

# Formal Proofs of Tarjan’s Strongly Connected Components Algorithm in Why3, Coq and Isabelle

Ran Chen

Institute of Software of the Chinese Academy of Sciences, Beijing  
chenr@ios.ac.cn

Cyril Cohen

Université Côte d’Azur, Inria  
cyril.cohen@inria.fr

Jean-Jacques Lévy

Irif & Inria Paris  
jean-jacques.levy@inria.fr

Stephan Merz

Université de Lorraine, CNRS, Inria, LORIA  
stephan.merz@inria.fr

Laurent Théry

Université Côte d’Azur, Inria  
laurent.thery@inria.fr

---

## Abstract

Comparing provers on a formalization of the same problem is always a valuable exercise. In this paper, we present the formal proof of correctness of a non-trivial algorithm from graph theory that was carried out in three proof assistants: Why3, COQ, and Isabelle.

**2012 ACM Subject Classification** Logic and verification, Automated reasoning, Higher order logic

**Keywords and phrases** Mathematical logic, Formal proof, Graph algorithm, Program verification

## 1 Introduction

Graph algorithms are notoriously obscure in the sense that it is hard to grasp why exactly they work. Therefore proof of correctness are more than welcome in this domain. In this paper, we consider Tarjan’s algorithm [28] for discovering the strongly connected components in a directed graph and present a formal proof of its correctness in three different systems: Why3, COQ and Isabelle/HOL. The algorithm is treated at an abstract level with a functional programming style manipulating finite sets, stacks and mappings, but it respects the linear time behaviour of the original presentation.

To our knowledge this is the first time that the formal correctness proof of a non-trivial program is carried out in three very different proof assistants: Why3 is based on a first-order logic with inductive predicates and automatic provers, COQ on an expressive theory of higher-order logic and dependent types, and Isabelle/HOL combines higher-order logic with automatic provers. We claim that our proof is direct, readable, elegant, and follows Tarjan’s presentation. Crucially for our comparison, the algorithm is defined at the same level of abstraction in all three systems, and the proof relies on the same arguments in the three formal systems. Note that a similar exercise but for a much more elementary proof (the irrationality of square root of 2) and using many more proof assistants (17) was presented in [32].

Formal and informal proofs of algorithms about graphs were already performed in [24, 30, 25, 13, 17, 29, 19, 27, 26, 15, 8]. Some of them are part of a larger library, others focus on the treatment of pointers or on concurrent algorithms. In particular, only Lammich and

## 2 Formal proofs of Tarjan’s SCC algorithm

45 Neumann [17] gave an alternative formal proof of Tarjan’s algorithm within their framework  
46 for verifying graph algorithms in Isabelle/HOL.

47 We expose here the key parts of the proofs. The interested reader can access the details  
48 of the proofs and run them on the web [7, 9, 20]. In this paper, we recall the principles of the  
49 algorithm in section 2; we describe the proofs in the three systems in sections 3, 4, and 5 by  
50 emphasizing the differences induced by the logics which are used; we conclude in sections 6  
51 and 7 by commenting the developments and advantages of each proof system.

### 52 **2** The algorithm

In a directed graph, two vertices  $x$  and  $y$  are strongly connected if there exists a path from  $x$  to  $y$  and a path from  $y$  to  $x$ . A strongly connected component (scc) is a maximal set of vertices where all pairs of vertices are strongly connected. A fundamental property relates sccs and depth-first search (DFS) traversal in a directed graph: each scc is a prefix of a single subtree in the corresponding spanning forest (see figure 1c). Its root is named the base of the scc. Tarjan’s algorithm [28] relies on the detection of these bases and collects the sccs in a pushdown stack. It performs a single DFS traversal of the graph assigning a serial number  $num[x]$  to any vertex  $x$  in the order of the visit. It computes the following function for every vertex  $x$ :

$$LOWLINK(x) = \min\{num[y] \mid x \xRightarrow{*} z \hookrightarrow y \wedge x \text{ and } y \text{ are in the same scc}\}$$

53 The relation  $x \xRightarrow{*} z$  means that  $z$  is a son of  $x$  in the spanning forest, the relation  $\xRightarrow{*}$  is  
54 its transitive and reflexive closure, and  $z \hookrightarrow y$  means that there is a cross-edge from  $z$  to  
55  $y$  in the spanning forest (a cross-edge is an edge of the graph which is not an edge in the  
56 spanning forest). In figure 1c,  $\xRightarrow{*}$  is drawn in thick lines and  $\hookrightarrow$  in dotted lines; in figure 1b  
57 the table of the values of the *LOWLINK* function is shown. The minimum of the empty set  
58 is assumed to be  $+\infty$  (this is a slight simplification w.r.t. the original algorithm).

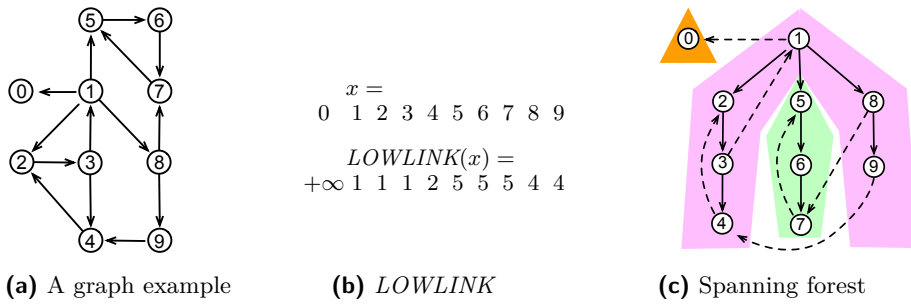
The base  $x$  of an scc is found when  $LOWLINK(x) \geq num[x]$ , and the component is formed by the nodes of the subtree rooted at  $x$  and pruned of the sccs already discovered in that subtree. Notice that  $LOWLINK(x)$  need neither be the lowest serial number of a vertex accessible from  $x$ , nor of an ancestor of  $x$  in the spanning forest. Take for instance, vertices 8 or 9 in figure 1c. Moreover, the DFS traversal sets to  $+\infty$  the serial numbers of vertices in already discovered sccs. The definition of *LOWLINK* can therefore be written as:

$$LOWLINK(x) = \min\{num[y] \mid x \xRightarrow{*} z \hookrightarrow y\}$$

59 Our implementation of graphs uses an abstract type *vertex* for vertices, a constant *vertices*  
60 for the finite set of all vertices in the graph, and a *successors* function from vertices to their  
61 adjacency set. The algorithm maintains an environment  $e$  implemented as a record of type  
62 *env* with four fields: a stack  $e.stack$ , a set  $e.sccs$  of strongly connected components, a fresh  
63 serial number  $e.sn$ , and a function  $e.num$  from vertices to serial numbers.

```
64 type vertex  
65 constant vertices: set vertex  
66 function successors vertex : set vertex  
67 type env = {stack: list vertex; sccs: set (set vertex); sn: int; num: map vertex int}
```

70 The DFS traversal is organized by two mutually recursive functions *dfs1* and *dfs*. The  
71 function *dfs1* visits a new vertex  $x$  and computes  $LOWLINK(x)$ . Furthermore it adds a new  
72 scc when  $x$  is the base of a new scc. The function *dfs* takes as argument a set  $r$  of roots and



■ **Figure 1** The vertices are numbered and pushed onto the stack in the order of their visit by the recursive function *dfs1*. When the first component  $\{0\}$  is discovered, vertex 0 is popped; similarly when the second component  $\{5, 6, 7\}$  is found, its vertices are popped; finally all vertices are popped when the third component  $\{1, 2, 3, 4, 8, 9\}$  is found. Notice that there is no cross-edge to a vertex with a number less than 5 when the second component is discovered. Similarly in the first component, there is no edge to a vertex with a number less than 0. In the third component, there is no edge to a vertex less than 1 since we have set the serial number of vertex 0 to  $+\infty$  when 0 was popped.

73 an environment  $e$ . It calls *dfs1* on non-visited vertices in  $r$  and returns a pair consisting of an  
 74 integer and the modified environment. The integer is the minimum of the values computed  
 75 by *dfs1* on non-visited vertices in  $r$  and the serial numbers of already visited vertices in  $r$ . If  
 76 the set of roots is empty, the returned integer is  $+\infty$ .

