (and rearrangement) is not repeated for every element individually. A Pair Insertion Sort in which two elements are handled at a time is used by Oracle's implementation of the Java Development Kit (JDK) for sorting primitive values in $java.util.Arrays.sort(int[])$ [1]. In the following code snippet, a is the array to be sorted, and the integer variables *left* and *right* are valid indices into a that set the range to be sorted.

```
for (int k = left; ++left <= right; k = ++left) {
    int a1 = a[k], a2 = a[left];
    if (a1 < a2) {
        a2 = a1; a1 = a[left];
    }
    while (a1 < a[--k]) {
        a[k + 2] = a[k];
    }
    a[++k + 1] = a1;
    while (a2 < a[--k]) {
        a[k + 1] = a[k];
    }
    a[k + 1] = a2;
}
int last = a[right];
while (last < a[--right]) {
    a[right + 1] = a[right];
}
a[right + 1] = last;
```

This implementation is an optimised algorithm which uses the borders $a[left]$ and $a[right]$ as sentinels.

While the problem was proposed as a Java implementation, the challenge does not use specific language features and can easily be formulated in other languages. A simplified variant of the algorithm in pseudo code,

[1] in http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8-b132/java/util/DualPivotQuicksort.java

Table 1. Teams participating in VerifyThis 2017 (alphabetically by tool).

#	Team members	Tool		Team attributes
1	Stephen Siegel	CIVL	[16]	developer
2	Jon Mediero Iturrioz	Dafny	[15]	student,non-developer
3	Lionel Blatter, Jean-Christophe Léchenet	Frama-C	[13]	student,non-developer
4	Mihai Herda	KeY	[1]	student
5	Michael Kirsten, Jonas Schiffl	KeY	[1]	student
6	Stefan Bodenmüller, Jörg Pfähler	KIV	[5]	student
7	Marieke Huisman, Wytse Oortwijn	VerCors	[3]	developer
8	Jean-Christophe Filliâtre	Why3	[6]	developer
9	Léon Gondelman, Marc Schoolderman	Why3	[6]	student
10	Mário Pereira, Raphael Rieu-Helft	Why3	[6]	student

for sorting an array A whose indices range from 0 to $length(A) - 1$, is the following:

```
i = 0                   //i is the running index
while i < length(A)-1
    x = A[i]            //x,y hold the next 2 elements
    y = A[i+1]
    if x < y then
        swap x and y
    j = i - 1           //j finds the insertion point
    while j >= 0 and A[j] > x
                        //find the insertion point for x
        A[j+2] = A[j]   //shift existing content by 2
        j = j - 1
    end while
    A[j+2] = x          //store x at its insertion place
                        //A[j+1] is an available space
    while j >= 0 and A[j] > y
                        //find the insertion point for y
        A[j+1] = A[j]   //shift existing content by 1
        j = j - 1
    end while
    A[j+1] = y          //store y at its insertion place
    i = i+2
end while

if i = length(A)-1      //if length(A) is odd, an extra
    y = A[i]            //single insertion is needed
    j = i - 1           //for the last element
    while j >= 0 and A[j] > y
        A[j+1] = A[j]
        j = j - 1
    end while
    A[j+1] = y
end if
```

2.1 Verification Tasks

1. Specify and verify that the result of the pair insertion sort algorithm is a sorted array.
2. Specify and verify that the result of the pair insertion sort algorithm is a permutation of the input array.

Getting Started. To make the exercise more accessible, feel free to start with stripped down versions of the problem. A few possibilities for simplifications are:

- Absence of unexpected runtime exceptions.
- Verify a single-step insertion sort algorithm in which every element is handled individually.
- For permutations proofs, it may be simpler to not remember the values in temporary variables (x and y in the pseudocode), but to swap repeatedly.

Challenge. Try to get as close as possible to Oracle's implementation (outlined above) from the beginning.

Verification Bounds. In reality, pair insertion sort is used only for small problem instances: in JDK's case, if the array has less than 47 elements. If it helps your efforts, you may assume a suitable length restriction for the array.

