# Using Dafny to Solve the VerifyThis 2021 Challenges*

Marie Farrell
marie.farrell@mu.ie
Maynooth University
Maynooth, Co. Kildare, Ireland

Conor Reynolds
conor.reynolds@mu.ie
Maynooth University
Maynooth, Co. Kildare, Ireland

Rosemary Monahan
rosemary.monahan@mu.ie
Maynooth University
Maynooth, Co. Kildare, Ireland

## ABSTRACT

This paper provides an experience report of using the Dafny program verifier, at the VerifyThis 2021 program verification competition. The competition aims to evaluate the usability of logic-based program verification tools in a controlled experiment, challenging both the verification tools and the users of those tools. We present the two challenges that we tackled during the competition and discuss our solutions. As a result, we identify strengths and weaknesses of Dafny in the verification of relatively complex algorithms, and report on our experience of applying Dafny in this setting.

## CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

## KEYWORDS

Deductive Verification, VerifyThis, Dafny, Verification Challenges

**ACM Reference Format:**
Marie Farrell, Conor Reynolds, and Rosemary Monahan. 2021. Using Dafny to Solve the VerifyThis 2021 Challenges. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21), July 13, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3464971.3468422

## 1 INTRODUCTION

This paper provides an experience report of using the Dafny program verifier, at the VerifyThis 2021 program verification competition. During the competition our team, comprised of one post-doc and one PhD student tackled two of the three challenges posed[1] using the Dafny program verifier [14].

The competition examined both the verification tools and the users of those tools. As a result, we were able to identify some strengths and weaknesses in Dafny, as well as harness the opportunity to educate ourselves on applying this formal verification tool to relatively complex algorithmic problems.

This paper is structured as follows. Section 2 briefly summarises the VerifyThis competition series and the Dafny program verifier. Section 3 describes and discusses two of the competition challenges and our corresponding Dafny solutions, constructed during the competition period. In Section 4 we discuss our experience and outline some of the strengths and weaknesses of our chosen tool for these challenges. Finally, Section 5 concludes.

*This work is partially funded by the Irish Research Council: GOIPG/2019/4529.
[1] Authors Farrell and Reynolds participated as the "Maynooth University" team.



*FTfJP '21, July 13, 2021, Virtual, Denmark*

ACM ISBN 978-1-4503-8543-5/21/07.
https://doi.org/10.1145/3464971.3468422

## 2 BACKGROUND

In this section, we briefly summarise the structure of the VerifyThis competition series and describe the verification tool that we used.

### 2.1 VerifyThis Competition

VerifyThis is a series of program verification competitions that has been held at FoVeOOS2011 [2], FM2012 [7], Dagstuhl Seminar 14171 (2014), ETAPS 2015–2019 [4, 8–11] and 2021, inspired by the VSTTE 2010 competition [12]. VerifyThis aims to evaluate the usability of logic-based program verification tools in an easily repeatable, controlled experiment, and to bring together and encourage discussion between those interested in formal verification. The competition challenges are described in a combination of natural language and pseudo code. Participants must formalise the requirements, implement a solution, and formally verify the implementation for adherence to the specification. There are no restrictions on the programming language or verification technology used.

The correctness properties posed in the problems focus on the input-output behaviour of programs. Solutions are judged for correctness, completeness, and elegance. Typically, VerifyThis is a 2-day event. The on-site competition on day 1 consists of three 90 minute challenges which are released consecutively. On day 2 the jury discusses the solutions with each team participating, in parallel all participants present and discuss their solutions and their different approaches. VerifyThis 2021 was unusual because it took place as an online-only event (due to COVID-19). The three challenges were issued concurrently, with a 24-hour period allocated for participants to attempt solutions. Challenges were intentionally made more difficult than those for the on-site competition because of the extended time period allowed. Day 2 then consisted of participants short online presentations of solutions. Twenty-three teams submitted solutions, three of which used Dafny as their main tool.

### 2.2 Dafny

Dafny is a programming language with a program verifier, used to statically verify functional program correctness, allowing users to define specification constructs e.g. pre-/post-conditions [14]. For verification, programs are translated into the Boogie intermediate verification language [1] and then the Z3 automated theorem prover discharges the associated proof obligations [3].