77 The main procedure *tarjan* initializes the environment with an empty stack, an empty set  
 78 of sccs, the fresh serial number 0 and the constant function giving the number  $-1$  to each  
 79 vertex. The result is the set of components returned by the function *dfs* called on all vertices  
 80 in the graph.

```

81 let rec dfs1 x e =
82   let n0 = e.sn in
83   let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
84   if n1 < n0 then (n1, e1) else
85     let (s2, s3) = split x e1.stack in
86     (+∞, {stack = s3; sccs = add (elements s2) e1.sccs;
87           sn = e1.sn; num = set_infty s2 e1.num})
88
89 with dfs r e = if is_empty r then (+∞, e) else
90   let x = choose r in let r' = remove x r in
91   let (n1, e1) = if e.num[x] ≠ -1 then (e.num[x], e) else dfs1 x e in
92   let (n2, e2) = dfs r' e1 in (min n1 n2, e2)
93
94 let tarjan () =
95   let e = {stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
96   let (_, e') = dfs vertices e in e'.sccs
97
98
99
100
101
102
103
104
105
106
107
108
109
110 let add_stack_incr x e = let n = e.sn in
111   {stack = Cons x e.stack; sccs = e.sccs; sn = n+1; num = e.num[x ← n]}

```

99 In the body of *dfs1*, the auxiliary function *add\_stack\_incr* updates the environment by  
 100 pushing  $x$  on the stack, assigning it the current fresh serial number, and incrementing that  
 101 number in view of future calls. The function *dfs1* performs a recursive call to *dfs* for the  
 102 adjacent vertices of  $x$  as roots and the updated environment. If the returned integer value  $n1$   
 103 is less than the number  $n0$  assigned to  $x$ , the function simply returns  $n1$  and the current  
 104 environment. Otherwise, the function declares that a new scc has been found, consisting of  
 105 all vertices that are contained on top of  $x$  in the current stack. Therefore the stack is popped  
 106 until  $x$ ; the popped vertices are stored as a new set in  $e.sccs$ ; and their serial numbers are all  
 107 set to  $+\infty$ , ensuring that they do not interfere with future calculations of min values. The  
 108 auxiliary functions *split* and *set\_infty* are used to carry out these updates.

```

109
110 let add_stack_incr x e = let n = e.sn in
111   {stack = Cons x e.stack; sccs = e.sccs; sn = n+1; num = e.num[x ← n]}

```

## 4 Formal proofs of Tarjan's SCC algorithm

```
112 let rec set_infty s f = match s with Nil → f
113   | Cons x s' → (set_infty s' f)[x ← +∞] end
114 let rec split x s = match s with Nil → (Nil, Nil)
115   | Cons y s' → if x = y then (Cons x Nil, s')
116   else let (s1', s2) = split x s' in (Cons y s1', s2) end
```

118 Figure 1 illustrates the behavior of the algorithm by an example. We presented the  
119 algorithm as a functional program, using data structures available in the Why3 standard  
120 library [3]. For lists we have the constructors *Nil* and *Cons*; the function *elements* returns  
121 the set of elements of a list. For finite sets, we have the empty set *empty*, and the functions  
122 *add* to add an element to a set, *remove* to remove an element from a set, *choose* to pick an  
123 arbitrary element in a (non-empty) set, and *is\_empty* to test for emptiness. We also use  
124 maps with functions *const* denoting the constant function, *[\_]* to access the value of an  
125 element, and *[\_ ← \_]* for creating a map obtained from an existing map by setting an  
126 element to a given value.

127 For a correspondence between our presentation and the imperative programs used in  
128 standard textbooks, the reader is referred to [8]. The present version can be directly translated  
129 into COQ or Isabelle functions, and it respects the linear running time behaviour of the  
130 algorithm, since vertices could be easily implemented by integers,  $+\infty$  by the cardinal of  
131 *vertices*, finite sets by lists of integers and mappings by mutable arrays (see for instance [7]).

132 Thus for each environment *e* in the algorithm, the working stack *e.stack* corresponds to a  
133 cut of the spanning forest where strongly connected components to its left are pruned and  
134 stored in *e.sccs*. In this stack, any vertex can reach any vertex higher in the stack. And if a  
135 vertex is a base of an scc, no cross-edge can reach some vertex lower than this base in the  
136 stack, otherwise that last vertex would be in the same scc with a strictly lower serial number.

137 We therefore have to organize the proofs of the algorithm around these arguments. To  
138 maintain these invariants we will distinguish, as is common for DFS algorithms, three sets of  
139 vertices: white vertices are the non-visited ones, black vertices are those that are already  
140 fully visited, and gray vertices are those that are still being visited. Clearly, these sets are  
141 disjoint and white vertices can be considered as forming the complement in *vertices* of the  
142 union of the gray and black ones.

143 The previously mentioned invariant properties can now be expressed for vertices in the  
144 stack: no such vertex is white, any vertex can reach all vertices higher in the stack, any  
145 vertex can reach some gray vertex lower in the stack. Moreover, vertices in the stack respect  
146 the numbering order, i.e. a vertex *x* is lower than *y* in the stack if and only if the number  
147 assigned to *x* is strictly less than the number assigned to *y*.

### 148 3 The proof in Why3

149 The Why3 system comprises the programming language WhyML used in previous section and  
150 a many sorted first-order logic with inductive data types and inductive predicates to express  
151 the logical assertions. The system generates proof obligations w.r.t. the assertions, pre- and  
152 post-conditions and lemmas inserted in the WhyML program. The system is interfaced with  
153 off-the-shelf automatic provers and interactive proof assistants.

154 From the Why3 library, we use pre-defined theories for integer arithmetic, polymorphic  
155 lists, finite sets and mappings. There is also a small theory for paths in graphs. Here we  
156 define graphs, paths and sccs as follows.

```
157 axiom successors_vertices: ∀x. mem x vertices → subset (successors x) vertices
158 predicate edge (x y: vertex) = mem x vertices ∧ mem y (successors x)
159 inductive path vertex (list vertex) vertex =
```

```

161 | Path_empty:  $\forall x$ : vertex. path x Nil x
162 | Path_cons:  $\forall x y z$ : vertex, l: list vertex.
163   edge x y  $\rightarrow$  path y l z  $\rightarrow$  path x (Cons x l) z
164
165 predicate reachable (x y: vertex) =  $\exists l$ . path x l y
166 predicate in_same_scc (x y: vertex) = reachable x y  $\wedge$  reachable y x
167 predicate is_subsc (s: set vertex) =  $\forall x y$ . mem x s  $\rightarrow$  mem y s  $\rightarrow$  in_same_scc x y
168 predicate is_scc (s: set vertex) = not is_empty s
169    $\wedge$  is_subsc s  $\wedge$  ( $\forall s'$ . subset s s'  $\rightarrow$  is_subsc s'  $\rightarrow$  s == s')

```

171 where *mem* and *subset* denote membership and the subset relation for finite sets.