2.2 Comments on Solutions

- All teams submitted a (not necessarily complete) solution to the challenge.
- Six teams considered the total problem (taking termination into account), three looked at the partial problem, and one team (CIVL) considered it using bounded verification.
- All teams tried to tackle the sortedness problem first before dealing with the permutation question.
- Every team that considered the permutation property also verified sortedness.
- The discussion revealed that modelling the permutation property using multilists (bags) is more promising than using an explicit notion of permutation. The conjecture is that the abstraction into a multiset captures the necessary information (same elements) whereas modelling the permutation explicitly also carries information about the elements' positions which is not requried for the property.
- The additional algorithmic complexity imposed by the real-world implementation made it harder to verify. No team verified the optimised Java version on

site. A proof using KeY was finished in the weeks following the event.
- The algorithm has been verified using bounded verification with CIVL up to bound $N = 5$.
- One team found a full solution during the challenge time using Why3. Existing verification libraries (in particular bags) was used and the proof was guided using additional autoactively annotated assertions.
- Notable tool features which assisted in the solutions were graphical representation of proof trees in KIV, access to multiple provers and use of logical cuts (to help debugging) in Why3 and symbolic execution in CIVL.
- Although no team used SPARK in the competition, this challenge was completed off-site to illustrate how one can reach different levels of software assurance with SPARK. Details can be found at http://www.spark-2014.org/entries/detail/verifythis-challenge-in-spark

3 Challenge 2: Maximum-sum subarray

3.1 Verification Task

The maximum-sum subarray problem was first surveyed by Bentley in his Programming Pearls column of CACM in 1984. The solution returns the sum of a contiguous subarray within a one- dimensional array of numbers which has the largest sum. In the two-dimensional case, the task is to find a submatrix such that the sum of its elements is maximized. This problem is widely used in applications such as pattern recognition, image processing, biological sequence analysis and data mining.

3.2 One-dimensional Case:

In the array [-2, -3, 4, -1, -2, 1, 5, -3] the maximum-sum subarray is [4,-1,-2, 1, 5] with a sum of 7 (4 + (-1) + (-2) + 1 + 5 = 7). A brute force solution checks all subarrays (which are quadratically many), but the problem is solvable in linear-time using Kadane's algorithm. Kadane's Algorithm for a one-dimensional array is given below as an implementation in C:

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}
```

The One-dimensional Verification Task:

1. Verify that Kadane's Algorithm returns a value which is the sum of a contiguous subarray within array a, and
2. Verify that the sum of every contiguous subarray within a is not greater than the returned value.

3.3 Two-dimensional Case:

For the given two-dimensional array

$$\begin{vmatrix} 0 & -2 & -7 & 0 \\ 9 & 2 & -6 & 2 \\ -4 & 1 & -4 & 1 \\ -1 & 8 & 0 & -2 \end{vmatrix}$$

the maximum-sum submatrix is

$$\begin{vmatrix} 9 & 2 \\ -4 & 1 \\ -1 & 8 \end{vmatrix}$$

To get an $O(N^3)$ algorithm, we create $O(N^2)$ one-dimensional subproblems by iterating over all possible contiguous sequences of row indices. We then apply Kadane's algorithm to each 1D subproblem, with the maximum-sum subarray amongst these returning the solution to the 2D problem. The subproblems are arrays which contain accumulated sums of contiguous rows. Dynamic programming can be used to obtain these efficiently. In the example above the contiguous sequences of row indices from which sums are formed are: [0], [0,1], [0,1,2], [0,1,2, 3], [1], [1,2], [1,2,3], [2], [2,3], [3].

The arrays holding the accumulated sums are:

[0]	0	-2	-7	0
[0, 1]	9	0	-13	2
[0, 1, 2]	5	1	-17	3
[0, 1, 2, 3]	4	9	-17	1
[1]	9	2	-6	2
[1, 2]	5	3	-10	3
[1, 2, 3]	4	11	-10	1
[2]	-4	1	-4	1
[2, 3]	-5	9	-4	-1
[3]	-1	8	0	-2

We apply Kadane's algorithm to each of these arrays, keeping the maximum sum found across all arrays. For example, above, the maximum sum of 15 is found in row sequence [1,2,3]. The maximum-sum submatrix column bounds (0 and 1 here) are obtained from the bounds of the maximum-sum subarray. The row bounds are obtained from the row sequence yielding the maximum sum (1-3 in the example).

The Two-dimensional Verification Task: Implement and verify a function that solves the maximum-sum submatrix problem. You should verify that

1. the function returns a value which is the sum of a submatrix within the input matrix, and that
2. the sum of every submatrix within the input matrix is not greater than the returned value.

3.4 Comments on Solutions

Since this challenge has been issued as a supplementary followup challenge in response to the example of the Why3 tutorial (1-dimensional Kadane), the submitted solutions were not assessed after the competition.

4 Challenge 3: Odd-even Transposition Sort

This sorting algorithm, developed originally for use on parallel processors, compares all odd- indexed list elements with their immediate successors in the list and, if a pair is in the wrong order (the first is larger than the second) swaps the elements. The next step repeats this for even-indexed list elements (and their successors). The algorithms iterates between these two steps until the list is sorted.

4.1 Single Processor Solution

The single-processor algorithm is simple, but not very efficient $O(n^2)$. It can be considered a variation of the bubble sort algorithm. Here a zero-based index is assumed:

```
function oddEvenSort(list) {
    function swap(list, i, j) {
        var temp = list[i];
        list[i] = list[j];
        list[j] = temp;
    }
    var sorted = false;
    while(!sorted) {
        sorted = true;
        for(var i = 1; i < list.length-1; i += 2) {
           if(list[i] > list[i+1]) {
                swap(list, i, i+1);
                sorted = false;
            }
        }
        for(var i = 0; i < list.length-1; i += 2) {
            if(list[i] > list[i+1]) {
                swap(list, i, i+1);
                sorted = false;
            }
        }
    }
}
```

4.2 Multi Processor Solution

On parallel processors, with one value per processor and only local left–right neighbour connections, the processors all concurrently do a compare–exchange operation with their neighbours, alternating between odd–even and even–odd pairings in each step. The algorithm has linear runtime as comparisons can be performed in parallel. A pseudocode implementation that uses message passing for synchronisation is presented in the following. The driver code spawns n processes, one for each array element and collects the results after termination.

```
process ODD-EVEN-PAR(n, id, myvalue)
    //n ... the length of the array to sort
    //id ... processors label (0 .. n-1)
    //myvalue ... the value in this process
begin
    for i := 0 to n-1 do
    begin
    //alternate between left and right partner
    if i+id is even then
        if id has a right neighbour
            sendToRight(myvalue);
            othervalue = receiveFromRight();
            myvalue = min(myvalue, othervalue);
    else
        if id has a left neighbour
            sendToLeft(myvalue);
            othervalue = receiveFromLeft();
            myvalue = max(myvalue, othervalue);
    end for
end ODD-EVEN-PAR

for i := 0 to array.length-1
    process[i] := new ODD-EVEN-PAR(n, i, array[i])
end for

start processes and wait for them to finish

for i := 0 to array.length-1
    array[i] := process[i].myvalue
end for
```

4.3 Verification Tasks

1. Specify and verify that the result of the even-odd sort algorithm is a sorted list.
2. Specify and verify that the result of the even-odd sort algorithm is a permutation of the input list.
3. Prove that the code terminates.

Concurrency: This algorithm was developed originally for parallel use. You should aim to have a parallel solution if your tool allows.

Synchronisation: We have proposed a synchronisation scheme using messages between neighbouring processes. You are free to use a different scheme (semaphores, locks, ...) if you wish.

Caution: The implementations shown above are for demonstration purposes only, they have not been thoroughly tested, let alone formally verified. That's your job!

4.4 Comments on Solutions

- Two teams/tools tackled the multi-processor problem (VerCors and CIVL)
- VerCors has used their distinguished feature: verifying kernel programs on GPUs. The challenge has been reformulated as a kernel function and and verification has been attempted. The remaining unsolved challenge was specifying the global invariant coupling the thread states together.
- CIVL was able to use their distinguished feature: bounded verification of concurrent programs. They implemented the distributed algorithm as a runnable C program using an MPI framework. The bound up to which it was successfully verified is $N = 5$ (number of nodes).
- No full deductive verification of the sequential challenge could be achieved during the challenge time slot.
- Due to the formulation of the challenge, showing sortedness after termination of the sequential program was rather simple (`while(!sorted)`). The challenging question in this task was proving termination (that the array is sorted eventually).
- Some teams independently came up with a suitable termination variant for the sequential problem: the number of comparisons within the array which are out of order (number of inversions, $\#\{(i,j) \mid i < j \text{ and } a[i] > a[j]\}$).

5 Challenge 4: Tree Buffer

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

The task is to verify a data structure called tree buffer. We start with a simple version, and then we will introduce more requirements related to efficiency.

Consider the OCaml interface

```
type 'a buf
val empty : int -> 'a buf
val add : 'a -> 'a buf -> 'a buf
val get : 'a buf -> 'a list
```

with the following implementation

