Symbolic Execution and Deductive Verification Approaches to VerifyThis 2017 Challenges

Ziqing Luo and Stephen F. Siegel

Verified Software Laboratory, Department of Computer & Information Sciences,
University of Delaware, Newark DE 19716, USA
{ziqing|siegel}@udel.edu

Abstract. We present solutions to the VerifyThis 2017 program verification challenges using the symbolic execution tool CIVL. Comparing these to existing solutions using deductive verification tools such as Why3 and KeY, we analyze the advantages and disadvantages of the two approaches. The issues include scalability; the ability to handle challenging programming language constructs, such as expressions with side-effects, pointers, and concurrency; the ability to specify complex properties; and usability and automation. We conclude with a presentation of a new CIVL feature that attempts to bridge the gap between the two approaches by allowing a user to incorporate loop invariants incrementally into a symbolic execution framework.

Keywords: VerifyThis, symbolic execution, deductive verification, loop invariants, CIVL

1 Introduction

The 2017 VerifyThis program verification competition [6, 9] took place April 22–23, 2017, in Uppsala, Sweden. The competition consisted of three programming/verification challenges. For each, an informal description of one or more algorithms was given, together with properties expected to hold. Participants were allowed to use any languages and tools to implement the algorithms and verify the properties. Teams of one or two members are given 90 minutes to complete each challenge.

A panel of judges evaluated each solution based on criteria such as correctness, completeness, readability, automation, and novelty. Given the variety of languages, verification approaches, and tools used, the judging is necessarily subjective, and reflects the skill of the team members as well as the quality of the verification tools and techniques used. This contrasts with other competitions, such as SV-COMP [3], which are fully automated and use algorithmic judging.

The second author of this paper participated in VerifyThis 2017 using the CIVL verification tool [12], and won the Distinguished Tool Feature award for the ability to automatically verify functional equivalence of two programs. Jean-Christophe Filliâtre won the Best Team award using Why3 [7]. After the competition, participants and others are welcome to publish cleaned up solutions

on the competition web site [6]. VerifyThis challenges are often used in other projects and papers to demonstrate new verification approaches or tool features.

CIVL uses symbolic execution to enumerate a finite state space of a sequential or concurrent C program. It checks assertions in the user code, as well as a number of generic safety properties, including absence of deadlocks, illegal pointer dereferences, divisions by zero, and memory leaks. As a finite state symbolic executor, CIVL requires the user to specify (typically small) bounds on program parameters, such as the sizes of input arrays or the number of processes in a concurrent system, and can only prove the program correct within those bounds. On the other hand, the C code requires very little modification and the tool is highly automated, making it relatively easy to use. This contrasts with the majority of the tools used at VerifyThis, which use deductive verification approaches. These require more in the way of annotations and skill on the part of the user, but are able to prove properties without such bounds.

In this paper, we present CIVL solutions to the 2017 VerifyThis challenges. These solutions are based on those submitted at the competition, but have been improved and extended. We also compare the CIVL solutions with some of those from deductive tools, including Why3, KeY [1] and Frama-C [5], with the goal of illuminating the advantages and disadvantages of each approach. Finally, we introduce a new feature which enables CIVL to use loop invariants to verify some programs without bounds—a symbolic execution analog to the standard use of loop invariants in verification condition generation.

All the CIVL solutions were evaluated using CIVL v1.17.1[1] on a Linux machine with a 3.70GHz Intel Xeon W-2145 CPU. We set the initial and maximum Java heap size to 32 GB and the Java thread stack size to 32 GB. Experimental materials are available at http://vsl.cis.udel.edu/civl/isola18/. Performance statistics for all CIVL solutions are given in Figure 1.

This paper makes the following contributions:

- an exploration of the limits—e.g., scalability, expressive power of the specification language, ability to deal with the complexities of a real programming languages—of a state-of-the-art symbolic execution tool on a challenging set of problems,
- an analysis of the trade-offs—including the need for manual code manipulation, the annotation burden, and the ability to deal with challenging programming language concepts such as pointers and parallelism—inherent in finite-state symbolic execution vs. deductive verification approaches, and
- an exploration of the possibilities of a hybrid approach that combines the best aspects of symbolic execution and deductive techniques.

2 Challenge 1: Pair Insertion Sort

The *pair insertion sort* is a variant of the standard insertion sort which handles two elements at a time. This algorithm occurs in Oracle's Java Development Kit

[1] Download from http://vsl.cis.udel.edu/lib/sw/civl/1.17.1. The SHA1 checksum for `civl-1.17.1_4987.jar` is `65006c21fd77cc5ed6791a9ed33283b95965121d`.

Challenge	Scale	States	Prover Calls	Time (seconds)
Pair-Insertion	$1 \leq \mathtt{n} \leq 5$	13,918	826	19
Pair-Insertion	$1 \leq \mathtt{n} \leq 6$	115,269	6,036	159
Pair-Insertion^{l}	$1 \leq \mathtt{n}$	1,136	226	15
Odd-Even Sort^{s}	$1 \leq \mathtt{n} \leq 4$	11,909	493	11
Odd-Even Sort^{s}	$1 \leq \mathtt{n} \leq 5$	103,606	3,892	87
Odd-Even Sort^{s}	$1 \leq \mathtt{n} \leq 6$	1,041,917	37,001	1,231
Odd-Even Sort^{p}	$1 \leq \mathtt{n} \leq 4$	61,116	499	26
Odd-Even Sort^{p}	$1 \leq \mathtt{n} \leq 5$	476,776	3,952	173
Odd-Even Sort^{p}	$1 \leq \mathtt{n} \leq 6$	4,491,620	37,006	2,000
Tree Buffers^{nc}	$1 \leq \mathtt{n} \leq 6$	1,795,270	1,945	102
Tree Buffers^{nc}	$\mathtt{n} = 7$	15,805,485	10,819	1,009
Tree Buffers^{nr}	$1 \leq \mathtt{n} \leq 6$	3,476,924	1,945	199
Tree Buffers^{nr}	$\mathtt{n} = 7$	30,543,383	10,819	2,190
Tree Buffers^{c}	$1 \leq \mathtt{n} \leq 6$	696,133	2,912	85
Tree Buffers^{c}	$\mathtt{n} = 7$	6,020,981	18,035	707
Tree Buffers^{r}	$1 \leq \mathtt{n} \leq 6$	1,579,100	3,809	169
Tree Buffers^{r}	$\mathtt{n} = 7$	13,392,054	22,970	1,380

