Automated Verification of Nested DFS

Jaco C. van de Pol(✉)

Formal Methods and Tools, Department of Computer Science,
CTIT, University of Twente, Enschede, The Netherlands
j.c.vandepol@utwente.nl

Abstract. In this paper we demonstrate the automated verification of the Nested Depth-First Search (NDFS) algorithm for detecting accepting cycles. The starting point is a recursive formulation of the NDFS algorithm. We use Dafny to annotate the algorithm with invariants and a global specification. The global specification requires that NDFS indeed solves the accepting cycle problem. The invariants are proved automatically by the SMT solver Z3 underlying Dafny. The global specifications, however, need some inductive reasoning on paths in a graph. To prove these properties, some auxiliary lemmas had to be provided. The full specification is contained in this paper. It fits on 4 pages, is verified by Dafny in about 2 minutes, and was developed in a couple of weeks.

1 Introduction

Model checking is an attractive verification technique because it is fully automatic. Since model checking is memory and time intensive, scalability of model checking to industrial systems requires sophisticated algorithms and high-performance implementations. This makes the construction of model checkers intricate and error prone. When model checkers are used for the verification of industrial critical systems, they themselves become part of the critical engineering infrastructure. This motivated several efforts to verify the verification algorithms and tools themselves.

Recently, model checkers have been verified using *interactive theorem provers*. Here users are responsible for creating a proof, which is then checked by the machine. Examples include the verification of a μ-calculus model checker in Coq [14], compositional model checkers in ACL2 [12], and a depth-first search algorithm for strongly connected components in Coq [11]. Probably the largest piece of work in this direction is the development of a reasonably efficient, certified automata-based LTL model checker in Isabelle/HOL [4]. This includes the translation of LTL properties to Büchi automata, and an algorithm to detect accepting cycles in the result graph.

The purpose of the current paper is to raise the level of automation. We investigated whether full functional correctness of graph-based verification algorithms can be established by *automatic program verifiers*. These tools depend on user added annotations to a program, like pre- and postconditions and loop invariants. The program verifier then generates proof obligations, which are discharged automatically by an SMT solver.

M. Núñez and M. Güdemann (Eds.): FMICS 2015, LNCS 9128, pp. 181–197, 2015.
DOI: 10.1007/978-3-319-19458-5_12

Concretely, this paper demonstrates how the Nested Depth-First Search algorithm (NDFS) can be expressed in Dafny, and how it can be verified in an incremental manner. The complete specification (Section A) demonstrates that NDFS correctly decides if the input graph contains an accepting cycle. Dafny is an automatic program verifier created by Rustan Leino and relies on the workhorse Z3 as the underlying SMT solver. We took inspiration from the verification of the Schorr-Waite graph algorithm, also by Leino [10]. However, we insist on the natural recursive formulation of NDFS.

As far as we know, we provide the first verification of *full correctness* of a model checking algorithm by an *automatic program verifier*. A related approach [6] applied automatic program verifiers to distributed state space generators (but not on the model checking algorithm). Another approach based on annotations is the PAT model checker, model checking its own annotations [15] (but not full functional correctness).

2 Nested Depth-First Search and Dafny

2.1 Dafny

Dafny [10] provides a straightforward imperative programming language. It supports sequential programs, with classes and dynamic allocation. The program can be mixed freely with specification annotations, like preconditions (`requires`), postconditions (`ensures`) and invariants. Loops and recursion require termination metrics (`decreases`) to ensure termination. In order to support modularity, framing conditions restrict read and write permissions on objects.

The specification language is quite liberal: specifications can introduce ghost variables in program text, mathematical functions, and built-in value types like sets and sequences. We heavily depend on these features.

Dafny parses and type-checks the program, and generates proof obligations to guarantee absence of runtime errors, termination, and the validity of all specification annotations. Dafny works in a modular fashion, method by method. It relies on the SMT solver Z3 [3] to discharge the proof obligations, and reconstructs sensible error messages at the program level when verification fails.

2.2 Nested Depth-First Search

The automata based approach [16] reduces the LTL model checking problem to the detection of *accepting cycles*. Given a graph $G = (V, E, s_0, A)$, with nodes V, edges E, root $s_0 \in V$ and accepting states $A \subseteq V$, the question is whether there exists a reachable accepting cycle, i.e. a state $t \in A$ with $s_0 E^* t$ and $t E^+ t$. The famous linear-time algorithm to detect accepting cycles on-the-fly is called Nested Depth-First Search [2]. NDFS performs a first (blue) DFS to detect accepting states, and a second (red) search to identify cycles on those states. Both searches visit nodes at most once, by colouring them cyan/blue and pink/red.

NDFS is heavily used as the core algorithm of LTL model checkers, starting with SPIN model checker [7], and also forms the basis of parallel LTL model checking in LTSmin [8]. Its memory overhead is negligible: only two bits per state [13]. The version verified in this paper is the new NDFS [13] without early cycle detection, and with a distinction in pink and red nodes. It corresponds to the sequential version of the parallel algorithm in [8]. We claim that the pink colour not only helped in parallelizing NDFS, but is also instrumental in the formal verification proof.