Figure 1 shows the structure of a method with specification constructs in Dafny. Pre- and post-conditions are indicated by the `requires` (line 2) and `ensures` (line 4) clauses respectively. The `modifies` clause (line 3) allows the user to specify a frame

```
1  method myMethod(x: int) returns (y: int)
2  requires ...
3  modifies ...
4  ensures ...
5  {
6   while (i<x) //this is a comment
7   invariant ...
8   decreases ...
9   {
10  ...
11  }
12 }
```

**Figure 1: The basic structure of a method with specification constructs in Dafny.**

condition that denotes the set of memory locations that are allowed to be modified. Loop invariants (line 7) are used to reason about the correctness of loops and provide support for verifying related post-conditions. The loop variant, identified as a `decreases` clause (line 8), is used to prove termination. Dafny supports specification-only `ghost` code, which is erased in the resulting executable and is only available for use by the verifier. Dafny has been used in previous VerifyThis competitions [8, 11] and also in larger systems [5, 6]. We used Dafny as part of Visual Studio Code [13][2].

## 3 THE CHALLENGES

VerifyThis 2021 consisted of three verification challenges. Of these three, we were able to partially complete two challenges during the allotted competition time. This section summarises and describes our efforts. Full source code is available in our repository[3].

### 3.1 Challenge 1: Lexicographic Permutations

"*This challenge considers an algorithm that takes any sequence A as input, and enumerates all possible permutations of A. Moreover, it enumerates these permutations in sorted (lexicographic) order.*"

The basic algorithm that was given to competitors is provided in Figure 2. The verification tasks are summarised as follows[4]:

**Task 1.1:** Verify that 'next' is memory-safe.
**Task 1.2:** Verify that 'next' terminates for every input.
**Task 1.3:** Verify that any changes on 'a' performed by 'next' are permutations.
**Task 1.4:** Verify that, if 'next(a)' returns false, then 'a' is left unmodified and is indeed the last permutation in the sequence.
**Task 1.5:** Verify that, if 'next(a)' returns true, then 'a' is modified to be the proper next permutation in the sequence.
**Task 1.6:** Verify that 'permut' is memory-safe.
**Task 1.7:** Verify that 'permut' terminates for every input.
**Task 1.8:** Verify that any permutation reported by 'permut' is unique.
**Task 1.9:** Verify that 'permut(a)' reports all permutations of 'a'.
**Task 1.10:** Verify that 'permut' outputs all permutations in lexicographic order.

We specified numerous predicates in Dafny to help with verifying properties about the alogorithm. These are summarised in Figure 3:

```
1 seq<int> permut(int[] A){
2     seq<int> result := seq();
3     if (A = null) return result;
4     sort(A);
5     do { result := result ++ seq(to_seq(A)); }
6     while (next(A));
7     return result;
8 }
9 bool next(int[] A) {
10    int i := A.length - 1;
11    while (i > 0 ∧ A[i - 1] ≥ A[i]) {
12        i := i - 1;
13    }
14    if (i ≤ 0) return false;
15    int j := A.length - 1;
16    while (A[j] ≤ A[i - 1]){
17        j := j - 1;
18    }
19    int temp := A[i - 1];
20    A[i - 1] := A[j];
21    A[j] := temp;
22    j := A.length - 1;
23    while (i < j) {
24        temp := A[i];
25        A[i] := A[j];
26        A[j] := temp;
27        i := i + 1;
28        j := j - 1;
29    }
30    return true;
31 }
```

**Figure 2: This pseudo code was given to the competitors as part of the Challenge 1 description. Note that '++' denotes sequence concatenation.**

```
1  predicate sorted(s: seq<int>)
2  {
3    ∀ i, j | 0 ≤ i < j < |s| · s[i] ≤ s[j]
4  }
5  predicate reverseSorted(s: seq<int>)
6  {
7    ∀ i, j | 0 ≤ i < j < |s| · s[i] ≥ s[j]
8  }
9  predicate sortedRange(s: seq<int>, l: int, u: int)
10 requires 0 ≤ l ≤ u ≤ |s|
11 {
12   sorted(s[l..u])
13 }
14 predicate reverseSortedRange(s:seq<int>, l:int, u:int)
15 requires 0 ≤ l ≤ u ≤ |s|
16 {
17   reverseSorted(s[l..u])
18 }
19 predicate lastPerm(s: seq<int>)
20 {
21   reverseSorted(s)
22 }
23 predicate ltseq(a: seq<int>, b: seq<int>)
24 requires |a| = |b|
25 {
26   ∃ j | 0 ≤ j < |a| · a[..j] = b[..j] ∧ a[j] < b[j]
27 }
28 predicate strictlySortedSeq(S:seq<seq<int>>)
29 requires ∀ i | 0 ≤ i < |S| - 1 · |S[i]| = |S[i + 1]|
30 {
31   |S|>1 ⇒ ∀ i | 0 ≤ i < |S|-1 · ltseq(S[i], S[i + 1])
32 }
```