Fig. 1. Results of CIVL verification of challenges. The solution marked with l uses loop invariants; s indicates the sequential version; p the parallel version; nc checks the functional equivalence between the naive and caterpillar versions; nr the functional equivalence between the naive and real-time versions; c (resp. r) verifies absence of memory leaks and other run-time errors for the caterpillar (resp. real-time) versions.

(JDK) version 8, class `DualPivotQuicksort.java`, lines 245–274[2]. The challenge description provided that exact Java code snippet, which is also valid C code, and which is identical (except for some white space and unnecessary braces, which we have removed for brevity) to lines 22–32 of Figure 2. The description further stated that "`left` and `right` are valid indices into `a` that set the range to be sorted" and the "implementation is an optimised algorithm which uses the borders `a[left]` and `a[right]` as sentinels." The task is to verify the usual sort properties: the resulting array is sorted and is a permutation of the original.

Our CIVL solution is given in Figure 2. Lines 1–4 specify the inputs and preconditions. The CIVL-C type qualifier `$input` specifies an *input variable*—a read-only variable that represents an input to the program. A concrete value can be specified on the command line, otherwise CIVL uses the value specified in the initializer, and if that is not present the variable is assumed to have an arbitrary value of its type. In symbolic execution, this last case is modeled by assigning a fresh symbolic constant to the variable. In this case there are five inputs: an integer `n` which is the length of the array, an upper bound `N` (given default value 5) for `n`, the original array `A`, and the start and end indexes `LEFT` and `RIGHT`.

[2] http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/DualPivotQuicksort.java

```
1 typedef int T;
2 $input int n, N = 5, LEFT, RIGHT; // n is array size with upper bound N
3 $assume(1<=n && n<=N && 1<=LEFT && LEFT<=RIGHT && RIGHT<n);
4 $input T A[n];
5 int count(int n, T *p, int val) {
6   int result = 0;
7   for (int i=0; i < n; i++) if (p[i] == val) result++;
8   return result;
9 }
10 /* Asserts the sequence starting at p2 is the permutation of that
11  * starting at p1 with elements occurring in nondecreasing order */
12 void assertSortOf(int n, T *p1, T *p2) {
13   $assert($forall (int i : 0..n-2) p2[i]<=p2[i+1]);
14   $assert($forall (int i : 0..n-1) $exists (int j : 0..n-1) p1[i]==p2[j]);
15   for (int i=0; i<n; i++) $assert(count(n, p1, p2[i]) == count(n, p2, p2[i]));
16 }
17 int main() {
18   for (int i=LEFT; i<=RIGHT; i++) $assume(A[LEFT-1]<=A[i]);
19   int a[n];
20   for (int i=0; i<n; i++) a[i]=A[i];
21   int left = LEFT, right = RIGHT;
22   for (int k = left; ++left <= right; k = ++left) {
23     int a1 = a[k], a2 = a[left];
24     if (a1 < a2) { a2 = a1; a1 = a[left]; }
25     while (a1 < a[--k]) a[k + 2] = a[k];
26     a[++k + 1] = a1;
27     while (a2 < a[--k]) a[k + 1] = a[k];
28     a[k + 1] = a2;
29   }
30   int last = a[right];
31   while (last < a[--right]) a[right + 1] = a[right];
32   a[right + 1] = last;
33   $assert($forall (int i:0..LEFT-1) a[i]==A[i]);
34   $assert($forall (int i:RIGHT+1..n-1) a[i]==A[i]);
35   assertSortOf(RIGHT-LEFT+1, A+LEFT, a+LEFT);
36 }
```

Fig. 2. Solution for Challenge 1: Pair insertion sort

Using CIVL, we were able to discover the correct preconditions for the code, though these were not explicit in the challenge description or the final report. First, `LEFT` cannot be *any* valid index; it must be positive. Originally, we had only assumed $0 \leq$ `LEFT`, but CIVL reported an out-of-bound array index at line 25:

```
Violation 0 encountered at depth 29:
CIVL execution violation in p0 (kind: OUT_OF_BOUNDS, certainty: PROVEABLE)
at challenge1-short.cvl:25.16-21
    while (a1 < a[--k]) a[k + 2] = a[k];
                ^^^^^^
possible negative array index: -1
```

After changing the assumption to $1 \leq$ `LEFT`, this error went away. Second, it is actually the element at position `LEFT` $- 1$ that acts as the "sentinel", and the correctness of the algorithm depends on the fact that every element in the range `LEFT..RIGHT` is greater than or equal to that sentinel (line 18). Again, that assumption was deduced easily after CIVL reported an error message without it. Finally, no sentinel is required on the right border: originally we had assumptions on the right which were the mirror image of those on the left, but after removing

them the program still verified successfully. We found this incremental, interactive process, in which assumptions were repeatedly added or modified, CIVL was run, and the output examined to understand a problem, to be very useful in determining the precise assumptions necessary for correctness.

The function `isSortOf` is used to specify the correctness of the post-state. Given two sequences p_1 and p_2 of length n, it asserts that p_2 is nondecreasing; every element of p_1 occurs in p_2; and every element of p_2 occurs in p_1 and with the same multiplicity as in p_2. Since CIVL does not provide any built-in primitive for `count`, this function was defined in ordinary C code. Finally, we also check that the array segments outside of the region `LEFT..RIGHT` are unchanged (lines 33–34).