2.3 Formulation of the NDFS Algorithm in Dafny

```
1  datatype Color = white | cyan | blue | pink | red;
2
3  class Node {
4    var next: seq<Node>;
5    var accepting: bool;
6    var color1: Color;
7    var color2: Color;
8  }
9
10 method ndfs(root:Node) returns (found:bool)
11 { found := dfsblue(root); }
12
13 method dfsblue(s:Node) returns (found:bool)
14 { s.color1 := cyan;
15   var i := 0;
16   while (i < |s.next|)
17   { var t := s.next[i];
18     i := i+1;
19     if (t.color1 = white)
20     { found := dfsblue(t);
21       if (found) { return; }
22     }
23   }
24   if (s.accepting)
25   { found := dfsred(s);
26     if (found) { return; }
27   }
28   s.color1 := blue;
29   return false;
30 }
31
32 method dfsred(s:Node) returns (found:bool)
33 { s.color2 := pink;
34   var i := 0;
35   while (i < |s.next|)
36   { var t := s.next[i];
37     i := i+1;
38     if (t.color1 = cyan) { return true; }
39     if (t.color2 = white)
40     { found := dfsred(t);
41       if (found) { return; }
42     }
43   }
44   s.color2 := red;
45   return false;
46 }
```

Fig. 1. Expressing the plain NDFS algorithm in DAFNY syntax

Figure 1 introduces the recursive formulation of the NDFS algorithm in Dafny syntax. After introducing the enumerated `datatype Color` (ℓ. 1), the `class Node` of nodes in the underlying graph is defined (ℓ. 3-8). Each node is equipped with a sequence `next` of successors in the graph and a Boolean `accepting`. These attributes will never be changed. Two colours are introduced as well, which will be manipulated by the algorithm. Alternatively, one could introduce distinct types for bluish and reddish colours.

The main algorithm is `method ndfs` (ℓ. 10,11). Its single argument is the `root:Node` where the algorithm starts, and its return value `found:Bool` indicates whether an accepting cycle was found. Return values are named in Dafny, so they can be referred to in the postcondition of the specification. They can be used as normal local variables in the method body. The main method just calls `method dfsblue`. The reason to have `ndfs` as a separate method is to be able to attach the top-level specification to it later.

The recursive formulation of method `dfsblue (s:Node)` (ℓ. 13-30) closely follows the textbook description, see for instance [13, Fig 3.]. After marking s cyan (ℓ. 14), all successor nodes t of s are iterated over (ℓ. 15-18). If t is seen for the first time (ℓ. 19), it is processed recursively (ℓ. 20) and the result is stored in `found`. As soon as an accepting cycle has been found, the search can be terminated (ℓ. 21); note that `return` is an abbreviation for `return found` in Dafny, since we named the return value `found`.

After processing all successors of s, the red search is started with `dfsred(s)` (ℓ. 25), provided that s is accepting (ℓ. 24). Again, if an accepting cycle is found we return immediately. When no cycle is found, node s is coloured blue and the procedure returns (ℓ. 28-29).

The `method dfsred(s:Node)` (ℓ. 32-46) performs the red search in a similar fashion. Initially, nodes are coloured pink (ℓ. 33). All successors t are processed sequentially (ℓ. 34-37). If t is cyan, a cycle has been found and is reported (ℓ. 38). Otherwise, the procedure continues recursively and the results are propagated (ℓ. 39-41). Finally, when no cycle has been found at all, node s is coloured red and the procedure returns (ℓ. 44-45).

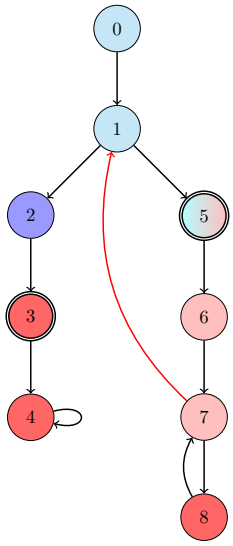

Fig. 2. Illustrating NDFS

Figure 2 illustrates the colours. Cyan and pink nodes are still in progress. After backtracking from the search, nodes are coloured blue or red. So for these colours we can establish strong invariants.

3 Developing the Correctness Proof

The verification was carried out incrementally. First, runtime errors are eliminated by appropriate preconditions, then termination is addressed. To verify the algorithm, the key approach was to identify invariants on the *local* properties of the colours in the graph. These invariants can be checked easily. Similar invariants played a crucial role in the manual proof of parallel NDFS [8]. Finally, completeness and soundness of NDFS are proved using auxiliary lemmas, which reason on global properties of paths and cycles in the graph.

3.1 Absence of Runtime Errors

Even though we did not specify any requirements on NDFS, the code in Figure 1 is not regarded correct by Dafny. It does not report syntax or type check errors, but the verifier complains (ℓ. 14, 33):

```
Error: assignment may update an object not in ... modifies clause
Error: target object may be null
```

First, in order to allow for modular verification, Dafny uses dynamic frames, insisting on explicit permissions to modify objects. In Figure 3, we added the permissions to modify the `color`-fields only (ℓ. 10, 16, 22). Note that `dfsred` only modifies `color2`.

In order to guarantee absence of runtime errors, Dafny has generated some implicit proof obligations. In our case, runtime errors could occur due to *null dereferences* (e.g. in `s.color1`) and *out-of-bound indexing* (e.g. in `next[i]`). The latter is excluded, since Dafny easily deduces $0 \leq i < |\texttt{s.next}|$ from the loop bounds. However, if initially `s:Node = null`, indeed `s.next` would lead to a run-time error.

