VerifyThis 2023: an International Program Verification Competition

Xavier Denis[1] and Stephen F. Siegel[2][0000−0001−9359−3332]

[1] Université Paris-Saclay, INRIA, ENS Paris-Saclay, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France
[2] University of Delaware, Newark DE 19716, USA
siegel@udel.edu

Abstract. The 11th VerifyThis program verification competition took place on April 22, 2023, at ETAPS 2023 in Paris, France. Contestants were tasked with solving three verification challenges in real time, using any tools of their choice. The subjects of the challenges were (1) in-place reversal of a linked list, (2) ordered binary decision diagrams, and (3) a concurrent FIFO queue. Fifteen teams, each with 1 or 2 members, participated; all were required to be present on site. In this report, we summarize the three challenges and some of the notable solutions.

1 Introduction

VerifyThis is a series of program verification competitions. It has taken place annually since 2011, except for 2013 and 2020. Since 2015, it has been held as a workshop at the European Joint Conferences on Theory and Practice of Software (ETAPS). The 2023 event was the 11th in the series. It took place on Saturday, April 22, in the library of the Institut Henri Poincaré, Sorbonne University, Paris. Sunday was used to present and discuss solutions, and for judging.

In contrast with several other competitions, VerifyThis is not a fully automated affair. Instead, participants are required to be physically present, working in teams of one or two members each. They are given a sequence of three *challenges* to solve in real time. The typical challenge starts with some combination of: a natural language description of a problem, pseudocode, and/or actual code. This is followed by a sequence of verification *tasks*. The tasks usually involve (re-)implementing the algorithm in a language appropriate for the verification tool(s) the participant will use, and then formalizing and verifying certain properties of the implementation. Teams may use any languages and verification tool or tools they wish. Each challenge lasts 90 minutes. (Some extra time was allowed for Challenge 2 this year).

The competition is therefore measuring the skill of the participants as well as the effectiveness of the tools. As the solutions may use tools with very different input languages, techniques, and kinds of correctness guarantees, there is no straightforward algorithm to rank the solutions. Instead, judges evaluate the solutions based on correctness, completeness, and elegance.

```
struct Cell { int value; struct Cell *next; };
struct Cell *list_reversal(struct Cell *l) {
  struct Cell *r = NULL;
  while (l != NULL) {
    struct Cell *tmp = l;
    l = l->next;
    tmp->next = r;
    r = tmp;
  }
  return r;
}
```

Fig. 1. In-place reversal of linked list.

This year, 23 people participated in 15 teams. Participants represented institutions from at least seven countries: Canada, France, Germany, the Netherlands, Switzerland, the United States, and the United Kingdom. They used a variety of tools, virtually all of them based on deductive verification approaches. The tools include Why3, TrustInSoft Analyzer, VerCors, Gobra, Viper, SPARK, SecC, Creusot, and GhostPtrToken. Some of these tools also used automated theorem provers, including Z3, Alt-ergo, and CVC4. Viper appears to have been the most popular tool, with at least 5 teams using it on at least one challenge.

The organizers, who are the authors of this report, solicited challenges from the community and composed some of their own. In the end, we selected one contributed challenge, Challenge 1, proposed by Jean-Christophe Filliâtre and Andrei Paskevich. Challenges 2 and 3 were written by us.

In the remainder of this report, we summarize the challenges and highlight some of the approaches and solutions. Conclusions and a list of the winners are given in the final section.

2 Challenge 1: In-place Reversal of Linked List

The first problem, contributed by Jean-Christophe Filliâtre and Andrei Paskevich, dealt with a standard linked list algorithm: in-place of reversal of a singly-linked list. The challenge provided C code implementing this algorithm, shown in Figure 1.

The algorithm clearly terminates on a NULL-terminated list, but less well-known is the fact that it also terminates on any infinite list that loops at some point, i.e., a lasso-shaped list with an initial segment (of any nonnegative length) and then a cycle (of any positive length). When applied to a list like this, list reversal reverses the cycle and leaves the initial segment unchanged (by reversing it twice). If memory is finite, any list is either NULL-terminated or loops at some point, and thus list reversal always terminates.