CIVL verifies this program in 19 seconds (19s) with an upper bound $\texttt{N} = 5$, and in 159s with $\texttt{N} = 6$.

Discussion. The code snippet contains programming constructs which allow the code to be very concise, but also make it difficult to understand and reason about. In particular, it has expressions with side-effects, namely the pre/post-increment and decrement operators. Furthermore, these are used within loop conditions of `while` loops and in both the incrementer and condition expressions of `for` loops. CIVL deals with these by first transforming the program into a normal form which has no side-effect expressions. This involves introducing auxiliary variables and restructuring certain loops. This internal transformation is usually invisible to the user, though there is an option that will cause CIVL to print the transformed program.

Verification tools for languages that do not have such constructs do not have to deal with these issues. But to apply those tools to code such as the pair-insertion snippet, one must either transform the code manually (which can be error-prone) or use another tool as a front end. In the case of this Challenge, the competition organizers also provided pseudocode for a simplified version of the algorithm. The simplified version uses only `while` loops, has no side-effect expressions, and does not have `left` and `right`, but instead just sorts the entire array. The sentinel assumptions described above do not arise in the simplified version. While the ultimate goal was to verify the original code, most teams used the simplified code; the winning Why3 team fully verified the simplified version for arbitrary inputs.

Like CIVL, Frama-C operates on C code, though the Frama-C solution[3] is based on the simplified version of the code. That solution adds approximately 50 lines of ACSL[4] annotations to the code. The annotations specify loop invariants, function contracts, and an inductive predicate for specifying that an array at one state is a permutation of the same array at another state. The deductive verification was carried out by the Frama-C WP plug-in. This solution verifies

[3] https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Solutions%202017/program.c

[4] ANSI/ISO C Specification Language. http://frama-c.com/acsl.html

```
let pair_insertion_sort (a: array int)
= let i = ref 0 in (* i is running index (inc by 2 every iteration)*)
  while !i < length a - 1 do
    let x = ref a[!i] let y = ref a[!i + 1] in
    if !x < !y then (* ensure that x is not smaller than y *)
      begin let tmp = !x in x := !y; y := tmp end (* swap x and y *)
    end;
    let j = ref (!i - 1) in while !j >= 0 && a[!j] > !x do
      j := !j - 1
    done;
    a[!j + 2] <- !x; (* store x at its insertion place *)
    while !j >= 0 && a[!j] > !y do (* #ind the insertion point for y *)
      j := !j - 1
    done;
    a[!j + 1] <- !y; i := !i + 2
done;
if !i = length a  - 1 then begin (* if length(A) is odd, an extra  *)
  let y = a[!i] in let j = ref (!i - 1) in
  while !j >= 0 && a[!j] > y do
    j := !j - 1
  done;
  a[!j + 1] <- y
end
```

Fig. 3. Executable portion of WhyML model of simplified pair insertion sort (excerpt)[5]

the sortedness of the resulting array, but it fails to prove that the result is a permutation of the initial array. In addition, Frama-C verified that the program is free of runtime errors such as division by zero or integer overflow.

CIVL also checks for certain runtime errors, such as division by zero, out-of-bound array indexes, and null pointer dereferences —all of which are checked by Frama-C+WP as well. Unlike Frama-C, CIVL does not currently check for integer overflows, because numerical values are modeled with the mathematical integers or reals in CIVL. CIVL does have a precise model of memory heaps and `malloc`/`free`, and verifies absence of double-frees, memory leaks, use-after-free, and similar properties. In contrast, reasoning about programs with dynamic memory allocation is currently very difficult with Frama-C+WP.

The Why3 solution[5] is a WhyML model of the simplified algorithm (Figure 3) with function contracts, loop invariants, and some intermediate assertions; it consists of approximately 90 lines. The target properties are expressed as post-conditions:

```
ensures { forall k l. 0 <= k <= l < length a -> a[k] <= a[l] }
ensures { permut_all (old a) a }
```

The WhyML program generates 113 proof obligations (after splitting), which can be proved in parallel.

WhyML is a language designed for verification—it includes high-level abstractions that make it easy to specify properties and algorithms, and avoids many of the tricky programming language constructs in languages such as C (e.g., pointers and pointer arithmetic, memory management, side-effect expressions, preprocessing, linking, and I/O functions). However, it is possible to gen-

[5] http://toccata.lri.fr/gallery/verifythis_2017_pair_insertion_sort.en.html

erate OCaml code from the verified WhyML program, and in this sense Why3 is also able to verify correctness of programs in a "real" programming language. This solution fully verifies that the resulting array is a sorted permutation of the initial array. It uses existing array permutation theories in the Why3 library, which reduces the specification burden.

We were able to find only one other successful verification of the original Java code. This solution uses KeY, and adds approximately 60 annotation lines to the code[6]. It verifies the final array is ordered, but does not verify that it is a permutation of the original.

For all of these solutions, the loop invariants are non-trivial and are discussed further in §5.