In order to solve both problems, we use the technique explained in [10] in the verification of the Schorr-Waite graph algorithm. We extend the specification as indicated in Figure 3. We define a ghost variable `G:set<Node>` (ℓ. 1), indicating the universe of all (reachable) nodes in the graph. As a ghost variable, G can only be used in specification annotations; it cannot modify the program execution. Next, the `predicate graph(G)` is defined (ℓ. 3-5). G is a valid graph if its nodes are non-null records and their successors are in G again. We equip all methods with a precondition that requires that the start node is contained in the valid graph G (e.g., ℓ. 14-15). Since `graph(G)` is closed, there is no risk to run into null nodes anymore.

3.2 Termination

Still, Dafny is not satisfied. In order to guarantee total correctness, it insists on termination. Termination of the while loops in our case (cf. Figure 1, ℓ. 16, 35) is easily discharged automatically. However, the recursive calls (ℓ. 20, 40) lead to the following complaint:

```
Error: cannot prove termination; try supplying a decreases clause
```

```
1  ghost var G: set<Node>;
2
3  predicate graph(G:set<Node>)
4    reads G;
5  { ∀ m • m ∈ G ⟹ (m ≠ null ∧ ∀ n • n ∈ m.next ⟹ n ∈ G) }
6
7  method ndfs(root:Node) returns (found:bool)
8    requires graph(G);
9    requires root ∈ G;
10   modifies G`color1, G`color2;
11 {...}
12
13 method dfsblue(s:Node) returns (found:bool)
14   requires s ∈ G;
15   requires graph(G);
16   modifies G`color1, G`color2;
17 {...}
18
19 method dfsred(s:Node) returns (found:bool)
20   requires s ∈ G;
21   requires graph(G);
22   modifies G`color2;
23 {...}
```

Fig. 3. Specifying a well-defined and closed graph

So why does NDFS terminate at all? Basically, because every node is visited at most twice: once during `dfsblue` and once during `dfsred`. This is realized by the colours: we only recurse on `white` nodes, and immediately colour them `cyan`. We specify this insight by declaring that the function `G-Cyan(G)` decreases for each call to `dfsblue` (ℓ. 11 in Figure 4), where the set `Cyan(G)` is defined as those nodes $n \in G$ with `n.color1=cyan` (ℓ. 1-3)[1]. We add similar definitions and annotations for pink nodes in `dfsred`.

Dafny is not yet convinced: We clearly need to require that initially all nodes are `white` (ℓ. 6) and we only meet `white` nodes along the way (ℓ. 10), otherwise the termination function wouldn't decrease. Moreover, recursive calls to `dfsblue` could manipulate the `Cyan` set arbitrarily in principle, leading to non-termination for calls to subsequent successors. To exclude this, `dfsblue` must ensure that it will leave the `Cyan` set unchanged (ℓ. 12). Note that this is realized in (ℓ. 23), but only in case no accepting cycle is found. An invariant (ℓ. 15) is required to reason about the value of `Cyan` during and after the loop.

We are nearly there, but not quite! The preconditions lead to new proof obligations. Obviously, the recursive call to `dfsblue(t)` (ℓ. 18) satisfies the precondition that `t.color1=white`. However, Dafny points out that at (Figure 4, ℓ. 21) there is a call to `dfsred`, but the precondition `t.color2=white` at (Figure 4, ℓ. 28) is not guaranteed:

```
Error: A precondition for this call might not hold.
Related location: This is the precondition that might not hold.
```

[1] An alternative is to introduce and manipulate a ghost variable `Cyan` in the method body, but we prefer our more declarative approach, since it does not clutter the code.

```
1  function Cyan(G:set⟨Node⟩): set⟨Node⟩
2    reads G; requires graph(G);
3  { set n | n ∈ G ∧ n.color1 = cyan • n }
4
5  method ndfs(root:Node) returns (found:bool)
6    requires ∀ s • s ∈ G ⟹ s.color1 = s.color2 = white;
7  {...}
8
9  method dfsblue(s:Node) returns (found:bool)
10   requires s.color1 = white;
11   decreases G − Cyan(G);
12   ensures ¬found ⟹ old(Cyan(G)) = Cyan(G);
13 {...
14   while (i < |s.next|)
15   invariant Cyan(G) = old(Cyan(G)) ∪ {s};
16   {...
17     if (t.color1 = white)
18     { found := dfsblue(t);
19     ...
20   if (s.accepting)
21     { found := dfsred(s); // still to prove: why is s.color2 white?
22     ...
23   s.color1 := blue;
24   return false;
25 }
26
27 method dfsred(s:Node) returns (found:bool)
28   requires s.color2 = white;
29   decreases G − Pink(G);
30   ensures ¬found ⟹ old(Pink(G)) = Pink(G);
31 {...}
```

Fig. 4. Specifying decreasing termination functions

Indeed, the insight that the red search does not escape the blue territory is subtle. It depends on the very depth-first nature of NDFS! Proving the main invariants on the NDFS colours will also complete the termination proof.

3.3 Main Local Invariants on NDFS Colours

In order to prove the main invariant `Red` $\subseteq$ `Blue` we have to provide several additional invariants. These invariants are needed in the termination proof, but they will be reused in the completeness proof of NDFS. All invariants in this section can be proved locally, without reasoning about the whole graph.

We now come to the formulation of the main invariants. They capture the very basic idea of Depth-First Search: A node is only coloured `blue` if its successors are processed, i.e. they are coloured `blue` or `cyan`. Similarly, all successors of `red` nodes are `red` or `pink`. We express these invariants concisely with a special predicate `Next`, where `Next(G,X,Y)` indicates that all successors in G of nodes X are in Y. See Figure 5 for the statement of the main invariants.