The challenge specified five tasks:

1. Implement list reversal in a language of your choice.

2. Show that your implementation, if invoked on a well-formed list, is free of runtime violations; in particular, no invalid pointer operation occurs. Part of this task is to specify precisely "well-formed." (Both NULL-terminated and lasso-shaped lists are considered well-formed.)
3. Show that your implementation, if invoked on any well-formed list in finite memory, terminates.
4. Show that, at termination, the resulting list is the reverse of the original. For lasso-shaped lists, this means the initial segment is unchanged and the cycle is reversed, as explained above.
5. Show that the space and time consumed are linear in the length of the list.

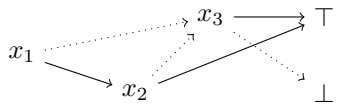
Comments on solutions. Despite its similarity to the classic introductory separation logic problem of reversing a linked-list, the introduction of a lasso at the end of the list makes this problem much harder to verify. Most teams (11 of 16) verified memory safety, fewer teams proved termination (5 of 16) though 2 additional teams demonstrated portions of the termination argument. One team each proved problems 3 and 4, while a single team proved the linear space usage of the algorithm.

Team *Wondering How and Why3* (Josué Moreau and Paul Patault) verified tasks (1), (2), and (3). An excerpt of their solution is shown in Appendix A. This solution is notable both for the amount of problems solved, but also for the tool used to do it. Why3 does not support separation logic, instead the team manually encoded a Burstall-Bornat memory model [2] to represent their heap. They then specified their lists using two representation predicates. The first predicate, `segList2L(mem, s, i, j, l, p)`, takes a memory `mem`, an *abstract value* for the list mapping indices of the list to their memory location, a start index i, a final index j, a starting location l, and a loop start index p. The predicate establishes that $s[i..j]$ is a list starting at l and either ending with at a null (and thus lasso-free list) or at the position $s[p]$ of the lasso head. The second predicate `segList2R(mem, s, i, j, l)` takes a memory, an abstract list s, a start index i, final index j and end location l. It establishes that $s[i..j]$ forms a list in memory, but starting from the final cell of the list l. Their loop starts with the whole list represented by `segList2R` but at each iteration removes one element from that predicate which is then captured by `segList2L`.

A solution by the authors of this challenge can be found at https://toccata.gitlabpages.inria.fr/toccata/gallery/linked_list_rev.en.html.

3 Challenge 2: Binary Decision Diagrams

The second challenge consisted of verifying a minimal Reduced Ordered Binary Decision Diagram (ROBDD) [1,3] library. BDDs allow compact representations of propositional formulas by sharing common sub-expressions. Formulas are stored as directed acyclic graphs, where each internal node represents the conditional test of a variable x. Two leaf nodes are used to indicate the values $\top$ and $\bot$. Using a BDD, the formula $(x_1 \wedge x_2) \vee x_3$ would be represented as the following graph:



Dotted lines correspond to the false branch of a node, while solid ones correspond to the true branch.

A BDD is *ordered* if all its variables appear in the same order along each path. A BDD is *reduced* if neither of the following transformations can be applied:

1. Eliminate a node with isomorphic children
2. Combine isomorphic subgraphs.

Starter code was provided in three languages: C, Java, and OCaml, the OCaml version is included in Appendix B. The challenge focuses on reasoning about memory aliasing and abstraction through the following tasks:

1. Verify the memory safety of all operations
2. Verify the termination of all operations
3. Verify the operations are correct:
   (a) Applying `mk_and`(a, b) should produce a node equivalent to the conjunction of the formulas represented by a and b
   (b) Applying `not`(a) should produce a node equivalent to the negation of the formulas represented by a
4. (a) Verify that each operation preserves the properties of a ROBDD
   (b) If the input BDD is ordered, then the output of each operation is an ordered BDD
   (c) Similarly, if the input BDD is reduced, then the output is also reduced.

Comments on solutions. This challenge proved to be too long for the time frame set out in VerifyThis. Though several teams were able to prove properties (1) and (2), they ran out of time to verify the other properties. Proving those properties required establishing that the formula memoized by the BDD is equivalent to the original input (i.e., that A is equivalent to $A \wedge A$). Depending on the implementation of memoization this property could prove challenging to establish.