3 Challenge 3: Odd-even Transposition Sort

Challenge 3[7] presented another sorting algorithm. The odd-even transposition sort,

> "...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...swaps the elements. The next step repeats this for even-indexed list elements.... The algorithm iterates between these two steps until the list is sorted." [9]

```
1 #include <stdbool.h>
2 typedef double T;
3 $input int n, N = 5; // N is upper bound on array size n
4 $assume(1<=n && n<=N);
5 $input T A[n]; // input array: constant, fixed forever
6 void swap(T *p, int i, int j) { T temp = p[i]; p[i] = p[j]; p[j] = temp; }
7 void oddEvenSort(int n, T *list) {
8   _Bool sorted = false;
9   while (!sorted) {
10    sorted = true;
11    for (int i = 1; i < n-1; i += 2)
12      if (list[i] > list[i+1]) { swap(list, i, i+1); sorted = false; }
13    for (int i = 0; i < n-1; i += 2)
14      if (list[i] > list[i+1]) { swap(list, i, i+1); sorted = false; }
15  }
16 }
17 int main() {
18   T a[n];
19   for (int i=0; i<n; i++) a[i] = A[i];
20   oddEvenSort(n, a);
21   assertSortOf(n, A, a); // see definition in Figure 2
22 }
```

Fig. 4. Sequential solution for challenge 3: odd-even transposition sort

[6] https://www.key-project.org/wp-content/uploads/2017/08/PairInsertionSort.java

[7] Challenge 2 was replaced by another challenge shortly before the competition began, hence the gap in the numbering.

```
#include <mpi.h>
#define T double
#define MPI_T MPI_DOUBLE
$input int n, N = 4;
$assume(0 < n && n <= N);
$input T A[n];
T myvalue; // this is a local variable for each process in MPI
void oddEvenPar(int n, int id) {
  T othervalue;
  for (int i=0; i<n; i++) {
    if ((i+id)%2 == 0) {
      if (id < n-1) {
        MPI_Send(&myvalue, 1, MPI_T, id+1, 0, MPI_COMM_WORLD);
        MPI_Recv(&othervalue, 1, MPI_T, id+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (othervalue < myvalue) myvalue = othervalue;
      }
    } else {
      if (id > 0) {
        MPI_Send(&myvalue, 1, MPI_T, id-1, 0, MPI_COMM_WORLD);
        MPI_Recv(&othervalue, 1, MPI_T, id-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (othervalue > myvalue) myvalue = othervalue;
      }
    }
  }
}
int main() {
  int nprocs, rank;
  MPI_Init(NULL, NULL); // initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // get the number of procs
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get my "rank" (id)
  $assume(nprocs == n); // number of procs must equal n, prune other cases
  myvalue = A[rank]; // get my value from the global input array A
  oddEvenPar(nprocs, rank); // call the function that does the work
  if (rank == 0) { // proc 0 will gather all data and check it is correct
    T a[n]; // used for verification---gather all elements into one array on proc 0
    MPI_Gather(&myvalue, 1, MPI_T, a, 1, MPI_T, 0, MPI_COMM_WORLD);
    assertSortOf(n, A, a); // see definition in Figure 2
  } else
    MPI_Gather(&myvalue, 1, MPI_T, NULL, 0, MPI_T, 0, MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Fig. 5. Parallel solution for challenge 3: odd-even transposition sort

Pseudocode for a sequential and a parallel version was given. Application of CIVL to the sequential version is straightforward; see Figure 4. We have reused the `assertSortOf` function from Challenge 1. The definition of function `oddEvenSort` is almost identical to the given pseudocode; we have added only the definition of `swap`. CIVL verifies the program with the upper bound `N` = 4 in 11s; `N` = 5 in 87s; and `N` = 6 in 1,231s.

The pseudocode for the parallel version is expressed using the message-passing style. Each process maintains one element in its local memory and performs exchanges with its left and right neighbors. CIVL supports MPI[8], the standard message-passing API, and the pseudocode maps directly to the C/MPI code in function `oddEvenPar` of Figure 5. We have added a driver in the `main` function which (1) includes standard MPI boilerplate, (2) initializes the local

[8] A Message-Passing Interface Standard, Version 3.1. http://www.mpi-forum.org/docs/

variable `myvalue` on each process by reading from the `$input` array `A`, (3) invokes `oddEvenPar`, and then (4) gathers the final result into a single array `a` on process 0, which (5) checks the result using the same function `assertSortOf` we used in the other sorting routines. The input variable `n` is the exact size of the array, which is also the number of processes, and `N` is an upper bound on `n`. CIVL verifies the case `N` = 4 in 26s; `N` = 5 in 173s; and `N` = 6 in 2,000s.

Discussion. The MPI functions used in the code are defined using lower-level CIVL concurrency primitives in CIVL's implementation of `mpi.h`. The MPI API is quite large, comprising hundreds of functions, types, and constants, and the CIVL implementation is not complete. However, it does support the most commonly-used MPI primitives, including the blocking, standard-mode send, receive, and send-receive operations, the primitive datatypes, multiple communicators, wildcard sources and tags, and the blocking collective operations (such as `MPI_Gather`). These suffice for this problem, and for many real-world applications. Support for more advanced MPI constructs is in progress.

There is a minor limitation: it is not possible to use a typedef name as the type of an input variable when using MPI, due to code transformations performed by CIVL which re-order certain declarations, so we used a preprocessor macro definition for `T` instead. This will be addressed in a future release of CIVL.

The Why3 solution fully verifies the sequential version of this challenge.[9] The solution uses standard theories, e.g., for array permutations, array swaps, and integer division. It also defines a predicate `odd_sorted` which states that each odd-indexed element of an array is less than or equal to the next (even-indexed) element in the array. A predicate `even_sorted` is defined similarly for the even-indexed elements. A lemma claims that if an array is `odd_sorted` and `even_sorted` then it is sorted. The lemma includes a few "hints" for its proof in the form of `by` directives:

```
let lemma odd_even_sorted (a: array int) (n: int)
    requires { 0 <= n <= length a }
    requires { odd_sorted a n }
    requires { even_sorted a n }
    ensures  { sorted_sub a 0 n }
  = if n > 0 && length a > 0 then
    for i = 1 to n - 1 do
      invariant { sorted_sub a 0 i }
      assert { forall j. 0 <= j < i -> a[j] <= a[i]
                  by a[i-1] <= a[i]
                  by i-1 = 2 * div (i-1) 2 \/
                     i-1 = 2 * div (i-1) 2 + 1 }
    done
```

The lemma is proved independently and used as an axiom while proving the algorithm. This kind of proof decomposition is common in deductive verifica-

[9] http://toccata.lri.fr/gallery/verifythis_2017_odd_even_transposition_sort.en.html