Another important local property is that there will never be an edge from a red node to a cyan node, `Next(G,Red(G),G-Cyan(G))`. This is guaranteed by the cycle detection in `dfsred` at (Section A.5, ℓ. 38).

For the complete proof we refer to Section A. One of the subtleties is that in `dfsred` (Section A.5, ℓ. 50) we colour the start node `red`, just before it

```
1  predicate Next(G:set⟨Node⟩,X:set⟨Node⟩,Y:set⟨Node⟩)
2    reads G; requires graph(G);
3  { ∀ n,i • n ∈ G ∧ 0 ≤ i < |n.next| ⟹ ( n ∈ X ⟹ n.next[i] ∈ Y ) }
4  ...
5  invariant Red(G) ⊆ Blue(G);
6  invariant Next(G, Blue(G), Blue(G) ∪ Cyan(G));
7  invariant Next(G, Red(G) , Red(G) ∪ Pink(G));
8  invariant Next(G, Red(G) , G − Cyan(G));
```

Fig. 5. Stating the main local invariants on the colours in NDFS

becomes `blue`, temporarily violating the main invariant. This is solved by remembering the starting point of `dfsred` in a ghost variable `ghost root:Node` (Section A.5, ℓ. 1). The invariants on `Blue` in `dfsred` are modified to `Blue` $\cup$ {`root`} (Section A.5, ℓ. 8, 10, 19, 32). Also, we must explicitly state that all successors of s up to i are in `Blue` $\cup$ `Cyan` (Section A.4, ℓ. 33), or `Red` $\cup$ `Pink` (Section A.5, ℓ. 29), respectively. In order to prove this under the given conditions, we introduce an invariant on the exact `types` that the two colour variables may assume (Section A.1, ℓ. 16-20).

Adding the invariants expands the DAFNY code considerably, since most invariants must be repeated six times: before and after each recursive call, and in the while loops. See for instance the six occurrences of `invariant types(G)` in Section A.4, A.5.

At this point, DAFNY is happy, since the code is guaranteed to terminate without run time errors. This run takes about 10 seconds (on a 2.7GHz Macbook).

```
Dafny program verifier finished with 13 verified, 0 errors
```

3.4 Completeness

We can now proceed to specify and prove that NDFS accomplishes a useful task. The correctness criterion is that the result `found` indicates correctly whether the graph G has an accepting cycle. In order to specify this, we first define the notions of paths and cycles, in terms of sequences of nodes, Figure 6.

A sequence of nodes p is a path from x to y in graph G if it starts with x, ends with y, and successive members are linked by an edge in G. A reachable accepting cycle is defined as a lasso: a path p from root x to accepting state y, and the cycle is a non-empty path q from y to itself.

Next, correctness of `ndfs` is ensured, distinguishing soundness and completeness. In the rest of this section, we prove completeness, i.e. the algorithm does not miss an accepting cycle. To this end we state the key invariant (Figure 6, ℓ. 13-15). The completeness proof consists of two parts: proving the key invariant, that no blue nodes can have an accepting cycle, and proving that all nodes will be `blue` if `ndfs` terminates with `found=false`.

Let us see what we know after `dfsblue(root)` terminates with `found=false`; we refer to the line numbers in Section A.4. We have already proved the invariant `Next(G,Blue(G),Blue(G)` $\cup$ `Cyan(G))` (ℓ. 16). Since `Old(Cyan)=Cyan` (ℓ. 19) and `Cyan=`{} initially (nodes start white), we obtain

```
1  function Path(G: set⟨Node⟩, x: Node, y: Node, p: seq⟨Node⟩): bool
2    reads G; requires graph(G);
3  { |p| > 0 ∧ p[0] = x ∧ p[|p|−1] = y
4    ∧ ∀ i • 0 ≤ i < |p|−1 ⟹ p[i] ∈ G ∧ p[i+1] ∈ p[i].next }
5
6  function Cycle(G: set⟨Node⟩, x: Node, y: Node, p: seq⟨Node⟩, q: seq⟨Node⟩): bool
7    reads G; requires graph(G);
8  { Path(G,x,y,p) ∧ Path(G,y,y,q) ∧ |q| > 1 ∧ y.accepting }
9
10 ensures found ⟹ (∃ a,p,q • Cycle(G, root, a,p,q));   // soundness
11 ensures (∃ s,p,q • Cycle(G, root, s,p,q)) ⟹ found;   // completeness
12
13 function KeyInvariant(G: set⟨Node⟩): bool
14   reads G; requires graph(G);
15 { ∀ s • s ∈ Blue(G) ∧ s.accepting ⟹ ¬ ∃ p • |p| > 1 ∧ Path(G,s,s,p) }
```

Fig. 6. Specification of the full functional correctness and key invariant of NDFS

`Next(G,Blue(G),Blue(G))`. Since `root.color1=blue` (ℓ. 18), we can now prove inductively that all reachable nodes are indeed in `Blue(G)`. So if the Key Invariant holds, there cannot be an accepting cycle.

Since finding inductive proofs is beyond the capabilities of Dafny, we must prove this with a separate lemma. In Dafny, an inductive proof corresponds to a recursive function that establishes the correct post condition. Function `NoCycle` (Section A.2, ℓ. 15-21) represents a proof by induction over the reachable nodes that the key invariant indeed implies that there is no accepting cycle. Note that we have to explicitly apply this lemma in `ndfs` (Section A.3, ℓ. 9).