Team *Morpho Labs* (Quentin Garchery) demonstrated property (3) using Why3. Their solution uses a verified implementation of hashmaps provided by Why3. Usage of this preexisting library allowed them to progress more rapidly. To verify (3), they provided an interpretation function `interp(f, n) -> bool` which interprets the BDD rooted at node `n` using a context `f` mapping variables to true values. This allows them to abstract over the concrete representation of the BDD to focus on its contents. For example, the specification for `mk_not` is

```
let rec mk_not (b : bdd) (n : node) : node
    variant { n }
    ensures { forall f. interp f result = not interp f n }
```

which states that the BDD found at the node `result` contains the negation of the one at `n`.

4 Challenge 3: Nonblocking Concurrent Queue using LL/SC Synchronization

The third challenge dealt with a concurrent FIFO queue, due to Claude Evéquoz, specified using the load-linked/store-conditional (LL/SC) synchronization instructions [4]. As explained in [4], a variable v accessed by these instructions "... can be regarded as a variable that has an associated shared set of thread identifiers" valid_v, which is initially empty. Each of the two instructions behaves like a function call which possibly modifies the valid set and returns a value, in a single atomic step. The semantics are specified in pseudocode as follows:

$\mathsf{LL}(v) \equiv$ $\mathsf{valid}_v \leftarrow \mathsf{valid}_v \cup \{tid\}$;
return v;

$\mathsf{SC}(v, x) \equiv$ **if** $tid \in \mathsf{valid}_v$ **then**
  $\mathsf{valid}_v \leftarrow \varnothing$;
  $v \leftarrow x$;
  return *true*;
else return *false*;

where *tid* is the ID of the thread executing the instruction. Specifically, evaluation of $\mathsf{LL}(v)$ results in the value stored in v, but also has the side-effect of adding *tid* to the valid set associated to v. The result of evaluating $\mathsf{SC}(v, x)$ depends on whether *tid* was in the valid set of v: if so, the valid set is cleared, the value of x is stored in v, and the evaluation result is *true*. If *tid* was not in the valid set of v, then the evaluation result is *false* and no change is made to v or its valid set.

The queue is implemented using an array `Q` as a cyclic bounded buffer. C code for the *enqueue* operation is shown in Figure 2. The code assumes the element type is `int` and that only nonnegative integers are added to the queue. The integer -1 is used to represent a "null" value. The challenge also noted the following:

- Initially, `Head` = `Tail` = 0 and `Q`$[i]$ = `null` = -1 for $0 \le i <$ `LEN`.
- `Head` and `Tail` increase monotonically; for this challenge, assume the `unsigned int` type is unbounded.
- The number of elements stored in the queue is `Tail` − `Head`, and these elements are located at positions `Head`%`LEN`, (`Head` + 1)%`LEN`, ..., (`Tail` − 1)%`LEN` of `Q`.
- Assume a sequentially consistent memory model, i.e., an execution is an interleaved sequence of atomic actions from the different threads, and the value read from a memory location is the last value written to that location.

The challenge posed the following tasks:

1. In your favorite language, write a program P incorporating Evéquoz's FIFO queue (only the enqueue operation is needed). The queue is initially empty. P generates `NT` threads (`NT` ≥ 1), with IDs $0, \ldots,$ `NT` $- 1$. Each thread calls *enqueue* on its thread ID, then terminates.
2. Show that all executions of P terminate (i.e., all threads terminate).

```
int Q[LEN];  // array with indexes in 0..LEN-1
unsigned int Head, Tail;
bool enqueue(int val) {
  unsigned int t, tailSlot;
  int slot;
  while (true) {
    t = Tail;
    if (t == Head + LEN) return false; // queue is full
    tailSlot = t % LEN;
    slot = LL(&Q[tailSlot]);
    if (t == Tail) {
      if (slot != null) {
        if (LL(&Tail) == t) SC(&Tail, t+1);
      } else if (SC(&Q[tailSlot], val)) {
        if (LL(&Tail) == t) SC(&Tail, t+1);
        return true; // success
      }
    }
  }
}
```

Fig. 2. Nonblocking concurrent queue using LL/SC: enqueue operation.

3. Show that no out-of-bound array indexes occur on any execution of P.
4. Assuming $\texttt{NT} \leq \texttt{LEN}$, show that in any execution of P,
   (a) all calls to `enqueue` return *true* (success);
   (b) at the final state, the size of the queue ($\texttt{Tail} - \texttt{Head}$) is $\texttt{NT}$;
   (c) at the final state, the contents of the queue are some permutation of the integers $0, \ldots, \texttt{NT} - 1$.
5. Assuming $\texttt{NT} > \texttt{LEN}$, show that in any execution of P,
   (a) at the final state, the queue is full ($\texttt{Tail} - \texttt{Head} = \texttt{LEN}$)
   (b) the data in the queue is some permutation of a subset of size $\texttt{LEN}$ of $0, \ldots, \texttt{NT} - 1$.