```
1 struct Node {
2   Node * parent;
3   int data;
4 };
5 struct Tree {
6   int history;
7   Node * last; // reference to the last added node
8 };
9 Node * make_node(int data) {
10   Node * node = (Node*)malloc(sizeof(Node));
11   node->parent = NULL; node->data = data; return node;
12 }
13 int get_data(Node * node) { return node->data; }
14 Tree * empty(int history) {
15   Tree * tree = (Tree*)malloc(sizeof(Tree));
16   tree->history = history; tree->last = NULL; return tree;
17 }
18 Tree * add(int data, Tree * tree) {
19   Node * child = make_node(data);
20   Tree * new_tree = empty(tree->history);
21   if (tree->last == NULL) new_tree->last = child;
22   else { child->parent = tree->last; new_tree->last = child; }
23   return new_tree;
24 }
25 void get(Tree * tree, Node * ancestors[]) {
26   int i = 0, history = tree->history;
27   Node * ancestor = tree->last; // "ancestors" is a NULL-terminated array
28   while (i < history && ancestor != NULL) {
29     ancestors[i++] = ancestor; ancestor = ancestor->parent;
30   }
31   ancestors[i] = NULL;
32 }
33 void delete(Tree * tree) {}
```

Fig. 6. The Naive Implementation of the Tree Buffer

tion, but is unnecessary in bounded symbolic execution. This WhyML program generates 57 proof obligations.

The Why3 team did not attempt to verify the parallel version because WhyML has no model of concurrency. As far as we know, VerCors [4] is the only other team that has attempted to verify the concurrent algorithm [6]. The VerCors team wrote the parallel algorithm as a GPU kernel function and attempted to verify it against a specification, but the global invariant remains unproved.

4 Challenge 4: Tree Buffers

Challenge 4 describes an abstract data structure called a *tree buffer*. It is essentially an immutable list which only needs to "remember" the h most recent elements added, for a specified constant integer h ("history"). It supports three operations: *empty* takes h and returns an empty tree buffer with parameter h associated to it; *add* consumes an element x and a tree buffer t, and returns a new tree buffer (with the same h as that associated to t) obtained by adding x to the front of t; *get* consumes a tree buffer and returns its first h elements, or all of its elements if it has fewer than h elements. Because it is observably immutable, sharing is allowed, e.g.,

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

defines four distinct tree buffers, but each may be contained in a single tree of 4 nodes. The interface and a simple naive implementation are given in OCaml. We have translated the naive implementation to C in Figure 6.

The naive implementation is memory inefficient: nodes are never removed, yet it is only required to remember at most h nodes for each tree buffer. The description then provides a *caterpillar* implementation in OCaml which offers a solution to this problem. The first task is to show the naive and caterpillar implementations are functionally equivalent, using any imperative language for the caterpillar implementation. Our C implementation of the caterpillar is given in Figure 7.

To establish functional equivalence, we first created a common interface for tree buffers, in the form of the header file `treebuffer.h`, shown in Figure 8. We then created a driver that can be linked to any tree buffer implementation, as it uses only the header file; see Figure 9. The idea is to show that any sequence of *get* and *add* operations will result in the same output from the two implementations. We take it as clear that *get* does not modify any tree buffer, so any two *get* operations commute. It therefore suffices to consider an arbitrary sequence of *add* operations followed by a single arbitrary *get* operation, and show the result must be the same across implementations.

The driver works by first creating an empty node, then repeatedly choosing an existing node and adding a child to it. In the end, it chooses a node to which to apply *get*, and records the result in an *output* variable. The choices in the driver are controlled by input variables, e.g., the input array `PICKS` determines which node will be chosen at each step to be the parent of the new node. Hence the program comprising the driver and a tree buffer implementation is a deterministic function of its input.

CIVL has the ability to verify that two programs with compatible input-output signatures are functionally equivalent. In keeping with the general CIVL philosophy, this is accomplished primarily by code transformation: the new programs are merged into a single program that invokes each constituent program sequentially, on the same inputs, and then asserts that the outputs agree. This merged program is verified as usual. In this case, CIVL is able to verify the equivalence of the naive and caterpillar implementations, with an upper bound of 6 on the number of nodes, in 102s. For exactly 7 nodes, the time is 1,009s. When using CIVL in `compare` mode, the standard runtime properties are also checked, but in this case we turned off the check for memory leaks because the naive version (intentionally) does not free any of the memory it allocates. The command used looks like:

```
civl compare -checkMemoryLeak=false -inputN=6 -spec treebuffer-driver.cvl
  treebuffer-naive.c -impl treebuffer-driver.cvl treebuffer-realtime.c
```

The caterpillar implementation depends on a garbage collector, which we modeled in our C implementation with function `gc`. The next task involves an