Next we must still prove the Key Invariant (ℓ. 10, 23, 38). The crucial step is just before we assign `s.color1 = blue` for an accepting state s (ℓ. 58). At this point we apply a new lemma, `NoPath`, which basically reasons about the result of `dfsred`, with another inductive argument.

So what do we know after calling `dfsred` when `found=false`? We now refer to line numbers in Section A.5. We already proved `Next(G,Red(G),Red(G) ∪ Pink(G))` (ℓ. 18). Since `Old(Pink)=Pink` (ℓ. 16) and `Pink={}` before the call (Section A.4, ℓ. 31) we obtain `Next(G,Red(G),Red(G))`. Since `s.color2=red` (ℓ. 15), we can now prove inductively that all reachable nodes are in `Red(G)`. One of our main invariants on colours is that there is no edge between red nodes and cyan nodes (ℓ. 11). So indeed, the `root` node, which is still cyan, is not reachable.

Again, this requires an inductive argument, which is provided by the recursive function `NoPath` (Section A.2, ℓ. 5-13) that establishes the correct post-condition.

3.5 Soundness

The final task is to prove soundness, i.e., if `ndfs` reports `found=true`, then there exists an accepting cycle. This is intuitively an easy task, since the stack of the program execution corresponds to the accepting cycle (cf. the cyan and pink nodes in Figure 2). However, we have no access to the stack. Actually, the soundness proof posed some verification challenges to the underlying Z3 SMT solver, since it introduces quantifications over sequences. To limit the search

```
1  method ndfs(root:Node)
2  { try
3    { dfsblue(root);
4      assert ¬CycleExists;
5    }
6    catch CycleFound ⇒ { assert CycleExists; }
7  }
8
9  method dfsblue(s:Node) raises CycleFound
10 { s.color1 := cyan;
11   var i := 0;
12   while (i < |s.next|)
13   { var t := s.next[i];
14     i := i+1;
15     if (t.color1 = white) { dfsblue(t); }
16   }
17   if (s.accepting) { dfsred(s); }
18   s.color1 := blue;
19 }
20
21 method dfsred(s:Node) raises CycleFound
22 { s.color2 := pink;
23   var i := 0;
24   while (i < |s.next|)
25   { var t := s.next[i];
26     i := i+1;
27     if (t.color1 = cyan) { raise CycleFound; }
28     if (t.color2 = white) { dfsred(t); }
29   }
30   s.color2 := red;
31 }
```

Fig. 7. Specification in DAFNY language extended with exceptions

space, DAFNY does not try extensively to find a witness to an obligation of the form `exists p:seq<<Node>>`. So at certain places in the program we must add assertions, to suggest the correct witnesses. Alternatively, one could add ghost variables to manipulate paths explicitly, as done in [10].

The blue search ensures that when `found=true`, there is indeed an accepting cycle (Section A.4, ℓ. 22). The assertion at (ℓ. 46) shows how this cycle is constructed from the path obtained from the recursive call. In the other case, at (ℓ. 55), the situation is less trivial. Here we apply `lemma CycleFoundHere` to reconstruct the cycle.

Let us first consider this situation. Assume that the blue search started in node s, calls the red search from an accepting node t, which hits a cyan state r. Note that r is not necessarily accepting. In this case, the accepting cycle consists of the prefix s to t, followed by the loop t to r back to t ($s = 0$, $t = 5$, $r = 1$ in Figure 2).

The fact that there is a path from r to t is not obvious. We added a new precondition (Section A.4, ℓ. 11) that from any cyan state c (in particular r) there is a path to the current state s. When we color s cyan, the path is trivial, but still we must assert it (ℓ. 26). Before the recursive call (ℓ. 43) we use `lemma NextCyan` (Section A.2, ℓ. 23-29) to tell DAFNY how the path to the next cyan node is constructed by concatenation.

To reconstruct the path from t to some r, the red search ensures that if `found=true`, there is a path from the current state to some cyan state

(Section A.5, ℓ. 21). The assertion at (ℓ. 39) indicates how this path is constructed in case the cyan state is found, and (ℓ. 45) indicates how the path is created from the path obtained in the recursive call.

Lemma `CycleFoundHere` (Section A.2, ℓ. 31-41) checks the reasoning that we sketched above. It was fairly non-trivial to convince Dafny that this construction is correct, even though it is basic first-order reasoning without induction. Actually, the interaction with Dafny at this point was quite inconvenient: Dafny just tells that the proof does not go through, and the user has to find out, via numerous assertions, which facts Dafny does or does not see. This small part of the proof would have been easier with an interactive theorem prover. But then it is extra rewarding to see (in 2 minutes):

```
Dafny program verifier finished with 22 verified, 0 errors
```

4 Conclusion

The main conclusion is that verification of recursive graph algorithms with automatic program verifiers is feasible. In particular, the functional correctness proof of NDFS with Dafny was successful.

Success Factors. One of the success factors is the rich specification language of Dafny. We heavily depended on set values (for sets of nodes with a particular colour) and sequence values (representing paths and cycles in the graph). We also made extensive use of quantifiers. We feel that every line of the specification is straightforward and understandable. Also, the recursive nature of the algorithm did not pose any problem.

Another success factor is the power of the SMT solver Z3, and the error reporting by Dafny. In nearly all cases of a failed verification Dafny came back with a line number and a diagnosis of the cause, on which the user could take action.

This was the first experience with Dafny by the author, or with any automatic program verifier at all. Still it took only a couple of weeks to finish the complete proof. Here it should be noted that the author was already familiar with the details of the NDFS algorithm, and also with interactive theorem provers.