(A second part, dealing with the *dequeue* function, was provided in case anyone finished the first part with time to spare. No one did.)

Comments on solutions. This proved to be the most difficult of the three challenges. In part, this is due to the technology that participants used. The deductive verification tools have limited ability to reason about concurrency. When developing this problem, we had in mind model checking tools, such as Spin [5] or the CIVL Model Checker [6]. Most such tools use languages in which concurrent algorithms can be expressed naturally. Moreover, the reasoning required to check properties is mostly automatic, albeit limited to a finite (usually small) scope.

An excerpt from a CIVL solution to this challenge is shown in Appendix D. The language is primarily C, with additional primitives with names that begin

with `$`. The shared objects are represented using a C struct that bundles an integer value with a bit set representing the set of threads in the object's *valid* set. The `LL` and `SC` functions are defined in the scope of function `thread`, so that the thread ID `tid` is in scope. Function `enqueue` is almost a verbatim copy of the given code. The model checking engine exhaustively explores all interleavings and checks that the assertions (not shown) never fail. For this model, task 4 can be verified for `NT` = `LEN` = 4 in about 25 seconds. Task 5 verifies in under 10 seconds for `NT` = 3 and `LEN` = 4.

Several teams were able to complete or nearly complete tasks 1 and 3. One team was able to show termination for a single thread execution, but in this case the loop terminates after only one iteration.

A classic approach to proving properties of concurrent programs is to show some property of the state is invariant under every atomic action. One team, *Somehow Alex & Marco Returned* (Alex Summers and Marco Eilers), made an interesting attempt along those lines, excerpted in Appendix E. The concurrent program is essentially modeled as a sequential program with a nondeterministic scheduler. The state of a thread is represented using the original variables (`Head`, `Tail`, etc.) together with an integer *program counter* variable `pc`, which can take on approximately 12 values corresponding to specific locations in the code. To execute one atomic step, a thread is chosen nondeterministically. The program switches on the value of the thread's `pc` to a case that performs a single atomic update to the state. This is repeated until no thread is enabled. The goal was to verify that a number of global invariants were preserved at each iteration of this loop. While the solution was not complete, the approach is interesting and we see no reason it could not be extended to a full solution. Of course, it required manual effort to translate the given code to an explicit state machine, and this translation could also introduce errors. If this process could be automated, it could yield an effective approach to deductive verification of concurrent programs.

5 Closing Remarks

Like previous years, the competition awarded several prizes to reward participants. This year the prizes were:

- **Best overall team**: *Somehow Alex & Marco Returned*, Alexander J. Summers and Marco Eilers
- **Best contributed problem**: Jean-Christophe Filliâtre and Andrei Paskevich
- **Best student team**: *Loops and Dots*, Jonas Fiala and Thibault Dardinier
- **Most interesting tool feature**: *Wake Me Up When Verification Ends*, Linard Arquint and Joao Pereira, for the "first-class predicates" of Gobra.

The competition showcased the power of current verification technology, as well as the creativity of the participants. However, it also revealed areas where improvement is needed. Reasoning about pointers and memory continue to challenge the deductive verification tools commonly used at VerifyThis. While linked-list based problems are standard fare in imperative program verification, an

apparently small twist—allowing the list to terminate in a cycle—required predicates, variants and invariants that were very difficult to formulate, especially in separation logic. Reasoning about the memory sharing of BDDs was found to be even more challenging.

Concurrency also remains a stumbling block. The deductive verification tools used at VerifyThis need better ways to express and reason about concurrent programs. Techniques that allow one to easily express a global invariant, and prove that it remains invariant under each atomic action taken by a thread, would be one approach. It may also be the case that students and practitioners of program verification need more practice verifying concurrent algorithms.

In contrast, finite-state verification techniques, such as model checking, are particularly effective at reasoning about concurrent systems, and are generally easily automated. Of course, the downside is that such techniques typically only "prove" that properties hold in some finite scope. The two approaches—deductive and finite-state—have complementary strengths, and a team with expertise in both would certainly do very well at VerifyThis.