```
1 struct Node {
2   Node * parent;
3   int data, ref_count; // reference counter
4   int num_ancest; // number of ancestors
5 };
6 struct Tree {
7   int history;
8   Node * xs, * ys; // xs: reference to the last added node, ys: caterpillar
9 };
10 void gc(Node * node) { // garbage collection
11   if (node == NULL || node->ref_count > 0) return;
12   Node * next = node->parent;
13   free(node);
14   if (next != NULL) {next->ref_count--; gc(next);}
15 }
16 Node * make_node(int data) {
17   Node * node = (Node*)malloc(sizeof(Node));
18   node->parent = NULL;  node->data = data; node->ref_count = 0;  node->num_ancest = 0;
19   return node;
20 }
21 int get_data(Node * node) { return node->data; }
22 Tree * empty(int history) {
23   Tree * tree = (Tree*)malloc(sizeof(Tree));
24   tree->history = history; tree->xs = NULL; tree->ys = NULL; return tree;
25 }
26 Tree * add(int data, Tree * tree) {
27   Node * node = make_node(data);
28   Tree * new_tree = empty(tree->history);
29   if (tree->xs == NULL) {
30     new_tree->xs = node; new_tree->xs->ref_count++; new_tree->ys = tree->ys;
31     if (tree->ys!=NULL)   tree->ys->ref_count++;
32     else if (tree->xs->num_ancest < tree->history - 2) {
33       node->parent = tree->xs;  node->parent->ref_count++;
34       node->num_ancest = node->parent->num_ancest + 1;
35       new_tree->xs = node;  new_tree->xs->ref_count++; new_tree->ys = tree->ys;
36       if (tree->ys != NULL) new_tree->ys->ref_count++;
37       else {
38         node->parent = tree->xs; node->parent->ref_count++;
39         node->num_ancest = node->parent->num_ancest + 1;
40         new_tree->ys = node; new_tree->ys->ref_count++;
41       }
42     }
43   }
44   return new_tree;
45 }
46 void get(Tree * tree, Node * ancestors[]) {
47   int i = 0, history = tree->history;
48   Node * ancestor = tree->xs; // "ancestors" is a NULL-terminated array
49   while (i < history && ancestor != NULL) {
50     ancestors[i++] = ancestor; ancestor = ancestor->parent;
51   }
52   ancestor = tree->ys;
53   while (i < history && ancestor != NULL) {
54     ancestors[i++] = ancestor; ancestor = ancestor->parent;
55   }
56   ancestors[i] = 0;
57 }
58 void delete(Tree * tree) {
59   if (tree->ys != NULL) { // take ys off and collect garbage:
60     Node * ys = tree->ys; tree->ys = NULL; ys->ref_count--; gc(ys);
61   }
62   if (tree->xs != NULL) {
63     Node * xs = tree->xs;  tree->xs = NULL;  xs->ref_count--; gc(xs);
64   }
65   free(tree);
66 }
```

Fig. 7. The Caterpillar Implementation of the Tree Buffer

```
#ifndef TREEBUFFER_H
#define TREEBUFFER_H
typedef struct Node Node;
typedef struct Tree Tree;
void delete(Tree * tree);
void get(Tree * tree, Node * ancestors[]);
Tree * add(int data, Tree * tree); /* returns new tree with a new node added to given tree */
Tree * empty(int history); /* Creates a new tree */
int get_data(Node * node); /* Gets the data in a node */
#endif
```

Fig. 8. The common header `treebuffer.h` for all tree buffer implementations