**Figure 3: Predicates in Dafny for Challenge 1.**

[2]Dafny version 3.1.0.
[3]https://github.com/mariefarrell/MUVerifyThis2021.git
[4]https://www.pm.inf.ethz.ch/research/verifythis/Challenges.html

**Lines 1–18:** The `sorted` and `reverseSorted` predicates are used to describe what it means for the sequence to be sorted in ascending and descending order respectively. Based on these the `sortedRange` and `reverseSortedRange` predicates consider ascending and descending sortedness, respectively, given a range of the input sequence.

**Lines 19–22:** The `lastPerm` predicate allows us to identify when we have reached the final permutation which, since it is the last in the lexicographic order, will be `reverseSorted`. This is particularly useful when verifying Task 1.4.

**Lines 23–32:** The `ltseq` predicate takes two sequences of the same size as input and checks whether one is less than, in terms of the lexicographic ordering, than the other. This predicate is used by the `strictlySortedSeq` predicate which determines whether an input sequence of sequences is in lexicographical order. This is used to verify Task 1.10.

Figure 4 contains our Dafny implementation of the `next` and `permut` methods as outlined in Figure 2.

**Lines 1–5:** These pre- and post-conditions verify Tasks 1.3, 1.4 and 1.5. The `modifies` clause (line 2) relates to the frame condition and allows us to modify the input array of integers. This supports memory safety in Task 1.1 by specifying the memory locations (`a`) that are allowed to be modified. We verify that the modified array contains the same elements as the input array using Dafny's `multiset` construction on line 3, this corresponds to Task 1.3. The post-condition on line 4 corresponds to the verification of Task 1.4 which specifies that if `next` returns `false` then `a` is unmodified and is the last permutation in the sequence. In contrast, we partially verify Task 1.5 via the post-condition on line 5 which specifies that, if `next` returns `true` then `a` is modified to be the proper next permutation in the sequence. We *partially* verify this property because we do not give sufficient treatment to the *proper* next permutation. Instead, we focus on verifying that the permutation does come after the original but we could not verify that there are no other permutations in between.

**Lines 6–17:** This corresponds to the functionality on lines 10–13 of Figure 2. The addition of the loop invariants on lines 10–13 supports the verification of the post-conditions on lines 3–5. In particular, the invariant on lines 11–12 is necessary to prove the post-condition on line 4 corresponding to Task 1.4. The decreases clause on line 14 is provided to prove loop termination and contributes to the fulfilment of Task 1.2.

**Lines 18–29:** This corresponds to the functionality of lines 14–18 of Figure 2. The loop invariants on lines 24–25 ensure the correct functioning of the loop with the invariant on line 24 concerned with memory safety (Task 1.1). The `decreases` clause on line 26 serves to prove termination (Task 1.2).

**Lines 30–47:** These lines correspond to lines 19–31 of Figure 2. The ghost variable on line 32 is used in the loop invariant on lines 37–38 which supports the verification of the post-condition on line 5 corresponding (partially) to Task 1.5. Similarly, the invariant on line 39 supports the verification of the post-condition on line 3 (Task 1.3).

**Lines 48–54:** The `permut` method is specified by the pre-and post-conditions on lines 50–54. Memory safety (Task 1.6) is addressed by the `modifies` clause on line 50. The post-conditions on lines 51–53 ensure that all produced sequences are the same size and are in lexicographic order (Tasks 1.8 and 1.10), respectively. For Task 1.8, we get uniqueness from strict monotonicity, a strictly increasing sequence contains no duplicates.

**Lines 55–83:** These lines correspond to the functionality of lines 1–8 in Figure 2. We define a ghost variable (line 64) which is used by the invariant on lines 70–71. This is used to prove that the pre-condition for `ltseq` is not violated when it is called in the invariant on lines 74–75. This invariant on lines 74–75 is also needed to verify the invariant on line 76 which, in turn, supports the verification of the post-condition on line 53 (Tasks 1.8 and 1.10). The `decreases` clause on line 77 serves to prove termination (Task 1.7).

Overall, we were able to verify Tasks 1.1, 1.2, 1.3, 1.4, 1.6, 1.7, 1.8 and 1.10. We partially verified Task 1.5 but we did not provide a particularly elegant (or complete) solution to Task 1.9 because we did not have a nice way of specifying that *all* permutations were reported. This could be a limitation of us as Dafny users or it could be due to the way that we constructed the algorithm.

