Jaco C. van de Pol: Automated Verification of Nested DFS

FMICS: Formal Methods for Industrial Critical Systems
20th International Workshop, Oslo, Norway, June 22-23, 2015
pages 181-197

Overview

- model checking: memory and time intensive
- construction of model checkers is intricate and error-prone
- so they become part of the critical engineering infrastructure
- should be verified themselves

Overview

- a method: using interactive theorem provers, where users create a proof, which is checked by the machine
- the level of automation can be raised, e.g. by using automatic program verifiers
- this paper verifies the nested depth-first search algorithm (NDFS) in an incremental manner

Table of Contents

NDFS and Dafny

- Dafny: an automatic program verifier created by Rustan Leino
- provides a straightforward imperative programming language (classes and dynamic allocation)
- it supports sequential programs mixed with specification annotations:
- preconditions (requires), postconditions (ensures), invariants and termination metrics (decreases)
- the specification language is quite liberal

NDFS and Dafny

- Dafny parses and type-checks the program
- generates proof obligations to guarantee absence of runtime errors, termination, and the validity of all specification annotations
- works in a modular fashion, method by method

NDFS and Dafny

- the automata based approach reduces the LTL model checking problem to the detection of accepting cycles
- NDFS: a linear-time algorithm to detect accepting cycles
- performs a first (blue) DFS to detect accepting states, then 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
- is heavily used as the core algorithm of LTL model checkers

NDFS and 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 }
```

the initial implementation

NDFS and Dafny

enumeration of possible colours

```
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 }
```

the initial implementation

NDFS and 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 }
```

represents the nodes of the underlying graph

the initial implementation

NDFS and 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 }
```

the main algorithm

named return value, can be used in method body; indicates whether an accepting cycle is found or not

the initial implementation

NDFS and 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 }
```

a recursive method to visit all nodes at most once;
calls dfsred() on accepting nodes

the initial implementation

NDFS and 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 }
```

a recursive method to detect cycles;
also visits each node at most once

the initial implementation

Developing the Correctness Proof

- the verification was carried out incrementally

1. runtime errors are eliminated by appropriate preconditions
2. termination is addressed

- key approach: identify invariants on the local properties of the colours in the graph

Developing the Correctness Proof

- the initial implementation was not regarded correct by Dafny:

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

- explicit permission to modify objects is required
- if the root node is null, then reading its successor nodes would lead to a runtime error
- solution to the latter problem: a ghost variable and the predicate graph(G) is defined

Developing the Correctness Proof

```
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 { ... }
```

specifying a well-defined and closed graph

Developing the Correctness Proof

```
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 { ... }
```

ghost variable; a set of Node objects

specifying a well-defined and closed graph

Developing the Correctness Proof

```
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 { ... }
```

the predicate is a simple method which returns a boolean;
ensures that no node is null in G and every successor of a node is in G

specifying a well-defined and closed graph

Developing the Correctness Proof

```
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 { ... }
```

all methods are equipped with preconditions which require the arguments to be from the valid G graph;
graph(G) is closed, so there's no risk of null values

specifying a well-defined and closed graph

Developing the Correctness Proof

- Dafny is still not satisfied :(
- the while loops are correct, but the recursive calls lead to the following complaint:

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

- we must guarantee that every node must be visited at most twice: once during `dfsblue()` and once during `dfsred()`

Developing the Correctness Proof

- we need to require that initially all nodes are white and we only meet white nodes along the way
- `dfsblue()` must not modify the `Cyan` set, or it will lead to non-termination for calls to subsequent successors
- `t.color2=white` must be guaranteed

Developing the Correctness Proof

```
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 { ... }
```

specifying decreasing termination functions

Developing the Correctness Proof

```
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 { ... }
```

defines the Cyan set:
set of all nodes in G with color1 = cyan

specifying decreasing termination functions

Developing the Correctness Proof

```
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 { ... }
```

requires that at the start, color1 and color2 is white for all nodes

specifying decreasing termination functions

Developing the Correctness Proof

```
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 { ... }
```

requires that color1 of the argument is white;
states that the number of non-cyan nodes is decreased in each call;

ensures that Cyan(G) is not modified if no accepting cycle is found

the invariant reasons about the value of Cyan during and after the loop

specifying decreasing termination functions

Developing the Correctness Proof

- Dafny still complains that the precondition of `dfsred()` is not guaranteed (`color2` is white):

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

- several additional invariants must be provided on the NDFS colours

Developing the Correctness Proof

```
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));
```

stating the main local invariants on the colours

Developing the Correctness Proof

```
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));
```

Next(G,X,Y) indicates that all nodes in G that are from X have successors only from Y

stating the main local invariants on the colours

Developing the Correctness Proof

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

Completeness and Soundness

- prove that NDFS accomplishes a useful task
- `found` should indicate whether the graph G has an accepting cycle
- paths and cycles must be defined in terms of sequences of nodes
- completeness: no blue nodes have an accepting cycle and all nodes are blue when `found` is false
- soundness: if `found` is true there must be an accepting cycle
- intuitive to prove with the stack of the program, but during verification there is no access to stack

Completeness and Soundness

```
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) }
```

specification of the full functional correctness and key invariant of NDFS

Completeness and Soundness

```
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) }
```

soundness: if found is true, there must be an accepting cycle
completeness: if an accepting cycle exists, found will be true

specification of the full functional correctness and key invariant of NDFS

Conclusion

- verification of recursive graph algorithms with automatic program verifiers is feasible
- functional correctness proof of NDFS with Dafny was successful
- success factors: rich specification language (set values, sequence values, quantifiers) and useful, actionable error reporting (with line numbers and diagnosis of cause)

Conclusion

- some useful extensions to Dafny:
- 2 minutes for complete verification is fine once, but for repeated attempts it implies considerable slowdown
- on failed proof attempts Dafny does not come back within a reasonable amount of time (main reason for incremental approach)
- exceptions to mitigate superfluous handling of return values with more natural coding (possibly non-trivial for the verification condition generator in Dafny)

Thank you for your attention!