```
#include "treebuffer.h"
$input int n, H, NODE_TO_READ, N = 5;
$assume(0 < n && n <= N && 0 < H && H < n && 0 <= NODE_TO_READ && NODE_TO_READ < n);
$input int DATA[n-1], PICKS[n];
$assume($forall (int i : 0 .. n-1) 0 <= PICKS[i] && PICKS[i] <= i);
$output int out[H]; // output
int main() {
  Tree * node = empty(H), * nodes[n]; // n new refs to node + empty()
  int num_nodes = 0;
  nodes[num_nodes++] = node; $elaborate(n);
  while (num_nodes < n) {
    int idx = PICKS[num_nodes-1];
    node = nodes[idx]; $elaborate(idx);
    Tree * new_node = add(DATA[num_nodes-1], node);
    nodes[num_nodes++] = new_node; node = new_node;
  }
  Node * history[H + 1];
  int i;
  $elaborate(NODE_TO_READ); get(nodes[NODE_TO_READ], history);
  for (i = 0; i < H; i++)
    if (history[i] != NULL) out[i] = get_data(history[i]); else break;
  for (; i < H; i++) out[i] = -1;
  for (int i = 0; i < num_nodes; i++) delete(nodes[i]);
}
```

Fig. 9. The CIVL-C Driver for Tree Buffer implementations. The driver performs an arbitrary sequence of `add` operations and finally a single arbitrary `get` operation to yield the output. All choices are controlled by the input variables.

even more complex "real-time" implementation that manually manages memory in a time and space-efficient way. The description presents untested C++ pseudocode, and references a git repository with a well-tested but much more verbose C implementation[10], which is 329 lines long. The second task is to show that one of these real-time implementations is also functionally equivalent to the naive implementation.

We downloaded the C implementation, and made very minor modifications to it so that it implements the interface in our header file. (We renamed the `Tree` structure `TBTree` and added a shell structure `Tree` to wrap a `TBTree`.) Using the same driver, CIVL verified equivalence with the naive implementation, taking 199s for the number of nodes $\mathtt{n} \leq 6$; and 2,190s for $\mathtt{n} = 7$.

CIVL is also able to verify that neither the caterpillar nor the real-time implementation has a memory-leak: every malloc-ed object is eventually freed.

[10] https://github.com/rgrig/treebuffers/blob/master/treebuffer.c

This is accomplished by verifying each program normally (without comparing to the naive version); results are shown in Figure 1.

A final task is to verify the space efficiency property of the real-time implementation, i.e., to show that the memory usage is bounded by a constant factor of the live-heap size of the caterpillar version. We have constructed a solution to this challenge as well, but space does not permit us to discuss it here. The solution can be found in the online archive.

Discussion. The Why3 team implemented all three versions in WhyML.[11] The functional equivalence of the naive version with the advanced versions is expressed by adding a ghost field to the data structures in the advanced versions. The ghost field has the type of the structure used in the naive implementation and each advanced operation also updates the ghost field using the corresponding naive operation. A postcondition captures this behavior by stating that the resulting value of the ghost field equals the value obtained by invoking the naive version. Finally, an invariant states that the data in the ghost field always corresponds to the data in the non-ghost part of the structure.

The real-time version is based on the given C++ code. The entire Why3 solution comprises less than 160 lines of code and generates 33 proof obligations.

Neither the KeY nor the Frama-C solutions deal directly with functional equivalence of different implementations. Unlike WhyML, the languages of those tools do not permit function calls in specifications. The KeY team did implement both the naive and caterpillar versions and verified them separately against contract specifications.[12]

We are not aware of any application of verification tools to the C implementation of the real-time tree buffer algorithm, other than the CIVL one described above.

5 Conclusion and Future Directions

Of the two verification approaches—symbolic execution and deductive reasoning—there is no "better", but each has different advantages and disadvantages. Our examination of solutions to VerifyThis 2017 challenges confirms earlier impressions that symbolic execution is easier to use: it is easier to automate, it requires less in the way of code annotation and modification, it is better able to support complexities such as concurrency (including for C/MPI programs), it provides useful error traces when properties are violated, and it requires less sophistication in logic and specification [11].

To compare execution times, we verified the Why3 solutions on the same platform used for the CIVL runs. The Why3 verification process is not always

[11] http://toccata.lri.fr/gallery/verifythis_2017_tree_buffer.en.html

[12] https://www.ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Verify%20This/Solutions%202017/Kirsten_TreeBuffer.java

fully automatic, even when using only automated provers in the back end, as it can require user interaction to select provers and strategies. Using Why3 1.0.0, we specified a custom strategy that first applies the *split_goal_right* transform; then applies provers in the following order, until a conclusive result is obtained: Alt-Ergo 1.30, Z3 4.4.1, E theorem prover 1.8, and CVC3 2.4.1, each with 1s timeout and 1GB memory; then CVC4 1.5 with 3s timeout and 1GB memory. With this strategy, all programs were proved without further interaction. It took approximately 5s and 162 prover calls to verify Challenge 1, 2s and 90 calls for Challenge 3 (sequential part), and 10s and 60 calls for Challenge 4.

The Why3 solutions are quite different programs from the C programs verified by CIVL (especially in Challenge 4), so one cannot conclude much from this comparison. Nevertheless, these results indicate that symbolic execution is generally more computationally intensive than deductive verification, except possibly when using very small bounds. The same conclusion can be drawn by comparing the number of theorem prover calls. This is not surprising: finite-state symbolic execution is essentially a "brute-force" approach in which nondeterministic choices are explored exhaustively. This is also the main reason execution time blows up as parameter `n` increases in the CIVL solutions.

We also see some advantages in the rich logical theories provided by deductive tools such as Why3, e.g., support for a theory of permutation, which simplify the specification task.

The primary drawback of symbolic execution is that it requires some way to bound the number of iterations of each loop. Earlier work, e.g., [2,8,10,13], has explored ways to incorporate loop invariants into the basic symbolic execution algorithm in order to overcome this limitation. We have recently added a similar capability to CIVL.

Our approach has two advantages.

First, it is incremental. This means a user can start by developing a simple bounded program, as in the preceding examples of this paper. Once that is working, the user can add loop invariants for a single loop, and (possibly) get some advantage, such as the ability to remove the bounds on a parameter. Once that is working, the user can move on to a second loop, and so on. At any point in this process, one has a useful model that is at least as good as the preceding one. Once every loop is annotated, the program can be verified without any bounds, exactly as in deductive verification.

Second, our approach is language-based. Instead of implementing extensive changes in the verification kernel, we have added a small number of new, general-purpose language primitives. Code with loop invariants is transformed by the CIVL front end to one without invariants, but with uses of these new primitives. This is in keeping with the general philosophy of CIVL, which is to support a small number of general primitives well, and do the rest of the work by source code transformation. This keeps the symbolic execution engine simple and easy to optimize and maintain. The new primitives are as follows:

1. `$assuming(`*`expr`*`)` *`stmt`* adds the assumption *`expr`* to the current path condition, but only for the duration of the execution of *`stmt`*. The path condition

is actually represented as a stack of conditions; *expr* is pushed onto the stack at the beginning, and popped after *stmt* completes;
2. $mem is a type representing a set of memory locations;
3. $mem_contains(*m1*,*m2*) is a boolean function that return true iff the set of memory locations represented by *m1* contains all of those represented by *m2*;
4. $mem_havoc(*m*) "havocs" (assigns arbitrary values to) all the memory locations in the $mem *m*;
5. $capture *stmt* is an expression that is evaluated by executing *stmt* and returning a $mem consisting of all memory locations that were modified in the course of executing *stmt*.

A loop annotated with an invariant and a frame condition has the form

```
/*@ loop invariant θ;
  @ loop assigns Δ; */
while (b) C
```

and is transformed to

```
$assert(θ); // check loop invariant establishment
while ($choose_bool()) {  // nondeterministic choice: stay or exit
  $assuming(b && θ) {
    $mem ws = $capture C // execute body, capturing the write set
    $assert($mem_contains(Δ, ws)); // check the frame condition
    $assert(θ); // check loop invariant preservation
    $mem_havoc(Δ);
  }
}
$assume(¬b && θ);
```

It is not hard to see this transformation is sound: if θ is not a loop invariant or C can write to a memory location not in Δ, an assertion will be violated. Otherwise, all of the executions in the original program correspond to executions in the transformed program. The transformation is effective because the symbolic constants used in the havoc operation are renamed in a canonical way at each step, which enables the loop to eventually return to a state that has been seen before. In our experience, most loops converge after two or three iterations. The result is a finite symbolic state space that can be explored in the usual way.

Loop invariants are expressed in ACSL. Support is still partial; limitations include (1) there are some cases where the loop does not converge, and (2) in some cases, the original code has to be modified by hand to remove side-effects.

We have successfully applied this new CIVL capability to verify the sorted property of the pair insertion sort, without any bounds on the input size. The CIVL input is given in Figure 10. This program was verified by CIVL in 15s. Support for the permutation property is in progress.

While there are similarities between this approach and that of verification generation tools such as Why3, there are significant differences. Symbolic execution may explore many paths through the loop, each with a different path condition. This has the potential to generate more, but simpler, theorem prover