```
type 'a buf = { h : int; xs : 'a list }

let rec take n xs = match n, xs with
  | 0, _ | _, [] -> []
  | n, x :: xs -> x :: take (n-1) xs

let empty h = { h; xs = [] }
let add x { h; xs } = { h; xs = x :: xs }
let get { h; xs } = take h xs
```

When we create a tree buffer, we fix a parameter h, and all invocations to get will return the last h elements added in the tree buffer. Incidentally, this is a tree in the sense that one can write code like the following:

```
let e = empty 3;;       (* e is a root *)
let t1 = add 1 e;;      (* t1 has parent e *)
let t2 = add 2 t1;;     (* t2 has parent t1 *)
let t3 = add 3 t1;;     (* t3 has parent t1 *)
```

One problem with our current implementation is that it wastes memory. A possible solution is the following caterpillar implementation:

```
type 'a buf = { h : int; xs : 'a list;
                xs_len : int; ys : 'a list }
let empty h = { h; xs = []; xs_len = 0; ys = [] }
let add x { h; xs; xs_len; ys } =
    if xs_len = h - 1
    then { h; xs = []; xs_len = 0; ys = x :: xs }
    else { h; xs = x :: xs; xs_len = xs_len + 1; ys }
let get { h; xs; xs_len; ys } = take h (xs @ ys)
```

The length of xs is always less that h because, whenever it would become h, its content is moved into ys, and the old content of ys is thrown away.

5.1 Verification Tasks

1. Verify that the naive and the caterpillar implementations are functionally equivalent. You are encouraged to translate
   (a) the naive implementation into your favorite specification language, and
   (b) the caterpillar implementation into your favorite imperative language. Your caterpillar implementation must use only constant time for add, ignoring time possibly spent in the garbage collector.
2. In this task we want to move to a real-time setting. This means, in particular, that we can no longer ignore the time spent in the garbage collector. But, we still want
   (a) a constant time add operation, and
   (b) memory use that is within a constant factor of the caterpillar's live-heap size.

The idea is to add reference counters to the caterpillar. In addition, we need to control the rate at which objects are deleted by the reference counting scheme. We control the rate by holding some references in the queue *to_delete*.

```
// g++ -std=c++14

#include <memory>
#include <queue>

struct List { virtual ~List() {} };
typedef std::shared_ptr<List> PL;

struct Nil : List {};
struct Cons : List {
int head; PL tail;
Cons(int head, PL tail) : head(head), tail(tail) {}
virtual ~Cons();
};

std::queue<PL> to_delete;

// IMPORTANT: delay the deletion,
// so that time per operation is constant
```

```
Cons::~Cons() { to_delete.push(tail); }

struct Buf {
  int h; PL xs; int xs_len; PL ys;
  Buf(int h, PL xs, int xs_len, PL ys) : h(h), xs(xs),
      xs_len(xs_len), ys(ys) {}
  virtual ~Buf(); // manual GC is triggered
  PL get(); // unimplemented here; would use xs and ys
private:
};

void process_queue() {
  if (!to_delete.empty()) to_delete.pop();
  if (!to_delete.empty()) to_delete.pop();
}

// NOTE: This is called automatically by C++.
// In the C implementation, there is a deactivate
// function that plays the role of this destructor.
Buf::~Buf() { process_queue(); }

Buf empty(int h) {
  return Buf(h, PL(new Nil()), 0, PL(new Nil()));
}
Buf add(int x, Buf t) {
  process_queue();
  if (t.xs_len + 1 == t.h) {
    return
      Buf(t.h, PL(new Nil()), 0, PL(new Cons(x, t.xs)));
  } else {
    return
      Buf(t.h, PL(new Cons(x, t.xs)), 1+t.xs_len, t.ys);
  }
}