For Task 1.9, our approach was to use the fact that, for a sequence of size $n$, there will be at most $n!$ permutations. We thus included the loop variable `fac` on line 67 of Figure 4 and our loop condition enforces that `next` is not called more than `fac` times. Without this variable we could not prove that the loop would terminate or that `next` was only called the correct number of times.

With respect to Task 1.5, we were not able to completely verify that the *proper* next permutation is returned. Specifically, the post-condition that we were trying to verify was that there are no such sequences in between the one that has been returned and the previous one. This kind of property was difficult to verify in Dafny.

## 3.2 Challenge 2: DLL to BST

"*This challenge is to verify an in-place algorithm to convert a sorted doubly-linked list (DLL) into a balanced binary search tree (BST).*"

Figure 5 contains the corresponding pseudo code that was given to competitors. The verification tasks were as follows:

**Task 2.1:** Prove that this algorithm converts an input list into a tree.

**Task 2.2:** Prove that the algorithm is memory-safe.

**Task 2.3:** Prove that if the input list is sorted then the resulting tree is a BST.

**Task 2.4:** Prove that the resulting BST is balanced.

**Task 2.5:** Prove that the algorithm terminates.

**Task 2.6:** (Optional) Prove the above for an iterative version of size.

We were less successful with this challenge than we were with the previous one because we spent a lot of time encoding predicates to determine the validity of a binary search tree and doubly linked list. These predicates and a necessary constructor are contained in

```
method next(a: array<int>) returns (ok: bool)
modifies a
ensures multiset(a[..]) = old(multiset(a[..]))
ensures !ok  ⇒  a[..] = old(a[..]) ∧ lastPerm(a[..])
ensures  ok  ⇒  ltseq(old(a[..]), a[..])
{
  var len := a.Length;
  var i := len - 1;
  while i > 0 ∧ a[i - 1] ≥ a[i]
  invariant -1 ≤ i < len ∧ i = -1  ⇒  len = 0
  invariant
    i ≥ 0  ⇒  reverseSortedRange(a[..], i, a.Length)
  invariant a[..] = old(a[..])
  decreases i
  {
    i := i - 1;
  }
  if i ≤ 0 {
    ok := false;
    return;
  }
  var j := len - 1;
  while a[j] ≤ a[i - 1]
  invariant a[..] = old(a[..])
  invariant j > i - 1
  decreases j
  {
    j := j - 1;
  }
  a[i - 1], a[j] := a[j], a[i - 1];

  ghost var idx := min(i - 1, j);
  var k := i - 1;
  j := len - 1;
  while i < j
  invariant k < i < len ∧ k ≤ j < len
  invariant a[..idx] = old(a[..idx])
            ∧ a[idx] > old(a[idx])
  invariant old(multiset(a[..])) = multiset(a[..])
  decreases j - i
  {
    a[i], a[j] := a[j], a[i];
    i := i + 1;
    j := j - 1;
  }
  ok := true;
}
method permut(a: array<int>)
 returns (result: seq<seq<int>>)
modifies a
ensures ∀ i | 0 ≤ i < |result| - 1
          · |result[i]| = |result[i + 1]|
ensures strictlySortedSeq(result)
{
  result := [];

  if a.Length ≤ 0 {
    return [[]];
  }
  sort(a);

  result := result + [a[..]];

  ghost var len := |a[..]|;
  var ok := next(a);
  var fac := factorial(a.Length);

  while ok ∧ fac > 0
  invariant |result| ≥ 1
  invariant ∀ i | 0 ≤ i < |result|
            · |result[i]| = len
  invariant ∀ i | 0 ≤ i < |result| - 1
            · |result[i]| = |result[i + 1]|
  invariant ok  ⇒  ∀ i | 0 ≤ i < |result|
            · ltseq(result[i], a[..])
  invariant strictlySortedSeq(result)
  decreases fac
  {
    result := result + [a[..]];
    ok := next(a);
    fac := fac - 1;
  }
}
```