172 We add two ghost fields in environments for the black and gray sets of vertices. These  
 173 fields are used in the proofs but not used in the calculation of the sccs, which is checked by  
 174 the type-checker of the language.<sup>1</sup>

```

175 type env = {ghost black: set vertex; ghost gray: set vertex;
176   stack: list vertex; sccs: set (set vertex); sn: int; num: map vertex int}
177

```

179 The functions now become:

```

180 let rec dfs1 x e =
181   let n0 = e.sn in
182   let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
183   if n1 < n0 then (n1, add_black x e1) else
184     let (s2, s3) = split x e1.stack in
185     (+ $\infty$ , {black = add x e1.black; gray = e.gray; stack = s3;
186       sccs = add (elements s2) e1.sccs; sn = e1.sn; num = set_infty s2 e1.num})
187 with dfs r e = ... (* unmodified *)
188 let tarjan () =
189   let e = {black = empty; gray = empty;
190     stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
191   let (_, e') = dfs vertices e in e'.sccs
192

```

194 with a new function *add\_black* turning a vertex from gray to black and the modified  
 195 *add\_stack\_incr* adding a new gray vertex with a fresh serial number to the current stack.

```

196 let add_black x e =
197   {black = add x e.black; gray = remove x e.gray;
198   stack = e.stack; sccs = e.sccs; sn = e.sn; num = e.num}
199 let add_stack_incr x e =
200   let n = e.sn in
201   {black = e.black; gray = add x e.gray;
202   stack = Cons x e.stack; sccs = e.sccs; sn = n+1; num = e.num[x  $\leftarrow$  n]}
203

```

205 The main invariant ( $I$ ) of our program states that the environment is well-formed:

```

206 predicate wf_env (e: env) =
207   let {stack = s; black = b; gray = g} = e in
208   wf_color e  $\wedge$  wf_num e  $\wedge$  simplelist s  $\wedge$  no_black_to_white b g  $\wedge$ 
209   ( $\forall x y$ . lmem x s  $\rightarrow$  lmem y s  $\rightarrow$  e.num[x]  $\leq$  e.num[y]  $\rightarrow$  reachable x y)  $\wedge$ 
210   ( $\forall y$ . lmem y s  $\rightarrow$   $\exists x$ . mem x g  $\wedge$  e.num[x]  $\leq$  e.num[y]  $\wedge$  reachable y x)  $\wedge$ 
211   ( $\forall cc$ . mem cc e.sccs  $\leftrightarrow$  subset cc b  $\wedge$  is_scc cc)
212

```

214 where *lmem* stands for membership in a list. The well-formedness property is the conjunction  
 215 of seven clauses. The two first clauses express elementary conditions about the colored sets  
 216 of vertices and the numbering function (see [7, 8] for a detailed description). The third clause  
 217 states that there are no repetitions in the stack, and the fourth that there is no edge from a

<sup>1</sup> In Why3-1.2.0, this check is performed differently

## 6 Formal proofs of Tarjan's SCC algorithm

218 black vertex to a white vertex. The next two clauses formally express the property already  
 219 stated above: any vertex in the stack reaches all higher vertices and any vertex in the stack  
 220 can reach a lower gray vertex. The last clause states that the *sccs* field is the set of all sccs  
 221 all of whose vertices are black.

222 Since at the end of the *tarjan* function, all vertices are black, the *sccs* field will contain  
 223 exactly the set of all strongly connected components.

```
224 let tarjan () = returns {r → ∀cc. mem cc r ↔ subset cc vertices ∧ is_scc cc}
225   let e = {black = empty; gray = empty;
226     stack = Nil; sccs = empty; sn = 0; num = const (-1)} in
227   let (_, e') = dfs vertices e in assert {subset vertices e'.black};
228   e'.sccs
229
```

231 Our functions *dfs1* and *dfs* modify the environment in a monotonic way. Namely they  
 232 augment the set of visited vertices (the black ones); they keep invariant the set of the ones  
 233 currently under visit (the gray set); they increase the stack with new black vertices; they  
 234 also discover new sccs and they keep invariant the serial numbers of vertices in the stack,

```
235 predicate subenv (e e': env) =
236   subset e.black e'.black ∧ e.gray == e'.gray
237   ∧ (∃s. e'.stack = s ++ e.stack ∧ subset (elements s) e'.black)
238   ∧ subset e.sccs e'.sccs ∧ (∀x. lmem x e.stack → e.num[x] = e'.num[x])
239
```

241 Once these invariants are expressed, it remains to locate them in the program text and  
 242 to add assertions which help to prove them. The pre-conditions of *dfs1* are quite natural:  
 243 the vertex  $x$  must be a white vertex of the graph, and it must be reachable from all gray  
 244 vertices. Moreover invariant ( $\mathcal{I}$ ) must hold. The post-conditions of *dfs1* are of three kinds.  
 245 Firstly ( $\mathcal{I}$ ) and the monotony property *subenv* hold in the resulting environment. Vertex  
 246  $x$  is black at the end of *dfs1*. Finally we express properties of the integer value  $n$  returned  
 247 by this function which should be *LOWLINK*( $x$ ) as noted previously. In this proof, we give  
 248 three implicit properties for characterizing  $n$ . First, the returned value is never higher than  
 249 the number of  $x$  in the final environment. Secondly, the returned value is either  $+\infty$  or the  
 250 number of a vertex in the stack reachable from  $x$ . Finally, if there is an edge from a vertex  $y'$   
 251 in the new part of the stack to a vertex  $y$  in its old part, the resulting value  $n$  must be lower  
 252 or equal to the serial number of  $y$ .

```
253 let rec dfs1 x e =
254   (* pre-condition *)
255   requires {mem x vertices ∧ not mem x (union e.black e.gray)}
256   requires {∀y. mem y e.gray → reachable y x}
257   requires {wf_env e} (* I *)
258   (* post-condition *)
259   returns {(_, e') → wf_env e' ∧ subenv e e'}
260   returns {(_, e') → mem x e'.black}
261   returns {(n, e') → n ≤ e'.num[x]}
262   returns {(n, e') → n = +∞ ∨ num_of_reachable_in_stack n x e'}
263   returns {(n, e') → ∀y. xedge_to e'.stack e.stack y → n ≤ e'.num[y]}
264
265
```

266 The auxiliary predicates used above are formally defined in the following way.

```
267 predicate num_of_reachable_in_stack (n: int) (x: vertex)(e: env) =
268   ∃y. lmem y e.stack ∧ n = e.num[y] ∧ reachable x y
269 predicate xedge_to (s1 s3: list vertex) (y: vertex) =
270   (∃s2. s1 = s2 ++ s3 ∧ ∃y'. lmem y' s2 ∧ edge y' y) ∧ lmem y s3
271
```

273 Notice that the definition of *xedge\_to* fits the definition of *LOWLINK* when the cross edge  
 274 ends at a vertex residing in the stack before the call of *dfs1*. The pre- and post-conditions

275 for the function *dfs* are quite similar up to a generalization to sets of vertices considered as  
 276 the roots of the algorithm (see [7]).

277 We now add seven assertions in the body of the *dfs1* function to help the automatic  
 278 provers. In contrast, the function *dfs* needs no extra assertions in its body. In *dfs1*, when the  
 279 number  $n0$  of  $x$  is strictly greater than the number  $n1$  resulting from the call to its successors,  
 280 the first assertion states that  $n1$  cannot be  $+\infty$ ; it helps proving the next assertion. The  
 281 second assertion states that a lower gray vertex is reachable from  $x$  and that thus the scc of  
 282  $x$  is not fully black at end of *dfs1*. In that assertion the inequality  $y \neq x$  is redundant, but  
 283 helps showing the *sccs* constraint at the end of *dfs1*. When  $n1 \geq n0$ , the next four assertions  
 284 show that the strongly connected component *elements s2* of  $x$  is on top of  $x$  in the current  
 285 stack and that then  $x$  is the base of that scc. The seventh assertion helps proving that the  
 286 coloring constraint is preserved at the end of *dfs1*.

```

287 let n0 = e.sn in
288 let (n1, e1) = dfs (successors x) (add_stack_incr x e) in
289 if n1 < n0 then begin
290   assert {n1 ≠ +∞};
291   assert {∃y. y ≠ x ∧ mem y e1.gray ∧ e1.num[y] < e1.num[x] ∧ in_same_scc x y};
292   (n1, add_black x e1) end
293 else
294   let (s2, s3) = split x e1.stack in
295   assert {is_last x s2 ∧ s3 = e.stack ∧ subset (elements s2) (add x e1.black)};
296   assert {is_subsc (elements s2)};
297   assert {∀y. in_same_scc y x → lmem y s2};
298   assert {is_scc (elements s2)};
299   assert {inter e.gray (elements s2) == empty};
300   (+∞, {black = add x e1.black; gray = e.gray; stack = s3;
301     sccs = add (elements s2) e1.sccs; sn = e1.sn; num = set_infty s2 e1.num})
302

```

304 where *inter* is set intersection, and *is\_last* is defined below.

```

305 predicate is_last (x: α) (s: list α) = ∃s'. s = s' ++ Cons x Nil
306