Finally, the proof strategy to split local invariants on colours from inductive arguments on paths helped to structure the proof. These invariants contribute to the global understanding of the NDFS algorithm. Also, the modular approach was essential to build up the specification incrementally, even though Dafny does not provide extra support for specification refinement. But also it was necessary to keep the verification task manageable.

Useful Extensions to Dafny. There are still a few issues where Dafny could be improved to be even more useful. The complete verification takes about 2 minutes, which is fine. However, when the user checks intermediate attempts frequently, 2 minutes imply a considerable slowdown. But even worse, Dafny (or rather Z3) chokes on failed proof attempts: Dafny simply does not come

back at all within a reasonable amount of time. This was the main reason to follow an incremental approach. Maybe the IDE interface to Dafny would have better supported an incremental approach.

There are also two reasons why the specification has increased more than necessary. First, many invariants are repeated six times: in both while-loops, and before and after each recursive procedure. This could be mitigated by allowing an `invariant` keyword for recursive functions. A more drastic solution would be to generate invariants, possibly guided by some hints. For example, specify once `hint types(G);` instead of six times as in Section A.4, A.5.

The other extension on the wish list is exceptions. Several lines in the plain code (Figure 1) just handle return values. A more natural coding would use exceptions, as illustrated in (Figure 7). This is probably a non-trivial extension to the verification condition generator in Dafny.

Finally, SMT solving for full first-order logic is necessarily incomplete. So, when Dafny reports that an assertion at some line number fails, that assertion may hold or not. No further diagnostic information is given. At one place we struggled hard to come up with the three intermediate assertions to convince Dafny, cf. `CycleFoundHere` in (Section A.2, ℓ. 38-40). It would be nice if the user had the possibility to fall back on an interactive proof session to deal with such cases, in order to avoid blind guessing.

Perspectives for Future Work. It is now possible to easily play with variants of NDFS, for instance those introduced by [5]. After submission of the paper, the author modified the code and proof in a couple of hours to the 2-bit version of [13], basically by replacing the pink colour by a ghost variable `OnStack`. The basic setup might also be reused to automate the verification of other DFS algorithms, e.g. SCC-based algorithms [9]. A more challenging assignment would be to include partial-order reduction, the LTL-to-Büchi translation, or the operational semantics of a modeling language. It is not clear that these tasks are in the scope of an automatic program verifier right now.

The most useful extension would be the application to parallel graph algorithms, like the parallel NDFS in [8]. This would require a program verifier for multi-threaded programs that synchronize by reading and writing colours on a shared graph. Such a tool would help researchers in developing parallel graph algorithms, especially when small input graphs could be generated as counter examples for faulty programs.

Acknowledgement. The author is grateful to the organisers and participants of Dagstuhl Seminar 14171 [1] in April 2014 on Software Verification System benchmarks, which initiated this research. In particular, the author is thankful to Rustan Leino for creating Dafny, and supporting the author during and after the workshop. The reviewers of this paper provided several useful suggestions to improve the presentation.

References

1. Beyer, D., Huisman, M., Klebanov, V., Monahan, R.: Evaluating Software Verification Systems: Benchmarks and Competitions (Dagstuhl Reports 14171). Dagstuhl Reports 4(4), 1–19 (2014)
2. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-Efficient Algorithms for the Verification of Temporal Properties. Formal Methods in System Design 1(2/3), 275–288 (1992)
3. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013)
5. Gaiser, A., Schwoon, S.: Comparison of Algorithms for Checking Emptiness on Büchi Automata. CoRR, abs/0910.3766 (2009)
6. Gava, F., Fortin, J., Guedj, M.: Deductive verification of state-space algorithms. In: Johnsen, E.B., Petre, L. (eds.) IFM 2013. LNCS, vol. 7940, pp. 124–138. Springer, Heidelberg (2013)
7. Holzmann, G.J., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: The Spin Verification System, pp. 23–32. American Mathematical Society (1996)
8. Laarman, A.W., Langerak, R., van de Pol, J.C., Weber, M., Wijs, A.: Multi-Core Nested Depth-First Search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
9. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Heidelberg (2014)
10. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
11. Pottier, F.: Depth-first search and strong connectivity in Coq. Journées Francophones des Langages Applicatifs (JFLA 2015) (January 2015)
12. Ray, S., Matthews, J., Tuttle, M.: Certifying compositional model checking algorithms in ACL2. In: IW on ACL2 Theorem Prover and its Applications (2003)
13. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
14. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998)
15. Sun, J., Liu, Y., Cheng, B.: Model checking a model checker: A code contract combined approach. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 518–533. Springer, Heidelberg (2010)
16. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 332–344. Cambridge (1986)

A Full NDFS Proof in Dafny

This section contains the full specification of the NDFS algorithm and is completely verified with Dafny. The verification was run with Dafny version 1.8.2.10419 on a Macbook 2.7 GHz Intel Core i7 processor with 8GB RAM under MacOS 10.10.1 and Mono version 3.2.5. The verification time varies around 2 minutes. The code in this section has been typeset with `dafny.sty` by Rustan Leino, obtained from https://searchcode.com/codesearch/view/28108731/.