**Figure 4: Dafny implementation of `next` (lines 1–47) and `permut` (lines 48–83).**

```
// Ref is the type of nodes used for both list
// and tree, and has these fields:
field data: Int
field prev: Ref
field next: Ref
method size(head: Ref) returns(count: Int) {
    if (head != null) {
      count := size(head.next)
      count := count + 1
    } else {
      count := 0
    }
}
method dll_to_bst(head: Ref) returns(root: Ref) {
    var n: Int
    var right: Ref
    n := size(head)
    root, right := dll_to_bst_rec(head, n)
}
// Converts a sorted DLL into a balanced BST
method dll_to_bst_rec(head: Ref, n: Int)
  returns(root: Ref, right: Ref) {
    if (n > 0) {
      // Recursively construct the left subtree
      var left: Ref
      left, root := dll_to_bst_rec(head, n/2)

      // Set pointer to left subtree
      root.prev := left

      // Recursively construct the right subtree
      var temp: Ref
      temp, right
        := dll_to_bst_rec(root.next, n-n/2-1)

      // Set pointer to right subtree
      root.next := temp
    } else {
      root := null
      right := head
    }
}
```

**Figure 5: This pseudo code was given to the competitors as part of the Challenge 2 description.**

Figure 6 which we describe below. We broadly followed a specification of binary trees that was available in the Dafny repository[5].

**Lines 1–5:** This class describes the basic components of a variable of type `Ref`. This corresponds to the `field` definitions on lines 1–5 of Figure 5 that was given to competitors.

**Lines 6–8:** We define two ghost variables to be used later. It is common when creating classes in Dafny to include both non-ghost and ghost variables. Here, the ghost variable `Contents` provides an abstract representation of the linked list for specification purposes. The set `Repr` contains all objects making up the *representation set* of a `Ref`, including itself and its subobjects. This is mainly used to describe frame conditions (the set of objects in `reads` and `modifies` clauses). For example, requiring that the sets `prev.Repr` and `next.Repr` are disjoint, using the '`!!`' operator, ensures that no object in the left subtree also appears in the right subtree, and *vice versa*. This also ensures that there are no cycles. We furthermore require that the left and right subtrees are 'subsets' of their parent, using the '`<=`' operator. We also prove termination by showing that

[5] https://git.io/JG2i7

```
1  class Ref {
2    var data: int
3    var next: Ref?
4    var prev: Ref?
5
6    ghost var Repr: set<object>
7    ghost var Contents: seq<int>
8
9    constructor(data: int)
10   ensures ValidLL() ∧ ValidBST() ∧ SortedLL()
11   {
12     this.data := data;
13     next := null;
14     prev := null;
15     Repr := {this};
16     Contents := [data];
17   }
18   predicate ValidLL()
19   reads this, Repr
20   decreases Repr
21   {
22     this in Repr ∧ prev = null
23     ∧ (next ≠ null  ⇒  next in Repr
24         ∧ next.Repr ≤ Repr ∧ this ∉ next.Repr
25         ∧ Contents = [data] + next.Contents
26         ∧ next.ValidLL())
27     ∧ (next = null  ⇒  Contents = [data])
28   }
29   predicate SortedLL()
30   reads this, Repr
31   requires ValidLL()
32   decreases Repr
33   {
34     if next = null then true else
35     if data ≤ next.data then next.SortedLL()
36       else false
37   }
38   predicate ValidBST()
39   reads this, Repr
40   decreases Repr
41   {
42     this in Repr
43     ∧ (prev ≠ null  ⇒  prev in Repr
44         ∧ prev.Repr ≤ Repr ∧ this ∉ prev.Repr
45         ∧ prev.ValidBST()
46         ∧ ∀ x | x in prev.Contents · x < data)
47     ∧ (next ≠ null  ⇒  next in Repr
48         ∧ next.Repr ≤ Repr ∧ this ∉ next.Repr
49         ∧ next.ValidBST()
50         ∧ ∀ x | x in next.Contents · x > data)
51     ∧ (prev = null ∧ next = null
52         ⇒ Contents = [data])
53     ∧ (prev ≠ null ∧ next = null
54         ⇒ Contents = prev.Contents + [data])
55     ∧ (prev = null ∧ next ≠ null
56         ⇒ Contents = [data] + next.Contents)
57     ∧ (prev ≠ null ∧ next ≠ null
58         ⇒ prev.Repr !! next.Repr
59         ∧ Contents = prev.Contents
60                      + [data] + next.Contents)
61   }
62   predicate BalancedBST()
63     reads this, Repr
64     decreases Repr
65     requires ValidBST()
66   {
67     ValidBSTisValidRef();
68     (depth(next) = depth(prev) ∨
69       depth(next) = depth(prev) + 1 ∨
70       depth(next) + 1 = depth(prev)) ∧
71     (next ≠ null  ⇒  next.BalancedBST()) ∧
72     (prev ≠ null  ⇒  prev.BalancedBST())
73   }
74 }
75 function getRepr(ref: Ref?): set<object>
76 reads ref
77 {
78   if ref ≠ null then ref.Repr else {}
79 }
```