```

308 All proofs are discovered by the automatic provers except for two proofs carried out  
 309 interactively in COQ. One is the proof of the black extension of the stack in case  $n1 < n0$ .  
 310 The provers could not work with the existential quantifier, although the COQ proof is quite  
 311 short. The second COQ proof is the fifth assertion in the body of *dfs1*, which asserts that any  
 312  $y$  in the scc of  $x$  belongs to *s2*. It is a maximality assertion which states that the set *elements*  
 313 *s2* is a complete scc. The proof of that assertion is by contradiction. If  $y$  is not in *s2*, there  
 314 must be an edge from  $x'$  in *s2* to some  $y'$  not in *s2* such that  $x$  reaches  $x'$  and  $y'$  reaches  $y$ .  
 315 There are three cases, depending on the position of  $y'$ . Case 1 is when  $y'$  is in *sccs*: this is  
 316 not possible since  $x$  would then be in *sccs* which contradicts  $x$  being gray. Case 2 is when  $y'$   
 317 is an element of *s3*: the serial number of  $y'$  is strictly less than the one of  $x$  which is  $n0$ . If  
 318  $x' \neq x$ , the cross-edge from  $x'$  to  $y'$  contradicts  $n1 \geq n0$  (post-condition 5); if  $x' = x$ , then  $y'$   
 319 is a successor of  $x$  and again it contradicts  $n1 \geq n0$  (post-condition 3). Case 3 is when  $y'$   
 320 is white, then  $x' \neq x$  is impossible since  $x'$  is then black in *s2* and would be the origin of a  
 321 black-to-white edge to  $y'$ ; if  $x' = x$ , then  $y'$  is not white by post-condition 2 of *dfs*.

322 Some quantitative information about the Why3 proof is listed in table 1. Alt-Ergo 2.3  
 323 and CVC4 1.5 proved the bulk of the proof obligations.<sup>2</sup> The proof uses 49 lemmas that were  
 324 all proved automatically, but with an interactive interface providing hints to apply inlining,  
 325 splitting, or induction strategies. This includes 13 lemmas on sets, 16 on lists, 5 on lists

<sup>2</sup> In addition to the results reported in the table, Spass was used to discharge one proof obligation.

## 8 Formal proofs of Tarjan’s SCC algorithm

provers	Alt-Ergo	CVC4	E-prover	Z3	#VC	#PO
49 lemmas	1.91	26.11	3.33		70	49
split	0.09	0.16			6	6
add_stack_incr	0.01				1	1
add_black	0.02				1	1
set_infty	0.03				1	1
dfs1	77.89	150.2	19.99	13.67	79	20
dfs	4.71	3.52		0.26	58	25
tarjan	0.85				15	5
total	85.51	179.99	23.32	13.93	231	108

■ **Table 1** Performance results with provers in Why3-0.88.3 (in seconds, on a 3.3 GHz Intel Core i5 processor). Total time is 341.15 seconds. The two last columns contain the numbers of verification conditions and proof obligations. Notice that there may be several VCs per proof obligation.

326 without repetitions, 3 on paths, 5 on sccs and 7 very specialized lemmas directly involved  
 327 in the proof obligations of the algorithm. Among the lemmas, a critical one is the lemma  
 328 *xpath\_xedge* on paths which reduces a predicate on paths to a predicate on edges. In fact,  
 329 most of the Why3 proof works on edges which are handled more robustly by the automatic  
 330 provers than paths. Another important lemma is *subsc\_after\_last\_gray* which shows that  
 331 the stack elements on top of the last gray vertex form a subset of an scc. This means that  
 332 another program with the *split* call before the if-statement would make a simpler proof, but  
 333 it would be a non-linear-time program. The two COQ proofs are only 9 and 81 lines long (the  
 334 COQ files of 677 and 680 lines include preambles that are automatically generated during  
 335 the translation from Why3 to COQ). The interested reader is referred to [7] where the full  
 336 proof is available.

337 The proof explained so far only showed the partial correctness of the algorithm. But after  
 338 adding two lemmas about union and difference for finite sets, termination is automatically  
 339 proved by the following lexicographic ordering on the number of white vertices and roots.

```
340 let rec dfs1 x e = variant{cardinal (diff vertices (union e.black e.gray)), 0}
341 with dfs r e = variant{cardinal (diff vertices (union e.black e.gray)), 1, cardinal r}
```

### 344 4 The proof in Coq

345 COQ is based on type theory and the calculus of constructions, a higher order lambda-calculus,  
 346 for expressing formulae and proofs. Some basic notions of graph theory are provided by  
 347 the Mathematical Components Library [18]. Our formalization is parameterized by a finite  
 348 type  $V$  for the vertices and the function *successors* such that *successors*  $x$  is the adjacency  
 349 set of any vertex  $x$ . The boolean *gconnect*  $x y$  indicates that a path connects the vertex  
 350  $x$  to the vertex  $y$ . It is straightforward to define the set *gscCs* of the sccs using *gconnect*.  
 351 Components are represented as sets of sets ( $\{set\ \{set\ V\}\}$ ). We use library operations for  
 352 creating singletons ( $\{set\ x\}$ ), taking unions ( $S_1 \cup S_2$ ), differences ( $S_1 \setminus S_2$ ), complements  
 353 ( $\sim : S$ ), and unions of all sets of a set of sets (*cover*  $S$ ).

354 COQ proposes several mechanisms to put together properties (boolean conjunction,  
 355 propositional conjunction, record, inductive family) that have their own specificities. In order  
 356 to make the presentation more readable for a non-COQ expert, we write them all with the  
 357 propositional conjunction  $[\wedge P_1, \dots \& P_n]$ . We refer to [9] for the actual code.

358 The COQ proof differs from the one in Why3: it uses natural numbers only and does  
 359 not mention colors (white, gray and black). In particular, the number  $\infty$  is defined as the  
 360 cardinality of  $V$ , vertices with  $\infty + 1$  as serial number correspond to the white vertices of the  
 361 previous section and the environment is defined as a record with only two fields, a set of sccs  
 362 and the mapping assigning serial numbers to vertices:

363



```
364 Record env := Env {escs : {set {set V}}; num : {ffun V → nat}}.
```

366 Given an environment  $e$ , the set of visited vertices is  $visited\ e$  (the vertices with serial number  
367 less or equal to  $\infty$ ), the current fresh serial number is  $sn\ e$  (the cardinal of visited vertices),  
368 and the stack is  $stack\ e$  (the list of elements  $x$  which satisfy  $num\ e\ x < sn\ e$ , sorted by  
369 increasing serial number).

370 Another difference with the Why3 algorithm is the disentanglement of the mutually  
371 recursive function *tarjan* into two separate functions. The first one *dfs1* treats a vertex  $x$  and  
372 the second one *dfs* a set of vertices *roots* in an environment  $e$ .

```
373 Definition dfs1 dfs x e :=
374   let: (n1, e1) as res := dfs (successors x) (visit x e) in
375   if n1 < sn e then res else (∞, store (stack e1 \ stack e) e1).
376
377 Definition dfs dfs1 dfs (roots : {set V}) e :=
378   if [pick x in roots] isn't Some x then (∞, e) else
379   let: (n1, e1) := if num e x ≤ ∞ then (num e x, e) else dfs1 x e in
380   let: (n2, e2) := dfs (roots \ [set x]) e1 in (minn n1 n2, e2).
381
382
```

383 where *visit*  $x\ e$  produces the environment where  $x$  gets the next serial number, *store* stores a  
384 new strongly connected component.

385 Then, the two functions are glued together in a recursive function *rec* where the parameter  
386  $k$  controls the maximal recursive height.

```
387 Fixpoint rec k r e := if k is k'.+1 then dfs (dfs1 (rec k')) (rec k') r e else (∞, e).
388
```

390 If  $k$  is not zero (i.e. it is a successor of some  $k'$ ), *rec* calls *dfs* taking care that its parameters  
391 can only use recursive calls to *rec* with a smaller recursive height, here  $k'$ . This ensures  
392 termination. A dummy value is returned in the case where  $k$  is zero. Finally, the top level  
393 *tarjan* calls *rec* with the proper initial arguments.

```
394 Definition tarjan := let: (_, e) := rec (∞ * ∞.+2) V (Env ∅ [ffun ⇒ ∞.+1]) in escs e.
395
396
```

397 Initially, the roots are all the vertices ( $V$ ) and the environment has no component and all  
398 vertices are not visited (their number is  $\infty.+1$ ). As both *dfs* and *dfs1* cannot be applied  
399 more than the number of vertices, the value  $\infty * \infty.+2$  encodes the lexicographic product  
400 of the two maximal heights. It gives *rec* enough fuel to never encounter the dummy value  
401 so *tarjan* correctly terminates the computation. This allows us to separate the proof of the  
402 termination from the algorithm itself, and this last statement is of course proved formally  
403 later and named *rec\_terminates*.