A.1 Basic Definitions

```
datatype Color = white | cyan | blue | pink | red;

class Node {
  var next: seq⟨Node⟩;
  var accepting: bool;
  var color1: Color;
  var color2: Color;
}

ghost var G: set⟨Node⟩;

predicate graph(G:set⟨Node⟩)
  reads G;
{ ∀ m • m ∈ G ⟹ (m ≠ null ∧ ∀ n • n ∈ m.next ⟹ n ∈ G) }

predicate types(G:set⟨Node⟩)
  reads G; requires graph(G);
{ ∀ m • m ∈ G ⟹
     m.color1 ∈ {white,cyan,blue}
  ∧  m.color2 ∈ {white,pink,red}}

function Cyan(G:set⟨Node⟩): set⟨Node⟩
  reads G; requires graph(G);
{ set n | n ∈ G ∧ n.color1 = cyan • n }

function Blue(G:set⟨Node⟩): set⟨Node⟩
  reads G; requires graph(G);
{ set n | n ∈ G ∧ n.color1 = blue • n }

function Pink(G:set⟨Node⟩): set⟨Node⟩
  reads G; requires graph(G);
{ set n | n ∈ G ∧ n.color2 = pink • n }

function Red(G:set⟨Node⟩): set⟨Node⟩
  reads G; requires graph(G);
{ set n | n ∈ G ∧ n.color2 = red • n }

predicate Next(G:set⟨Node⟩,X:set⟨Node⟩,Y:set⟨Node⟩)
  reads G; requires graph(G);
{ ∀ n,i • n ∈ G ∧ 0 ≤ i < |n.next| ⟹ ( n ∈ X ⟹ n.next[i] ∈ Y ) }

function Path(G:set⟨Node⟩,x:Node,y:Node,p:seq⟨Node⟩):bool
  reads G; requires graph(G);
{ |p| > 0 ∧ p[0] = x ∧ p[|p|−1] = y
  ∧ ∀ i • 0 ≤ i < |p|−1 ⟹ p[i] ∈ G ∧ p[i+1] ∈ p[i].next }

function Cycle(G:set⟨Node⟩,x:Node,y:Node,p:seq⟨Node⟩,q:seq⟨Node⟩):bool
  reads G; requires graph(G);
{ Path(G,x,y,p) ∧ Path(G,y,y,q) ∧ |q| > 1 ∧ y.accepting }
```

A.2 Auxiliary Lemmas on Paths and Cycles

```
function KeyInvariant(G:set⟨Node⟩):bool
  reads G; requires graph(G);
{ ∀ s • s ∈ Blue(G) ∧ s.accepting ⟹ ¬∃ p • |p| > 1 ∧ Path(G,s,s,p) }

function NoPath(G:set⟨Node⟩,s:Node,t:Node,p:seq⟨Node⟩):bool
  reads G; requires graph(G);
  requires Next(G,Red(G),Red(G));
  requires Next(G,Red(G),G − Cyan(G));
  requires s ∈ Red(G);
  requires t ∈ Cyan(G);
  ensures NoPath(G,s,t,p);
  ensures |p| > 1 ⟹ ¬Path(G,s,t,p);
{ |p| > 1 ∧ p[0] = s ∧ p[1] ∈ p[0].next ⟹ NoPath(G,p[1],t,p[1..]) }

function NoCycle(G:set⟨Node⟩,root:Node,s:Node,p:seq⟨Node⟩,q:seq⟨Node⟩):bool
  reads G; requires graph(G);
  requires root ∈ Blue(G);
  requires Next(G,Blue(G),Blue(G));
  requires KeyInvariant(G);
  ensures ¬Cycle(G,root,s,p,q);
{ |p| > 1 ∧ p[0] = root ∧ p[1] ∈ p[0].next ⟹ NoCycle(G,p[1],s,p[1..],q) }

lemma NextCyan(G:set⟨Node⟩,s:Node,t:Node)
  requires graph(G);
  requires s ∈ G;
  requires t ∈ s.next;
  requires ∀ c • c ∈ Cyan(G) ⟹ ∃ q • Path(G,c,s,q);
  ensures ∀ c • c ∈ Cyan(G) ⟹ ∃ q • Path(G,c,t,q);
{ assert ∀ c,p • Path(G,c,s,p) ⟹ Path(G,c,t,p+[t]); }

lemma CycleFoundHere(G:set⟨Node⟩,s:Node)
   requires graph(G);
   requires s ∈ G;
   requires s.accepting;
   requires ∃ c,p • c ∈ Cyan(G) ∧ Path(G,s,c,p) ∧ |p|>1;
   requires ∀ c • c ∈ Cyan(G) ⟹ ∃ q • Path(G,c,s,q);
   ensures ∃ p,q • Cycle(G,s,s,p,q);
{ assert ∃ c,p,q • c ∈ Cyan(G) ∧ |p|>1 ∧ Path(G,s,c,p) ∧ Path(G,c,s,q);
  assert ∀ c,p,q • Path(G,s,c,p) ∧ Path(G,c,s,q) ⟹ Path(G,s,s,p+q[1..]);
  assert ∀ q • |q| > 1 ∧ Path(G,s,s,q) ⟹ Cycle(G,s,s,[s],q);
} // this was very hard to prove and rather sensitive..
```

A.3 Main Method and Correctness Statement

```
method ndfs(root:Node) returns (found:bool)
  requires graph(G);
  requires root ∈ G;
  requires ∀ s • s ∈ G ⟹ s.color1 = s.color2 = white;
  modifies G`color1, G`color2;
  ensures found ⟹ (∃ a,p,q • Cycle(G,root,a,p,q)); // soundness
  ensures (∃ s,p,q • Cycle(G,root,s,p,q)) ⟹ found; // completeness
{ found := dfsblue(root);
  assert ¬found ⟹ ∀ s,p,q • NoCycle(G,root,s,p,q) ⟹ ¬Cycle(G,root,s,p,q);
}
```