**Figure 6: Predicates and `Ref` class in Dafny for Challenge 2.**

```
1  method size(ref: Ref?) returns (count: nat)
2  requires ref = null ∨ ref.ValidLL()
3  ensures ref = null  ⇒  count = 0
4  ensures ref ≠ null  ⇒  count = |ref.Contents|
5  decreases if ref ≠ null then ref.Repr else {}
6  {
7    if ref ≠ null {
8      count := size(ref.next);
9      count := count + 1;
10   } else {
11     count := 0;
12   }
13 }
14 method dll2bst(head: Ref?) returns (root: Ref?)
15 modifies if head ≠ null then head.Repr else {}
16 requires head = null
17          ∨ (head.ValidLL() ∧ head.SortedLL())
18 ensures root = null ∨ root.ValidBST()
19 {
20   var n := size(head);
21   var right: Ref?;
22   root, right := dll2bstrec(head, n);
23 }
24 method dll2bstrec(head: Ref?, n: nat)
25   returns (root: Ref?, right: Ref?)
26 modifies if head ≠ null then head.Repr else {}
27 requires head = null
28          ∨ (head.ValidLL() ∧ head.SortedLL())
29 ensures root = null ∨ root.ValidBST()
30 ensures right = null ∨ right.ValidLL()
31 ensures head ≠ null  ⇒
32             (n = |head.Contents|  ⇒  right = null) ∧
33             (n < |head.Contents|  ⇒  right ≠ null)
34 ensures getRepr(head) = getRepr(root) + getRepr(right)
35 ensures getRepr(root) !! getRepr(right)
36 decreases n
37 {
38   if n > 0 {
39     var left: Ref?;
40     left, root := dll2bstrec(head, n/2);
41     root.prev := left;
42
43     var temp: Ref?;
44     temp, right := dll2bstrec(root.next, n - n/2 - 1);
45     root.next := temp;
46   } else {
47     root := null;
48     right := head;
49   }
50 }
```

**Figure 7: Dafny implementation for Challenge 2.**

the size of `Repr` decreases with each recursive call (lines 20, 32, 40 and 64 of Figure 6).

**Lines 9–17:** Here we provide a constructor for this class.

**Lines 18–28:** This predicate specifies what it means to be a valid linked list. The `reads` clause on line 19 specifies the memory locations that the predicate is allowed to access. We use the `decreases` clause on line 20 to prove termination in the usual way. Note that we do not actually require that the linked list is a true doubly-linked list, merely that it is a valid singly-linked list, and that the reference `prev` exists.

**Lines 29–37:** This predicate specifies what it means to be a sorted linked list.

**Lines 38–61:** Here, we specify what it means for a binary search tree to be valid. In particular, the `prev` and `next` subtrees should also be valid binary search trees (lines 45 and 49 respectively). The `prev` and `next` subtrees should also be disjoint indicated by the '`!!`' operator on line 58.

**Lines 62–74:** The `ValidRef` predicate is the same as the `ValidBST` predicate, but omits the requirement for left

subtrees to contain smaller data, and right subtrees to contain larger data. It is our least specific requirement, but we still enforce that there are no cycles, since termination would otherwise be very difficult to determine in general.

**Lines 75–79:** This function is used to collect the set of representative objects for a given `Ref`.

The predicates defined in Figure 6 are necessary to verify Tasks 2.3 and 2.4. Our Dafny implementation of the functionality that was provided to competitors (shown in Figure 5) is contained in Figure 7 and we describe it as follows:

**Lines 1–13:** This is our Dafny implementation of the `size` function that was given on lines 6–13 of Figure 7. The precondition on line 2 requires that the input `ref` is either null or it is a valid linked list. We then ensure (line 3) that if `ref` is null then `count` will be 0. Otherwise, it returns the size of `ref` (line 4). The `decreases` clause on line 5 is used to prove termination, which supports Task 2.5. In the case that `ref` is not null, the set of objects making up the representation of `ref` decreases in size, however if `ref` *is* null, then there is no such set, so we simply say that the empty set decreases.

**Lines 14–23:** This corresponds to lines 14–19 of the pseudo code that was given in Figure 5. We added the specifications on lines 15–18 for verification. The `modifies` clause is needed here and supports memory safety as prescribed by Task 2.2. We ensure (line 18) that the output is a valid binary search tree using the predicate that was previously defined. We note that these specifications were verified for this method, however, they depend on those in the subsequent method which we could not completely verify.