404 The invariants of the COQ proof are usually shorter than in the Why3 proof since they  
405 do not mention colors. We first define well-formed environments and their valid extension:

```
406 Definition wf_env e := [∧ escs e ⊆ gscs,
407   ∀ x, num e x < ∞ → num e x < sn e,
408   ∀ x, (num e x = ∞) = (x ∈ cover (escs e)) &
409   ∀ x y, num e x ≤ num e y < sn e → gconnect x y].
410
411 Definition subenv e1 e2 := [∧ escs e1 ⊆ escs e2,
412   ∀ x, num e1 x < ∞ → num e2 x = num e1 x & ∀ x, num e2 x < sn e1 → num e1 x < sn e1].
413
```

415 Then we state that new visited vertices are the ones reachable by paths accessible from roots  
416 with non-visited vertices (i.e. by white paths in the colored setting). The function *nexts* such  
417 that *nexts*  $D\ X$  returns the set of vertices reachable from the set  $X$  by a path which only  
418 contains vertices in  $D$  except maybe the last one.

```

419
420 Definition outenv (roots : {set V}) (e e' : env) := [∧
421   ∀ x y, x ∈ stack e' \ stack e → y ∈ stack e' \ stack e → gconnect x y,
422   ∀ x, x ∈ stack e' \ stack e → ∃ y, y ∈ stack e ∧ gconnect x y &
423   visited e' = visited e ∪ nexts (λ: visited e) roots ].
424

```

The post-condition is the conjunction of these three properties and the characterization of the output rank:

```

427 Definition dfs_spec (ne' : nat * env) (roots : {set V}) e := let: (n, e') := ne' in
428   [∧ n = \min_(x in nexts (λ: visited e) roots) inord (num e' x),
429     wf_env e', subenv e e' & outenv roots e e'].
430

```

Here, the argument  $ne'$  is the result of a *dfs*. The output rank  $n$  is the minimum of the serial numbers of the vertices which can be reached from the roots through a path where all the vertices except maybe the last one were not already visited. Note that this characterization differs from the notion of *LOWLINK* which requires that the last vertex was visited.

Finally, we express correctness as the implication between pre- and post-conditions:

```

437 Definition dfs_correct dfs (roots : {set V}) e := wf_env e →
438   (∀ x y, x ∈ stack e → y ∈ roots → gconnect x y) → dfs_spec (dfs roots e) roots e.
439 Definition dfs1_correct dfs1 x e := wf_env e → x ∉ visited e →
440   (∀ x y, x ∈ stack e → y ∈ [set x] → gconnect x y) → dfs_spec (dfs1 x e) [set x] e.
441

```

These invariants are expressed differently from the formulation in Why3, but they reflect essentially the same ideas. Rephrasing the invariants made it possible to reduce by approximately 50% the size of the Coq proofs. The two central theorems are:

```

446 Lemma dfsP dfs1 dfsrec (roots : {set V}) e : (∀ x, x ∈ roots → dfs1_correct dfs1 x e) →
447   (∀ x, x ∈ roots → ∀ e1, subenv e e1 → dfs_correct dfsrec (roots \ [set x]) e1) →
448   dfs_correct (dfs dfs1 dfsrec) roots e.
449
450 Lemma dfs1P dfs x e : dfs_correct dfs (successors x) (visit x e) →
451   dfs1_correct (dfs1 dfs) x e.
452

```

They state that *dfs* and *dfs1* are correct if their respective recursive calls are correct. The proof of the first lemma is straightforward since *dfs* simply iterates on a list. It mostly requires book-keeping between what is known and what needs to be proved. This is done in about 54 lines. The second one is more intricate and requires 124 lines. Gluing these two theorems together and proving termination gives us an extra 12 lines to prove the theorem

```

459 Theorem rec_terminates k (roots : {set V}) e :
460   k ≥ #|λ: visited e| * ∞.+1 + #|roots| → dfs_correct (rec k) roots e.
461

```

The correctness of *tarjan* follows directly in 19 lines of straightforward proof.

```

464 Theorem tarjan_correct : tarjan = gsccs.
465

```

We now provide some quantitative information. The COQ contribution is composed of two files. The *extra\_nocolors* file defines the *bigmin* operator and some notions of graph theory that we intend to add to Mathematical Components. This file is 294 lines long. The main file is *tarjan\_nocolors* and is 605 lines long. It is compiled in 12 seconds with a memory footprint of 800 Mb (3/4 of which is resident) on a Intel® i7 2.60GHz quad-core laptop running Linux. The proofs are performed in the SSREFLECT proof language [14] with very little automation. The proof script is mostly procedural, alternating book-keeping tactics (*move*) with transformational ones (mostly *rewrite* and *apply*), but often intermediate steps

Number of lines	1	2	3	4	5	6	11	12	16	19	54	124
Number of proofs	19	7	5	2	1	2	2	1	1	1	1	1

■ **Table 2** Distribution of the numbers of lines of the 43 proofs in the file *tarjan\_nocolors*.

475 are explicitly declared with the *have* tactic. There are more than fifty of such intermediate  
 476 steps in the 320 lines of proof of the file *tarjan\_nocolors*. Table 2 gives the distribution of  
 477 the numbers of lines of these proofs. Most of them are very short (26 are less than 2 lines)  
 478 and the only complicated proof is the one corresponding to the lemma *dfs1P*.

## 479 5 The proof in Isabelle/HOL

480 Isabelle/HOL [21] is the encoding of simply typed higher-order logic in the logical framework  
 481 Isabelle [23]. Unlike Why3, it is not primarily intended as an environment for program  
 482 verification and does not contain specific syntax for stating pre- and post-conditions or  
 483 intermediate assertions in function definitions. Logics and formalisms for program verification  
 484 have been developed within Isabelle/HOL (e.g., [16]), but they target imperative rather  
 485 than functional programming, so we simply formalize the algorithm as an Isabelle function.  
 486 Isabelle/HOL provides an extensive library of data structures and proofs. In this development  
 487 we mainly rely on the set and list libraries. We start by introducing a *locale*, fixing parameters  
 488 and assumptions for the remainder of the proof. We explicitly assume that the set of vertices  
 489 is finite.

```
490 locale graph =
491   fixes vertices ::  $\nu$  set and successors ::  $\nu \Rightarrow \nu$  set
492   assumes finite vertices and  $\forall v \in \text{vertices}. \text{successors } v \subseteq \text{vertices}$ 
493
```

495 We introduce reachability in graphs using an inductive predicate definition, rather than via  
 496 an explicit reference to paths as in the Why3 definition. Isabelle then generates appropriate  
 497 induction theorems for use in proofs.

```
498 inductive reachable where
499   reachable x x
500 |  $\llbracket y \in \text{successors } x; \text{reachable } y \ z \rrbracket \implies \text{reachable } x \ z$ 
501
```

503 The definition of strongly connected components mirrors that used in Why3. The follow-  
 504 ing lemma states that SCCs are disjoint; its one-line proof is found automatically using  
 505 *Sledgehammer* [2], which heuristically selects suitable lemmas from the set of available facts  
 506 (including Isabelle’s library), invokes several automatic provers, and finally reconstructs a  
 507 proof that is checked by the Isabelle kernel.

```
508 lemma scc-partition:
509   assumes is-scc S and is-scc S' and  $x \in S \cap S'$ 
510   shows  $S = S'$ 
511
```

513 Environments are represented by records, similar to the formalization in Why3, except  
 514 that there is no distinction between regular and “ghost” fields. Also, the definition of the  
 515 well-formedness predicate closely mirrors that used in Why3.<sup>3</sup>

```
516 record  $\nu$  env =
517   black ::  $\nu$  set      gray ::  $\nu$  set
518   stack ::  $\nu$  list    sccs ::  $\nu$  set set    sn :: nat    num ::  $\nu \Rightarrow \text{int}$ 
519 definition wf_env where wf_env e  $\equiv$ 
520   wf_color e  $\wedge$  wf_num e  $\wedge$  distinct (stack e)  $\wedge$  no_black_to_white e
521
```

<sup>3</sup> We use the infix operator  $\preceq$  to denote precedence in lists.