As in prior years, the participants of VerifyThis 2023, and other members of the community, have been invited to submit revised or new solutions to the challenges, which will be posted on the Archive of the VerifyThis web site [7].

6 Acknowledgements

Financial support for the prizes was provided by Amazon Web Services. S.F. Siegel was supported by the U.S. National Science Foundation under awards CCF-1955852 and CCF-2019309.

A Challenge 1 Solution

The following is an excerpt from solution to Challenge 1 by team *Wondering How and Why3* (J. Moreau and P. Patault):

```
let rev (ref l: loc) (ghost s: int -> loc) (ghost len p: int): loc
  requires { valid s len }
  requires { len >= 0 }
  requires { segList2R mem s 0 len l p }
  ensures  {
    (old mem).next (s (len-1)) = null -> (* case: non lasso *)
      segList2L mem s 0 len result }
  ensures  {
    (old mem).next (s (len-1)) <> null -> (* case: lasso *)
      segList2R mem s 0 p l p /\ (* initial order until loop *)
      segList2L mem s p len l }
=
  let ref r = null in
  let ghost ref i = 0 in
  while l <> null do
```

```
    invariant { segList2R mem s i len l p }
    invariant { segList2L mem s 0 i r }
    invariant { i <= len }
    variant   { len - i }
    let ref tmp = l in
    l <- get l;
    set tmp r;
    r <- tmp;
    i <- i + 1;
  done;
  r
```

B OCaml starter code for Challenge 2

```
type var = int
type node = Node of { var: int; left: node; right: node; } | True | False

let equal n m = match n, m with
  | True, True -> true
  | False, False -> true
  | Node n, Node m -> n.var = m.var && n.right == m.right &&
                      n.left == m.left
  | _ -> false

module Tbl = Hashtbl.Make(struct
  type t = node
  let equal = equal
  let hash = Hashtbl.hash
end)
type bdd = node Tbl.t
let mk_bdd () : bdd = Tbl.create 32

let mk_node (b : bdd) (n : node) : node = match Tbl.find_opt b n with
  | Some n -> n
  | None -> Tbl.add b n n; n

let mk_true b : node = mk_node b True
let mk_false b : node = mk_node b False

let mk_if b var left right =
  if equal left right then left else
  mk_node b (Node { var; left; right })

let mk_var b v = mk_if b v (mk_true b) (mk_false b)

let rec mk_not (b : bdd) (n : node) : node = match n with
  | True -> mk_false b
  | False -> mk_true b
  | Node { var; left; right } ->
```

```
      mk_if b var (mk_not b left) (mk_not b right)

let rec mk_and (b : bdd) (l : node) (r : node) : node = match l, r with
  | True, _ -> r
  | _, True -> l
  | False, _ | _, False -> mk_false b
  | Node {var = vara; left = lefta; right = righta },
    Node { var = varb; left = leftb; right = rightb } ->
      begin match compare vara varb  with
      | -1 -> mk_if b vara (mk_and b lefta r) (mk_and b righta r)
      | 0 -> mk_if b vara (mk_and b lefta leftb) (mk_and b righta rightb)
      | 1 -> mk_if b varb (mk_and b l leftb) (mk_and b l rightb)
      | _ -> assert false
      end
```

C Challenge 2 Solution

The following is extracted from team *Morpho Labs*'s (Q. Garchery) submission for Challenge 2:

```
type var = int
type node = Node var node node | T | F

let rec ghost function interp (f: var -> bool) node
= match node with
    | T -> true
    | F -> false
    | Node var left right ->
        if f var then interp f left
        else interp f right
    end

val equal (a b : node) : bool ensures { result <-> a = b }

clone hashtbl.Hashtbl as Tbl with type key=node
type bdd = Tbl.t node

let mk_node (b : bdd) (n : node) : node
    ensures { result = n }
= if not Tbl.mem b n then Tbl.add b n n;
    return n

let mk_true b : node
 ensures { forall f. interp f result = true }
= mk_node b T

let mk_false b : node
 ensures { forall f. interp f result = false }
= mk_node b F
```

```
let mk_if b var left right
    ensures { forall f. interp f result =
      if f var then interp f left else interp f right }
= if equal left right then left else
    mk_node b (Node var left right)

let rec mk_not (b : bdd) (n : node) : node
    variant { n }
    ensures { forall f. interp f result = not interp f n }
= match n with
    | T -> mk_false b
    | F -> mk_true b
    | Node var left right -> mk_if b var (mk_not b left) (mk_not b right)
  end
```