```
#pragma CIVL ACSL
$input int n, LEFT, RIGHT;
$assume(1<=n && 1<=LEFT && LEFT <= RIGHT && RIGHT<n);
$input int A[n];
int main() {
  $assume($forall (int j : LEFT .. RIGHT) A[j] >= A[LEFT-1]);
  int a[n], left = LEFT, right = RIGHT, k = left;
  memcpy(a, A, n * sizeof(int)); left++;
  /*@ loop invariant 2 <= left <= right + 2 && (k == left - 1);
    @ loop invariant LEFT <= k <= RIGHT+1 && k == left - 1;
    @ loop invariant \forall int t; LEFT <= t < k ==> a[t - 1] <= a[t];
    @ loop invariant \forall int t; LEFT <= t <= RIGHT ==> a[LEFT-1] <= a[t];
    @ loop assigns k, left, a[LEFT .. n-1]; @*/
  for (; left <= right; k = left++) {
    int a1 = a[k], a2 = a[left];
    if (a1 < a2) { a2 = a1; a1 = a[left]; }
    k--;
    /*@ loop invariant LEFT - 1 <= k && k < left - 1;
      @ loop invariant a1 < a[k] ==> k >= LEFT;
      @ loop invariant \forall int t; LEFT <= t <= k ==> a[t - 1] <= a[t];
      @ loop invariant \forall int t; k + 3 < t <= left ==> a[t - 1] <= a[t];
      @ loop invariant \forall int t; LEFT <= t <= RIGHT ==> a[LEFT-1] <= a[t];
      @ loop invariant k + 3 <= left ==> a[k + 3] >= a1 && a[k+3] >= a[k];
      @ loop assigns k, a[LEFT .. n-1]; @*/
    while (a1 < a[k]) { a[k + 2] = a[k]; k--; }
    a[++k + 1] = a1; k--;
    /*@ loop invariant LEFT - 1 <= k < left - 1 && (a2 < a[k] ==> k >= LEFT);
      @ loop invariant \forall int t; LEFT <= t <= k ==> a[t - 1] <= a[t];
      @ loop invariant \forall int t; k + 2 < t <= left ==> a[t - 1] <= a[t];
      @ loop invariant \forall int t; LEFT <= t <= RIGHT ==> a[LEFT-1] <= a[t];
      @ loop invariant k + 2 <= left ==> a[k + 2] >= a2 && a[k + 2] >= a[k];
      @ loop assigns k, a[LEFT .. n-1]; @*/
    while (a2 < a[k]) { a[k + 1] = a[k]; k--;}
    a[k + 1] = a2; left++;
  }
  int last = a[right--];
  /*@ loop invariant LEFT-1 <= right <= RIGHT-1 && (right < RIGHT-1 ==> last < a[right+1]);
    @ loop invariant right < RIGHT-1 ==> (\forall int i; LEFT <= i <= RIGHT ==>
    @                a[i-1] <= a[i]);
    @ loop invariant right == RIGHT-1 ==> (\forall int i; LEFT <= i <= RIGHT-1 ==>
    @                a[i-1] <= a[i]);
    @ loop assigns right, a[LEFT .. RIGHT]; @*/
  while (last < a[right]) { a[right + 1] = a[right]; right--;}
  a[right + 1] = last;
  $assert( $forall (int i : LEFT .. RIGHT-1) a[i] <= a[i+1] );
}
```

Fig. 10. The Loop Invariant Solution for Challenge 1: Pair Insertion Sort

queries. For this problem, Why3 verifies all properties of its solution with 162 prover calls, while CIVL takes 226 prover calls to verify only sortedness. It will take further investigation to understand the differences and trade-offs in the two approaches.

This research was supported by the U.S. National Science Foundation under Award CCF-1319571, and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of Unstructured Programs. In: Ernst, M.D., Jensen, T.P. (eds.) Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5-6, 2005. pp. 82–87. ACM (2005). https://doi.org/10.1145/1108792.1108813
3. Beyer, D.: Software verification with validation of results. In: Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206. pp. 331–349. Springer-Verlag, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
4. Blom, S., Huisman, M.: The VerCors Tool for Verification of Concurrent Programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014: Formal Methods. pp. 127–131. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
5. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing **27**, 573–609 (2012). https://doi.org/10.1007/s00165-014-0326-7
6. Eidgenössische Technische Hochschule Zürich: Chair of Programming Methodology. http://www.pm.inf.ethz.ch/research/verifythis/Archive/2017.html (2017)
7. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Conference on Programming Languages and Systems. pp. 125–128. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
8. Hentschel, M., Bubel, R., Hähnle, R.: The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. International Journal on Software Tools for Technology Transfer (Mar 2018), https://doi.org/10.1007/s10009-018-0490-9
9. Huisman, M., Monahan, R., Müller, P., Mostowski, W., Ulbrich, M.: VerifyThis 2017: A Program Verification Competition. Tech. Rep. Karlsruhe Reports in Informatics 2017,10, Karlsruhe Institute of Technology, Faculty of Informatics (2017). https://doi.org/10.5445/IR/1000077160
10. Păsăreanu, C.S., Visser, W.: Verification of Java Programs Using Symbolic Execution and Invariant Generation. In: Graf, S., Mounier, L. (eds.) Model Checking Software. pp. 164–181. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24732-6_13
11. Siegel, S.F.: CIVL solutions to VerifyThis 2016 challenges. ACM SIGLOG News **4**(2), 55–75 (May 2017), http://doi.acm.org/10.1145/3090064.3090070
12. 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: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 61:1–61:12. SC '15, ACM, New York (2015), http://doi.acm.org/10.1145/2807591.2807635
13. Siegel, S.F., Zirkel, T.K.: Loop invariant symbolic execution for parallel programs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA,

USA, January 22–24, 2012, Proceedings. LNCS, vol. 7148, pp. 412–427. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_27