**Lines 24–50:** Here, we capture the main functionality of the algorithm, corresponding to lines 20–40 of Figure 5. We were able to verify termination of this method (line 35 and Task 2.5). We verified memory safety via the `modifies` clause on line 26. We verified the post-conditions on lines 31–34 which offers some evidence towards Task 2.4. However, we encountered difficulties when trying to verify the specifications on lines 29–30. Much of the difficulty revolves around verifying that we have permission to stitch `left` and `temp` onto `root`, and deciding what exactly we can say about the resulting data structure.

We were able to completely verify Task 2.5 but not the other tasks. As a result, we did not attempt the optional Task 2.6.

A significant amount of time was spent trying to formalise a predicate describing a valid doubly-linked list. Termination of such a predicate follows from the fact that `Repr`, the set of objects making up the representation of a `Ref` is finite, and that it decreases in size with each call. This is not so simple with a doubly-linked list, since, as presented, there is no easy way to prove that such a predicate terminates. This stems from the difficulty in navigating a doubly linked list and knowing your exact location in the list at any given time. We might have had a better result if we had used another data structure to represent the list, for example a pair of stacks, and defined an invariant to describe the relationship between this and the doubly linked list, giving us more visibility of how we traverse the structure and providing more explicit termination measures. However, this may not have solved all of our problems.

Further, the use of potentially null variables and the recursive nature of the method on lines 24–50 of Figure 7 caused us problems with respect to proving the frame condition (`modifies` clause). We specify that `head.Repr` may be modified (line 15), which means that Dafny can modify any object *in* `head.Repr`. However, Dafny does not recognise that it is allowed to modify `root` (on line 41), despite the fact that Dafny knows that `root` is not null, that `root` is in `head.Repr`, and that it can modify any object in `head.Repr`, from which it should follow immediately that `root` is modifiable. In the discussion session at VerifyThis 2021 other teams presented similar issues and some managed to solve this with their verification tools. We also noted that at least one of the other teams participating in the competition using Dafny switched to using Isabelle for Challenge 2.

## 4 DISCUSSION

The challenges in VerifyThis 2021 exposed a number of strengths and limitations of Dafny. Particular strengths included Dafny's ability to prove termination and memory safety with little input, especially in Challenge 1 (Tasks 1.1, 1.2. 1.6, 1.7 and 2.5). Dafny's built-in value types, i.e. sets, sequences, multisets, and maps, support ergonomic specification and verification. The `multiset` datatype was particularly useful for verification of Challenge 1, allowing us to state and easily prove that two arrays contained the same elements without needing to write additional functions or predicates to verify Task 1.3. Likewise, we used Dafny's immutable sequence types `seq` when specifying both challenges, and the `set` type in Challenge 2 for tracking the representation of an object.

We used two main constructs to support the specification of aggregate objects in Challenge 2. These are object invariants, which we express as a validity predicate e.g. `ValidLL` (lines 18–28 of Figure 6), and a representation set, expressed as a ghost variable `Repr` (line 15 of Figure 6) representing the object's dynamic frame[6]. This representation of an object's dynamic frame is provided as a set of objects, so `Repr` can contain references to objects of any type and that set of objects can dynamically change over time. We can specify permissions on this set, using `reads` and `modifies` clauses, specified for each predicate, function or method provided.

Dafny's predicates and lemmas helped us with verification tasks throughout the challenges, allowing us to make our specifications more concise. Dafny provides automatic induction proofs for lemmas which we found useful in our proofs. We also took advantage of Dafny's specification-only `ghost` constructs in both Challenge 1, for verification of loop invariants, and Challenge 2, for the abstract representation of the linked list and set of representation objects as mentioned above.

It was difficult to verify properties about possibly null objects in Challenge 2 e.g. lines 24–50 of Figure 7. A larger variety of built-in datatypes would potentially have been useful here. The `decreases` annotation provides an expression that decreases with every recursive call and is bounded below. If a recursive function or method is not given an explicit `decreases` clause, then

[6]The name of the programming language Dafny comes from a permutation of the letters in "dynamic frames".

the Dafny verifier guesses one based on the parameter provided by the function or method. This was useful apart from when more sophisticated measures were required, e.g. line 5 of Figure 5. Further to this, the error messages provided by Dafny were often not informative, but this is a limitation of many verification tools.