## 12 Formal proofs of Tarjan's SCC algorithm

```

522  $\wedge (\forall x y. y \preceq x \text{ in } (\text{stack } e) \longrightarrow \text{reachable } x y)$ 
523  $\wedge (\forall y \in \text{set } (\text{stack } e). \exists g \in \text{gray } e. y \preceq g \text{ in } (\text{stack } e) \wedge \text{reachable } y g)$ 
524  $\wedge \text{sccs } e = \{ C . C \subseteq \text{black } e \wedge \text{is\_scc } C \}$ 

```

526 The definition of the two mutually recursive functions *dfs1* and *dfs* again closely follows their  
527 representation in Why3.

```

528 function (domintros) dfs1 and dfs where
529
530 dfs1 x e =
531   (let (n1,e1) = dfs (successors x) (add_stack_incr x e) in
532     if n1 < int (sn e) then (n1, add_black x e1)
533     else (let (l,r) = split_list x (stack e1) in
534       ( $+\infty$ , ( $\lfloor$  black = insert x (black e1), gray = gray e,
535         stack = r, sn = sn e1, sccs = insert (set l) (sccs e1),
536         num = set_infty l (num e1)  $\rfloor$  ))) and
537   dfs roots e =
538     (if roots = {} then ( $+\infty$ , e)
539     else (let x = SOME x. x  $\in$  roots;
540       res1 = (if num e x  $\neq$  -1 then (num e x, e) else dfs1 x e);
541       res2 = dfs (roots - {x}) (snd res1)
542       in (min (fst res1) (fst res2), snd res2))

```

544 The **function** keyword introduces the definition of a recursive function. Isabelle checks that  
545 the definition is well-formed and generates appropriate simplification and induction theorems.  
546 Because HOL is a logic of total functions, it introduces two proof obligations: the first one  
547 requires the user to prove that the cases in the function definitions cover all type-correct  
548 arguments; this holds trivially for the above definitions. The second obligation requires  
549 exhibiting a well-founded ordering on the function parameters that ensures the termination  
550 of recursive function invocations, and Isabelle provides a number of heuristics that work in  
551 many cases. However, the functions defined above will in fact not terminate for arbitrary  
552 calls, in particular for environments that assign sequence number  $-1$  to non-white vertices.  
553 The *domintros* attribute instructs Isabelle to consider these functions as “partial”. More  
554 precisely, it introduces an explicit predicate representing the domains for which the functions  
555 are defined. This “domain condition” appears as a hypothesis in the simplification rules  
556 that mirror the function definitions so that the user can assert the equality of the left- and  
557 right-hand sides of the definitions only if the domain predicate holds. Isabelle also proves  
558 (mutually inductive) rules for proving when the domain condition is guaranteed to hold. Our  
559 first objective is therefore to establish sufficient conditions that ensure the termination of the  
560 two functions. Assuming the domain condition, we prove that the functions never decrease  
561 the set of colored vertices and that vertices are never explicitly assigned the number  $-1$  by  
562 our functions. Denoting the union of gray and black vertices as *colored*, we introduce the  
563 predicate

```

564 definition colored_num where colored_num e  $\equiv$ 
565  $\forall v \in \text{colored } e. v \in \text{vertices} \wedge \text{num } e v \neq -1$ 
566
567

```

568 and show that this predicate is an invariant of the functions. We then prove that the triple  
569 defined as

```

570 (vertices - colored e, {x}, 1)
571 (vertices - colored e, roots, 2)
572

```

574 for the arguments of *dfs1* and *dfs*, respectively, decreases w.r.t. lexicographical ordering on  
575 finite subset inclusion and  $<$  on natural numbers across recursive function calls, provided  
576 that *colored\_num* holds when the function is called and *x* is a white vertex. These conditions

577 are therefore sufficient to ensure that the domain condition holds:<sup>4</sup>

```
578 theorem dfs1_dfs_termination:
579    $\llbracket x \in \text{vertices} - \text{colored } e; \text{colored\_num } e \rrbracket \implies \text{dfs1\_dfs\_dom } (\text{Inl}(x,e))$ 
580    $\llbracket \text{roots} \subseteq \text{vertices}; \text{colored\_num } e \rrbracket \implies \text{dfs1\_dfs\_dom } (\text{Inr}(\text{roots},e))$ 
581
```

583 The proof of partial correctness follows the same ideas as the proof presented for Why3.  
584 We define the pre- and post-conditions of the two functions as predicates in Isabelle. For  
585 example, the predicates for *dfs1* are defined as follows:

```
586 definition dfs1_pre where dfs1_pre e  $\equiv$ 
587   wf_env e  $\wedge$  x  $\in$  vertices  $\wedge$  x  $\notin$  colored e  $\wedge$  ( $\forall g \in \text{gray } e. \text{reachable } g \ x$ )
588 definition dfs1_post where dfs1_post x e res  $\equiv$ 
589   let n = fst res; e' = snd res
590   in wf_env e'  $\wedge$  subenv e e'  $\wedge$  roots  $\subseteq$  colored e'
591      $\wedge$  ( $\forall x \in \text{roots}. n \leq \text{num } e' \ x$ )
592      $\wedge$  (n =  $+\infty$   $\vee$  ( $\exists x \in \text{roots}. \exists y \text{ in set } (\text{stack } e'). \text{num } e' \ y = n \wedge \text{reachable } x \ y$ ))
593
```

595 We now show the following theorems:

- 596 ■ The pre-condition of each function establishes the pre-condition of every recursive call  
597 appearing in the body of that function. For the second recursive call in the body of *dfs*  
598 we also assume the post-condition of the first recursive call.
- 599 ■ The pre-condition of each function, plus the post-conditions of each recursive call in the  
600 body of that function, establishes the post-condition of the function.

601 Combining these results, we establish partial correctness:

```
602 theorem dfs_partial_correct:
603    $\llbracket \text{dfs1\_dfs\_dom } (\text{Inl}(x,e)); \text{dfs1\_pre } x \ e \rrbracket \implies \text{dfs1\_post } x \ e \ (\text{dfs1 } x \ e)$ 
604    $\llbracket \text{dfs1\_dfs\_dom } (\text{Inr}(\text{roots},e)); \text{dfs\_pre } \text{roots } e \rrbracket \implies \text{dfs\_post } \text{roots } e \ (\text{dfs } \text{roots } e)$ 
605
```

607 We define the initial environment and the overall function.

```
608 definition init_env where init_env  $\equiv$ 
609   ( $\lfloor$  black = {}, gray = {}, stack = [], sccs = {}, sn = 0, num =  $\lambda_. -1$   $\rfloor$ )
610 definition tarjan where tarjan  $\equiv$ 
611   sccs (snd (dfs vertices init_env))
612
```

614 It is trivial to show that the arguments to the call of *dfs* in the definition of *tarjan* satisfy  
615 the pre-condition of *dfs*. Putting together the theorems establishing termination and partial  
616 correctness, we obtain the desired total correctness results.

```
617 theorem dfs_correct:
618   dfs1_pre x e  $\implies$  dfs1_post x e (dfs1 x e)
619   dfs_pre roots e  $\implies$  dfs_post roots e (dfs roots e)
620 theorem tarjan_correct:
621   tarjan = { C . is_scc C  $\wedge$  C  $\subseteq$  vertices }
622
```

624 The intermediate assertions appearing in the Why3 code guided the overall proof: they  
625 are established either as separate lemmas or as intermediate steps within the proofs of the  
626 above theorems. Similarly to the COQ proof, the overall induction proof was explicitly  
627 decomposed into individual lemmas as laid out above. In particular, whereas Why3 identifies  
628 the predicates that can be used from the function code and its annotation with pre- and  
629 post-conditions, these assertions appear explicitly in the intermediate lemmas used in the

<sup>4</sup> Observe that Isabelle introduces a single operator corresponding to the two mutually recursive functions whose domain is the disjoint sum of the domains of both functions.