int main() {
  Buf e = empty(3);
  Buf t1 = add(1, e);
  Buf t2 = add(2, t1);
  Buf t3 = add(3, t1);
}
```

Verification Tasks.

1. Show that the real-time implementation is functionally equivalent to the naive implementation.
2. Prove bounds on the resources used, such as time spent in add, total memory used, and so on.

Warning: The C++ implementation above has not been tested, so it probably has bugs. A C implementation, which has been tested but has the drawback of verbosity, can be found on GitHub at https://github.com/rgrig/treebuffers/blob/master/treebuffer.c. There is also a (inefficient but pretty) javascript implementation in the presentation on Tree Buffers from the 27th International Conference on Computer Aided Verification (CAV) 2015 at http://rgrig.appspot.com/static/talks/treebuffers/index.html.

You are free to verify whichever implementation you choose: the important part is to satisfy the same, real-time performance characteristics. That does mean that you cannot use a language with garbage collection, unless you can control the garbage collector, and you can verify resource bounds for the system including your program and the garbage collector.

5.2 Comments on Solutions

- This challenge was less accessible than the previous ones. To some participants, the purpose of the data structure became clear only towards the end of the session.
- CIVL was able to use a distinguished feature of the tool: functional equivalence verification. The caterpillar verification was verified to have the same result as a straight forward C implementation. The bounded verification was scaled up to $N = 8$.
- KIV was able to use a distinguished feature of the tool: Abstract State Machine (ASM) refinement. With the tool, two formal refinement steps were conducted, connecting first the abstract naive functional algorithm to the functional caterpillar implementation, which in turn was refined into an imperative implementation operating on linked lists on an explicit heap. The provided solution was very close to the challenge statement (apart from the last implementation not being in C).
- The KeY teams used the functional implemenation as declarative specification of a Java implementation of the caterpillar version.
- One Why3 team used an explicit predicate "valid" that encodes observational equivalence of a naive and an optimsed data structure. It is used in pre- and postconditions to check the coupling invariant.

6 Results and Closing Remarks

6.1 Awarded Prizes

Prizes were awarded in the following categories:

- Best team: Jean-Christophe Filliâtre (Why3)
- Best student team—awarded to two teams:
  - Stefan Bodenmüller, Jörg Pfähler (KIV)
  - Mário Pereira, Raphael Rieu-Helft (Why3)
- Distinguished user-assistance tool feature: Stephen Siegel (CIVL) for CIVL Compare which compares two programs for functional equivalence
- Best challenge submission: Radu Grigore, (Challenge 4: Tree Buffer)

The best student teams each received a 300 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.

6.2 Final Remarks

Some final observations from the post-mortem sessions follow below. With-respect-to the challenge solutions:

- How you phrase your specification can often make a big difference in how easy it is to verify your programs. Example: sortedness predicate can be formulated in many ways. One best team award winner recommended not to use the inductive formulation $\forall k.(inRange(k) \rightarrow A[k] \leq A[k+1])$, as that is difficult for the first-order provers.
- Built-in support for mathematical structures is essential. Most tools provide support for sequences in one way or the other and that was really useful. This competition showed that many tools don't support bags, which would have been useful when describing the contents of a sequence or array.
- Proof debugging features are essential to quickly identify why something cannot be verified (intermediate assertions, the 'by' and 'so' reasoning, as used by Why3 teams). The more systematic this kind of support is, the easier it is to finish verifications.
- Ghost variables are really useful in practical program verification. From a theoretical point of view, they are not necessary, but they allow the implicit steps in the program to be captured in an explicit way [8]. For example in solutions to challenge 4, they were useful for reasoning about the list of deleted elements in the treeBuffer, instead of reasoning explicitly about prefixes.
- Libraries are often the bottleneck, or the winner. Having collections of properties that are already there, that do not have to be proven over and over again, can really speed up the verification process.
- Tool output is often not trimmed to be judged outside the tool. Annotation-based autoactive verification systems have a few advantages in that respect.

With-respect-to the competition organisation we note that:

- The competition challenges would benefit from more general phrasing to accommodate different verification technologies. Or, stated differently, alternative verification tasks w.r.t. different technologies could be stated explicitly.
- When judging we discussed preparing an online form for the participants to fill in upon challenge completion to "pre-grade" the solution based on self-evaluation. Such a form, however, should only be available at the end of each challenge, so that it does not guide the teams' work.
- Having different teams using the same tool on site was considered a benefit by some participants. Thus, different specification and verification styles for the same tool can be compared.

A new edition of the VerifyThis competition will be held as part of ETAPS 2018.

Acknowledgments

The organisers would like to thank Radu Grigore, Rustan Leino and Alex Summers for their feedback and support prior to and during the competition. The organisers also thank the competition's sponsors: The Automated Reasoning Group, Amazon Web Services and Galois, Inc. Their contributions helped us to support participants with travel grants, and to finance the various prizes.

References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, Dec. 2016.
2. J. Bentley. Programming pearls: Algorithm design techniques. *Commun. ACM*, 27(9):865–873, Sept. 1984.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. M. Hofmann and M. Pavlova. *Elimination of Ghost Variables in Program Logics*. Springer Berlin Heidelberg, 2008.
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.
12. M. Huisman, R. Monahan, P. Müller, and E. Poll. VerifyThis 2016. A program verification competition. *CTIT Technical Report Series*, 2016.
13. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
14. 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.
15. 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.
16. 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.