D CIVL model of Evéquoz queue

The following is an excerpt from the CIVL model of the Evéquoz queue. The main function (not shown) spawns `NT` threads, each executing `thread`. When $\texttt{NT} \leq \texttt{LEN}$, after all return, it checks the contents of `Q` is some permutation of $0..\texttt{NT}-1$. The complete solution is available at [7], in the 2023 Archive.

```
typedef struct fint { // an int bundled with a valid set
  int data;
  _Bool valid[NT];
} fint;
fint Q[LEN], Head, Tail; // shared variables
void thread(int tid) {
  $atomic_f int LL(fint * x) { // linked-load operation
    x->valid[tid] = 1;
    return x->data;
  }
  $atomic_f _Bool SC(fint * x, int val) {
    // store conditional operation
    if (x->valid[tid]) {
      for (int i=0; i<NT; i++) x->valid[i]=0;
      x->data = val;
      return 1;
    } else { return 0; }
  }
  _Bool enqueue(int val) { // attempts to enqueue val
    int t, tailSlot, slot;
    while (1) {
      t = Tail.data;
      if (t == Head.data + LEN) return FULL_QUEUE;
      tailSlot = t % LEN;
      slot = LL(&Q[tailSlot]);
      if (t == Tail.data) {
```

```
        if (slot != null) {
          if (LL(&Tail) == t) SC(&Tail, t+1);
        } else if (SC(&Q[tailSlot], val)) {
          if (LL(&Tail) == t) SC(&Tail, t+1);
          return OK;
        }
      }
    }
  }
  enqueue(tid); // Body of thread function
}
```

E Challenge 3: Explicit Sequentialization

The following excerpt is from the solution to Challenge 3 submitted by A. Summers and M. Eilers. It shows the use of Viper's statement macros to define `LL` and `SC`, and the explicit sequentialization of the concurrent enqueue function.

```
field valid: Set[Int]
...
define LL(tid, v, target) {
  v.valid := v.valid union Set(tid)
  target := v.val
}
define SC(tid, v, x, target) {
  if (tid in v.valid) {
    v.valid := Set()
    v.val := x
    target := true
  } else {
    target := false
  }
}
...
  while ... {
    var tid : Int
    tid := chooseThreadID(); // pick
    var T : Ref := threads[tid]
    var PC : Int := T.pc
    if(PC == 1) {
      T.t := Tail.val
      T.pc := 2
    } elseif (PC == 2) {
      if(T.t == Head.val + LEN()) {
        T.pc := 11
      } else {
        T.tailSlot := T.t % LEN()
        T.pc := 3
      }
```

```
  } elseif (PC == 3) {
      var ll : Int
      LL(tid,loc(Q,T.tailSlot),ll) // LL(&Q[tailslot])
      T.slot := ll
      T.pc := 4
  }
  ...
}
```

References

1. Akers: Binary decision diagrams. IEEE Transactions on computers **C-27**(6), 509–516 (Jun 1978). https://doi.org/10.1109/TC.1978.1675141
2. Bornat, R.: Proving pointer programs in Hoare logic. In: Backhouse, R., Oliveira, J.N. (eds.) International Conference on Mathematics of Program Construction. Lecture Notes in Computer Science, vol. 1837, pp. 102–126. Springer, Berlin Heidelberg (2000). https://doi.org/10.1007/10722010_8
3. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
4. Evéquoz, C.: Practical, fast and simple concurrent FIFO queues using single word synchronization primitives. In: Reliable Software Technologies—Ada-Europe 2008: 13th Ada-Europe International Conference on Reliable Software Technologies, Venice, Italy, June 16–20, 2008. Lecture Notes in Computer Science, vol. 5026, pp. 59–72. Springer-Verlag, Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-68624-8_5
5. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
6. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The Concurrency Intermediate Verification Language. In: SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York (Nov 2015). https://doi.org/10.1145/2807591.2807635, article no. 61, pages 1–12
7. VerifyThis Competition (web site), http://verifythis.ethz.ch, accessed 14-Feb-2024