$i = 1$	$i \leq 5$	$i \leq 10$	$i \leq 20$	$i \leq 30$	$i = 35$	$i = 43$	$i = 48$
28	8	4	1	2	1	1	1

■ **Table 3** Distribution of interactions in the Isabelle proofs.

630 proof of theorem *dfs\_partial\_correct*. The induction rules that Isabelle generated from the  
 631 function definitions were helpful for finding the appropriate decomposition of the overall  
 632 correctness proof.

633 Despite the extensive use of *Sledgehammer* for invoking automatic back-end provers,  
 634 including the SMT solvers CVC4 and Z3, from Isabelle, we found that in comparison to Why3,  
 635 significantly more user interactions were necessary in order to guide the proof. Although  
 636 many of those were straightforward, a few required thinking about how a given assertion  
 637 could be derived from the facts available in the context. Table 3 indicates the distribution  
 638 of the number of interactions used for the proofs of the 46 lemmas the theory contains.  
 639 These numbers cannot be compared directly to those shown in Table 2 for the COQ proof  
 640 because an Isabelle interaction is typically much coarser-grained than a line in a COQ proof.  
 641 As in the case of Why3 and COQ, the proofs of partial correctness of *dfs1* (split into two  
 642 lemmas following the case distinction) and of *dfs* required the most effort. It took about one  
 643 person-month to carry out the case study, starting from an initial version of the Why3 proof.  
 644 Processing the entire Isabelle theory on a laptop with a 2.7 GHz Intel® Core i5 (dual-core)  
 645 processor and 8 GB of RAM takes 35 seconds of CPU time.

## 646 6 General comments about the proof

647 Our formal proofs refer to colors, finite sets, and the stack, although the informal correctness  
 648 argument is about properties of strongly connected components in spanning trees. The  
 649 algorithmician would explain the algorithm with spanning trees as in Tarjan’s article. It  
 650 would be nice to extract a program from such a proof, but programmers like to understand  
 651 the proof in terms of variables and data that their program is using.

652 A first version of the formal proof used *ranks* in the working stack and a flat representation  
 653 of environments by adding extra arguments to functions for the black, gray, scc sets and the  
 654 stack. That was perfect for the automatic provers of Why3. But after remodelling the proof  
 655 in COQ and Isabelle/HOL, it was simpler to gather these extra arguments in records and  
 656 have a single extra argument for environments. Also *ranks* disappeared in favor of the *num*  
 657 function and the precedence relation, which are easier to understand. The automatic provers  
 658 have more difficulties with the inlining of environments, but with a few hints they could still  
 659 succeed.

660 Our proof is mainly about the correctness of Tarjan’s algorithm. It relies on surprisingly  
 661 few and elementary concepts of finite graphs. With the exception of the use of the Mathem-  
 662 atical Components library for COQ, we therefore did not use existing libraries formalizing  
 663 advanced concepts of graph theory [11, 22].

664 Finally, coloring of vertices is usual for graph algorithms. The stack used in our algorithm  
 665 is also not necessary since it is just used to efficiently output new strongly connected  
 666 components. The COQ formalization actually shows that proof can be done with just serial  
 667 numbers and the store of connected components. The stack and current serial number could  
 668 be added back using a program refinement, in order to recover a linear time computation.

669 There is always a tension between the concision of the proof, its clarity and its relation  
 670 to the real program. In our presentation, we have allowed for a few redundancies.

	Why3	Coq	Isabelle/HOL
expressivity	-	+	+
readability	+	-	+
stability	-	+	+
ease of use	-	-	-
automation	+	-	+
ignore termination	+	-	-
trusted base	-	+	+
automatic proof line-count	395	0	314 ui
manual proof line-count	90	898	1690

■ **Table 4** Compared usage of the three formal systems in the case of our three proofs

## 7 Conclusion

671

672 The formal proof expressed in this article was initially designed and implemented in Why3 [8]  
 673 as the result of a long process, nearly a 2-year half-time work with many attempts of  
 674 proofs about various graph algorithms (depth first search, Kosaraju strong connectivity,  
 675 bi-connectivity, articulation points, minimum spanning tree). Why3 has a clear separation  
 676 between programs and the logic. It makes the correctness proof quite readable for a  
 677 programmer. Also first-order logic is easy to understand. Moreover, one can prove partial  
 678 correctness without caring about termination.

679 Another important feature of Why3 is its interface with various off-the-shelf theorem  
 680 provers (mainly SMT provers). Thus the system benefits from the current technology in  
 681 theorem provers and clerical sub-goals can be delegated to these provers, which makes the  
 682 overall proof shorter and easier to understand. Although the proof must be split in more  
 683 elementary pieces, this has the benefit of improving its readability. Several hints about  
 684 inlining or induction reasoning are still needed and two COQ proofs were used. The system  
 685 records sessions and facilitates incremental proofs. However, the automatic provers are  
 686 sometimes no longer able to handle a proof obligation after seemingly minor modifications to  
 687 the formulation of the algorithm or the predicates, making the proof somewhat unstable.

688 The COQ and Isabelle proofs were inspired by the Why3 proof. Their development  
 689 therefore required much less time although their text is longer. The COQ proof uses  
 690 SSREFLECT and the Mathematical Components library, which helps reduce the size of the  
 691 proof compared to classical COQ. The proof also uses the bigops library and several other  
 692 higher-order features which makes it more abstract and closer to Tarjan’s original proof.

693 In COQ, one could prove termination using well-foundedness [1, 4], but because of nested  
 694 recursion the `Function` command fails, and both `Equations` and `Program Fixpoint` require the  
 695 addition of an extra proof argument to the function. Instead, we define the functionals `dfs1`  
 696 and `dfs` and recombine them in `rec` and `tarjan` by recursion on a natural number used as fuel.  
 697 We prove partial correctness on functionals and postpone termination on `rec`.

698 Our COQ proof does not use significant automation.<sup>5</sup> All details are explicitly expressed,  
 699 but many of them were already present in the Mathematical Components library. Moreover,  
 700 a proof certificate is produced and a functional program could in principle be extracted. The  
 701 absence of automation makes the system very stable to use since the proof script is explicit,  
 702 but it requires a higher degree of expertise from the user.

703 The Isabelle/HOL proof can be seen as a mid-point between the Why3 and COQ proofs.  
 704 It uses higher order logic and the level of abstraction is close to the one of the COQ proof,

<sup>5</sup> Hammers exist for Coq [10, 12] but unfortunately they currently perform badly when used in conjunction with the Mathematical Components library.

705 although more readable in this case study. The proof makes use of Isabelle's extensive support  
 706 for automation. In particular, *Sledgehammer* [2] was very useful for finding individual proof  
 707 steps. It heuristically selects lemmas and facts available in the context and then calls  
 708 automatic provers (SMT solvers and superposition-based provers for first-order logic). When  
 709 one of these provers finds a proof, Sledgehammer attempts to find a proof that can be  
 710 certified by the Isabelle kernel, using various proof methods such as combinations of rewriting  
 711 and first-order reasoning (blast, fastforce etc.), calls to the *metis* prover or reconstruction of  
 712 SMT proofs through the *smt* proof method. Unlike in Why3, the automatic provers used to  
 713 find the initial proof are not part of the trusted code base because ultimately the proof is  
 714 checked by the kernel. The price to pay is that the degree of automation in Isabelle is still  
 715 significantly lower compared to Why3. Adapting the proof to modified definitions was fast:  
 716 the Isabelle/jEdit GUI eagerly processes the proof script and quickly indicates those steps  
 717 that require attention.

718 The Isabelle proof also faces the termination problem to achieve general consistency.  
 719 We chose to delay handling termination, using the *domintros* attribute. The proofs of  
 720 termination and of partial correctness are independent; in particular, we obtain a weaker  
 721 predicate ensuring termination than the one used for partial correctness. Although the basic  
 722 principle of the termination proof is very similar to the COQ proof and relies on considering  
 723 functionals of which the recursive functions are fixpoints, the technical formulation is more  
 724 flexible because we rely on proving well-foundedness of an appropriate relation rather than  
 725 computing an explicit upper bound on the number of recursive calls.