A.4 Blue Search

```
1  method dfsblue(s:Node) returns (found:bool)
2    requires s ∈ G;
3    requires graph(G);
4    requires types(G);
5    requires s.color1 = white;
6    requires Next(G,Blue(G),Blue(G) ∪ Cyan(G));
7    requires Next(G,Red(G),Red(G) ∪ Pink(G));
8    requires Pink(G) = {};
9    requires Red(G) ⊆ Blue(G);
10   requires KeyInvariant(G);
11   requires ∀ c• c ∈ Cyan(G) ⟹ ∃ p • Path(G,c,s,p);
12   modifies G`color1, G`color2;
13   decreases G − Cyan(G);
14   ensures types(G);
15   ensures old(Blue(G)) ⊆ Blue(G);
16   ensures Next(G,Blue(G),Blue(G) ∪ Cyan(G));
17   ensures Next(G,Red(G),Red(G) ∪ Pink(G));
18   ensures ¬found ⟹ s ∈ Blue(G);
19   ensures ¬found ⟹ old(Cyan(G)) = Cyan(G);
20   ensures ¬found ⟹ Pink(G) = {};
21   ensures ¬found ⟹ Red(G) ⊆ Blue(G);
22   ensures found ⟹ (∃ a,p,q • Cycle(G,s,a,p,q));
23   ensures KeyInvariant(G);
24
25 { s.color1 := cyan;
26   assert Path(G,s,s,[s]);
27   var i := 0;
28   while (i < |s.next|)
29   invariant types(G);
30   invariant Cyan(G) = old(Cyan(G)) ∪ {s};
31   invariant Pink(G) = {};
32   invariant i ≤ |s.next|;
33   invariant ∀ j • 0 ≤ j < i ⟹ s.next[j] ∈ Blue(G) ∪ Cyan(G);
34   invariant old(Blue(G)) ⊆ Blue(G);
35   invariant Next(G,Blue(G),Blue(G) ∪ Cyan(G));
36   invariant Next(G,Red(G),Red(G) ∪ Pink(G));
37   invariant Red(G) ⊆ Blue(G);
38   invariant KeyInvariant(G);
39
40   { var t := s.next[i];
41     i := i+1;
42     if (t.color1 = white)
43     { NextCyan(G,s,t);
44       found := dfsblue(t);
45       if (found) {
46         assert ∀ a,p,q • Cycle(G,t,a,p,q) ⟹ Cycle(G,s,a,[s]+p,q);
47         return;
48       }
49     }
50   }
51   if (s.accepting)
52   { assert s ∉ Pink(G);
53     found := dfsred(s,s);
54     if (found) {
55       CycleFoundHere(G,s);
56       return;
57     }
58     assert ∀ p • NoPath(G,s,s,p);
59   }
60   s.color1 := blue;
61   return false;
62 }
```

A.5 Red Search

```
1  method dfsred(s:Node, ghost root:Node) returns (found:bool)
2    requires graph(G);
3    requires types(G);
4    requires s ∈ G;
5    requires root ∈ G;
6    requires s.color2 = white;
7    requires s = root ∨ s.color1 = blue;
8    requires Next(G,Blue(G) ∪ {root},Blue(G) ∪ Cyan(G));
9    requires Next(G,Red(G),Red(G) ∪ Pink(G));
10   requires Red(G) ⊆ Blue(G) ∪ {root};
11   requires Next(G,Red(G),G − Cyan(G));
12   modifies G`color2;
13   decreases G − Pink(G);
14   ensures types(G);
15   ensures ¬found ⟹ s ∈ Red(G);
16   ensures ¬found ⟹ old(Pink(G)) = Pink(G);
17   ensures old(Red(G)) ⊆ Red(G);
18   ensures Next(G,Red(G),Red(G) ∪ Pink(G));
19   ensures Red(G) ⊆ Blue(G) ∪ {root};
20   ensures Next(G,Red(G),G − Cyan(G));
21   ensures found ⟹ ∃ p,c • |p| > 1 ∧ c ∈ Cyan(G) ∧ Path(G,s,c,p);
22
23 { s.color2 := pink;
24   var i := 0;
25   while (i < |s.next|)
26   invariant types(G);
27   invariant old(Red(G)) ⊆ Red(G);
28   invariant i ≤ |s.next|;
29   invariant ∀ j • 0 ≤ j < i ⟹ s.next[j] ∈ Red(G) ∪ Pink(G);
30   invariant Next(G,Red(G),Red(G) ∪ Pink(G));
31   invariant Pink(G) = old(Pink(G)) ∪ {s};
32   invariant Red(G) ⊆ Blue(G) ∪ {root};
33   invariant Next(G,Red(G),G − Cyan(G));
34   invariant ∀ j • 0 ≤ j < i ⟹ s.next[j] ∈ G − Cyan(G);
35
36   { var t := s.next[i];
37     i := i+1;
38     if (t.color1 = cyan) {
39       assert Path(G,s,t,[s,t]);
40       return true;
41     }
42     if (t.color2 = white)
43     { found := dfsred(t,root);
44       if (found) {
45         assert ∀ p,c • Path(G,t,c,p) ⟹ Path(G,s,c,[s]+p);
46         return;
47       }
48     }
49   }
50   s.color2 := red;
51   return false;
52 }
```