A nice aspect of Dafny is that memory safety is verified without a large specification overhead. For Challenge 1, array bounds are verified via the associated variants and invariants. The use of `modifies` clauses allow us to specify which memory locations are allowed to be modified by the relevant methods. If no `modifies` clause is given then the default is that no memory locations can be modified. These `modifies` clauses are accompanied by `reads` clauses in Challenge 2 to control which memory locations are accessible (e.g. the current object and its representation on lines 19, 30, 39 and 63 of Figure 6).

The competition provided both a challenge to the tool and to the users of the tool. It was beneficial for us, the users, because it allowed us to learn more about Dafny as well as examine how Dafny performs when verifying relatively complex algorithmic problems. To date we have not seen details of the other Dafny solutions submitted to VerifyThis 2021 but we look forward to comparing our efforts when they are made available. Likewise, a deeper discussion with other teams, regarding how they used their tool to solve the challenges would be interesting. We hope to resolve the problems that we encountered and we thank K. Rustan. M. Leino for preliminary discussions regarding potential solutions.

## 5 CONCLUSIONS

Participation in the VerifyThis 2021 competition was educational because it allowed us to examine the usability and usefulness of Dafny for algorithmic verification tasks. It also gave us an opportunity to learn more about the tool and gave us a better understanding of its strengths and weaknesses. We would like to offer congratulations to the prize winners of VerifyThis 2021. VerifyThis 2021 was sponsored by Amazon (AWS) and the winners are listed at: https://www.pm.inf.ethz.ch/research/verifythis/Prizes.html.

## REFERENCES


[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (LNCS, Vol. 4111)*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[2] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen, and Mattias Ulbrich. 2012. The COST IC0701 Verification Competition 2011. In *Formal Verification of Object-Oriented Software (LNCS, Vol. 7421)*. Springer, 3–21. https://doi.org/10.1007/978-3-642-31762-0_2

[3] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[4] Claire Dross, Carlo A Furia, Marieke Huisman, Rosemary Monahan, and Peter Müller. 2021. VerifyThis 2019: a program verification competition. *International Journal on Software Tools for Technology Transfer* (2021), 1–11. https://doi.org/10.1007/s10009-021-00619-x

[5] Marie Farrell, Matthew Bradbury, Michael Fisher, Louise A Dennis, Clare Dixon, Hu Yuan, and Carsten Maple. 2019. Using Threat Tnalysis Techniques to Guide Formal Verification: A Case Study of Cooperative Awareness Messages. In *International Conference on Software Engineering and Formal Methods (LNCS, Vol. 11724)*. Springer, 471–490. https://doi.org/10.1007/978-3-030-30446-1_25

[6] Marie Farrell, Nikos Mavrakis, Clare Dixon, and Yang Gao. 2020. Formal Verification of an Autonomous Grasping Algorithm. In *International Symposium on Artificial Intelligence, Robotics and Automation in Space*. ESA.

[7] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. 2015. VerifyThis 2012: A Program Verification Competition. *International Journal on Software Tools for Technology Transfer* 17 (2015), 647–657. https://doi.org/10.1007/s10009-015-0396-8

[8] Marieke Huisman, Vladimir Klebanov, Rosemary Monahan, and Michael Tautschnig. 2017. VerifyThis 2015. *International Journal on Software Tools for Technology Transfer* 19, 6 (2017), 763–771. https://doi.org/10.1007/s10009-016-0438-x

[9] Marieke Huisman, Rosemary Monahan, Peter Müller, Wojciech Mostowski, and Mattias Ulbrich. 2017. *VerifyThis 2017: A Program Verification Competition.* Number 2017-10 in Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology.

[10] Marieke Huisman, Rosemary Monahan, Peter Müller, Andrei Paskevich, and Gidon Ernst. 2019. *VerifyThis 2018: A Program Verification Competition.* Research Report. Université Paris-Saclay.

[11] Marieke Huisman, Rosemary Monahan, Peter Muller, and Erik Poll. 2016. VerifyThis 2016: A program verification competition. (2016).

[12] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. 2011. The 1st Verified Software Competition: Experience Report. In *International Symposium on Formal Methods (LNCS, Vol. 6664)*. Springer, 154–168. https://doi.org/10.1007/978-3-642-21437-0_14

[13] Rafael Krucker and Markus Schaden. 2017. *Visual Studio Code Integration for the Dafny Language and Program Verifier.* Ph.D. Dissertation. HSR Hochschule für Technik Rapperswil.

[14] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LNCS, Vol. 6355)*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20