726 One strong point of Isabelle/HOL is its nice L<sup>A</sup>T<sub>E</sub>X output and the flexibility of its parser,  
 727 supporting mathematical symbols. Combined with the hierarchical Isar proof language [31],  
 728 the proof is in principle understandable without actually running the system, although some  
 729 familiarity with the system is still required.

730 In the end, the three systems Why3, COQ, and Isabelle/HOL are mature, and each one  
 731 has its own advantages w.r.t. readability, expressivity, stability, ease of use, automation,  
 732 partial-correctness, code extraction, trusted base and length of proof (see table 4). Coming  
 733 up with invariants that are both strong enough and understandable was by far the hardest  
 734 part in this work. This effort requires creativity and understanding, although proof assistants  
 735 provide some help: missing predicates can be discovered by understanding which parts of  
 736 the proof fail. We think that formalizing the proof in all three systems was very rewarding  
 737 and helped us better understand the state of the art in computer-aided deductive program  
 738 verification. It could be also interesting to implement this proof in other formal systems and  
 739 establish comparisons based on this quite challenging example.<sup>6</sup>

740 Another interesting work would be to verify an implementation of this algorithm with  
 741 imperative programs and concrete data structures. This will make the proof more complex,  
 742 since mutable variables and mutable data structures have to be considered. There is support  
 743 for verifying imperative programs in general-purpose proof assistants [5, 6, 16], and it would be  
 744 interesting to also develop them simultaneously in various formal systems and to understand  
 745 how these proofs can be derived from ours.

746 A final and totally different remark is about teaching of algorithms. Do we want students to  
 747 formally prove algorithms, or to present algorithms with assertions, pre- and post-conditions,  
 748 and make them prove these assertions informally as exercises? In both cases, we believe that  
 749 our work could make a useful contribution.

---

<sup>6</sup> We have set up a Web page <http://www-sop.inria.fr/marelle/Tarjan/contributions.html> in order to collect formalizations.



750 — **References** —

- 751 1 G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive  
752 functions: a practical tool for the coq proof assistant. In *Functional and Logic Programming*  
753 *Systems (FLOPS'06)*, volume 3945 of *LNCS*, pages 114–129, Fuji Susono, Japan, April 2006.
- 754 2 Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgeham-  
755 mer with SMT solvers. *J. Automated Reasoning*, 51(1):109–128, 2013.
- 756 3 François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei  
757 Paskevich. *The Why3 platform, version 0.86.1*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay,  
758 version 0.86.1 edition, May 2015. Available at [why3.lri.fr/download/manual-0.86.1.pdf](http://why3.lri.fr/download/manual-0.86.1.pdf).
- 759 4 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive  
760 theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88,  
761 2016.
- 762 5 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs.  
763 In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proc. 16th ACM*  
764 *SIGPLAN Intl. Conf. Functional Programming*, pages 418–430, Tokyo, Japan, 2011. ACM.
- 765 6 Arthur Charguéraud. Higher-order representation predicates in separation logic. In *Proc. 5th*  
766 *ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP 2016, pages 3–14, New York, NY,  
767 USA, 2016. ACM.
- 768 7 Ran Chen and Jean-Jacques Lévy. Full scripts of Tarjan SCC Why3 proof. Technical report,  
769 Iscas and Inria, 2017. [jeanjacqueslevy.net/why3](http://jeanjacqueslevy.net/why3).
- 770 8 Ran Chen and Jean-Jacques Lévy. A semi-automatic proof of strong connectivity. In Andrei  
771 Paskevich and Thomas Wies, editors, *Proc. 9th Working Conf. Verified Software: Theories,*  
772 *Tools, and Experiments (VSTTE 2017)*, volume 10712 of *Lecture Notes in Computer Science*,  
773 pages 49–65. Springer, 2017.
- 774 9 Cyril Cohen and Laurent Théry. Full script of Tarjan SCC Coq/ssreflect proof, 2017. Available  
775 at <https://www-sop.inria.fr/marelle/Tarjan/>.
- 776 10 Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory.  
777 *J. Autom. Reasoning*, 61(1-4):423–453, 2018.
- 778 11 Christian Doczkal, Guillaume Combette, and Damien Pous. A formal proof of the minor-  
779 exclusion property for treewidth-two graphs. In Jeremy Avigad and Assia Mahboubi, editors,  
780 *Proc. 9th Intl. Conf. Interactive Theorem Proving (ITP 2018)*, volume 10895 of *LNCS*, pages  
781 178–195. Springer, 2018.
- 782 12 Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds,  
783 and Clark W. Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *CAV*  
784 *(2)*, volume 10427 of *LNCS*, pages 126–133. Springer, 2017.
- 785 13 Jean-Christophe Filliâtre et al. The Why3 gallery of verified programs. Technical report,  
786 CNRS, Inria, U. Paris-Sud, 2015. [toccata.lri.fr/gallery/why3.en.html](http://toccata.lri.fr/gallery/why3.en.html).
- 787 14 Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *J.*  
788 *Formalized Reasoning*, 3(2):95–152, 2010.
- 789 15 Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. In *Proc.*  
790 *40th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages*, POPL '13,  
791 pages 523–536, New York, NY, USA, 2013. ACM.
- 792 16 Peter Lammich. Refinement to Imperative/HOL. In Christian Urban and Xingyuan Zhang,  
793 editors, *Proc. 6th Intl. Conf. Interactive Theorem Proving (ITP 2015)*, volume 9236 of *LNCS*,  
794 pages 253–269, Nanjing, China, 2015. Springer.
- 795 17 Peter Lammich and René Neumann. A framework for verifying depth-first search algorithms.  
796 In *Proc. 4th ACM SIGPLAN Conf. Certified Programs and Proofs*, CPP '15, pages 137–146,  
797 New York, NY, USA, 2015. ACM.
- 798 18 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Available at: [https://](https://math-comp.github.io/mcb/)  
799 [math-comp.github.io/mcb/](https://math-comp.github.io/mcb/), 2016.
- 800 19 Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In *CADE*,  
801 2003.

- 802 **20** Stephan Merz. Isabelle formalization of Tarjan's algorithm, 2018. Available at [https://](https://members.loria.fr/SMerz/papers/cpp2019.html)  
803 [members.loria.fr/SMerz/papers/cpp2019.html](https://members.loria.fr/SMerz/papers/cpp2019.html).
- 804 **21** Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for*  
805 *Higher-Order Logic*. Number 2283 in Lecture Notes in Computer Science. Springer Verlag,  
806 2002.
- 807 **22** Lars Noschinski. A graph library for isabelle. *Mathematics in Computer Science*, 9(1):23–39,  
808 2015.
- 809 **23** Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in*  
810 *Computer Science*. Springer Verlag, 1994.
- 811 **24** Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Funda-*  
812 *menta Informaticae*, 118(1-2):135–175, 2012.
- 813 **25** François Pottier. Depth-first search and strong connectivity in Coq. In *Journées Francophones*  
814 *des Langages Applicatifs (JFLA 2015)*, January 2015.
- 815 **26** Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying concurrent graph  
816 algorithms. In Atsushi Igarashi, editor, *Proc. 14th Asian Symp. Programming Languages*  
817 *and Systems (APLAS 2016)*, volume 10017 of *LNCS*, pages 314–334, Hanoi, Vietnam, 2016.  
818 Springer.
- 819 **27** Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-  
820 grained concurrent programs. In *Proc. 36th ACM SIGPLAN Conf. Programming Language*  
821 *Design and Implementation, PLDI '15*, pages 77–87, New York, NY, USA, 2015. ACM.
- 822 **28** Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*,  
823 1972.
- 824 **29** Laurent Théry. Formally-proven Kosaraju's algorithm. Inria report, Hal-01095533, 2015.
- 825 **30** Ingo Wengener. A simplified correctness proof for a well-known algorithm computing strongly  
826 connected components. *Information Processing Letters*, 83(1):17–19, 2002.
- 827 **31** Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In  
828 Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry,  
829 editors, *12th Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLS'99)*, volume 1690  
830 of *LNCS*, pages 167–184, Nice, France, 1999. Springer.
- 831 **32** Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, Berlin,  
832 Heidelberg, 2006.