A Framework for Verifying Depth-First Search Algorithms

Peter Lammich René Neumann
Technische Universität München
{lammich,neumannr}@in.tum.de

Abstract

Many graph algorithms are based on depth-first search (DFS). The formalizations of such algorithms typically share many common ideas. In this paper, we summarize these ideas into a framework in Isabelle/HOL.

Building on the Isabelle Refinement Framework, we provide support for a refinement based development of DFS based algorithms, from phrasing and proving correct the abstract algorithm, over choosing an adequate implementation style (e. g., recursive, tail-recursive), to creating an executable algorithm that uses efficient data structures.

As a case study, we verify DFS based algorithms of different complexity, from a simple cyclicity checker, over a safety property model checker, to complex algorithms like nested DFS and Tarjan's SCC algorithm.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness Proofs

Keywords Graph algorithms; interactive theorem proving; Isabelle/HOL; refinement proof

1. Motivation

Algorithms based on depth-first search (DFS) are widespread. They range from simple ones, like cyclicity checking and safety property model checking, to more complicated ones such as nested DFS [3, 6, 16], and Tarjan's algorithm for computing the set of strongly connected components (SCCs) [17]. In our verified LTL-model checker CAVA [4] we find multiple DFS-algorithms side-by-side: Nested DFS for counter example search, SCC-algorithms for counter example search and optimization of Büchi-automata, and graph search for counter example reconstruction.

Despite their common base, a lot of duplicated effort is involved in formalizing and verifying them, due to their ad hoc formalizations of DFS. The goal of this paper is to provide a framework that supports the algorithm developer in all phases of the (refinement based) development process, from the correctness proof of the abstract algorithm to generation of verified, efficiently executable code. In summary, we want to make the verification of simple DFS-based algorithms almost trivial, and greatly reduce the effort for complex algorithms.

CPP '15, January 13–14, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693165

2. Introduction

Depth-first search is one of the basic algorithms of graph theory. It traverses the graph as long as possible (i. e., until there are no more non-visited successors left) along a branch before tracking back. As mentioned in the previous section, it is the base of a multitude of graph and automata algorithms. In this paper, we present a framework in Isabelle/HOL [13] for modeling and verification of DFS based algorithms, including the generation of efficiently executable code.

The framework follows a parametrization approach: We model a general DFS algorithm with extension points. An actual algorithm is defined by specifying functions to *hook* into those extension points. These *hook functions* are invoked whenever the control flow reaches the corresponding extension point. The hook functions work on an opaque extension state, which is independent of the state of the base DFS algorithm.

Properties of the algorithm are stated by invariants of the search state. To establish new invariants, one only has to show that they are preserved by the hook functions. Moreover, our framework supports an incremental approach, i. e., upon establishing a new invariant, already established invariants may be assumed. This modularizes the proofs, as it is not necessary to specify one large invariant.

Our framework features a refinement based approach, exploiting the general concepts provided by the Isabelle Refinement Framework [11]: First, an abstract algorithm is specified and proven correct. Next, the abstract algorithm is refined towards an efficient implementation, possibly in many steps. Refinement is done in a correctness preserving way, such that one eventually gets correctness of the implementation. The refinement based approach introduces a separation of concerns: The abstract algorithm may focus on the algorithmic idea, while the refinements focus on how this idea is efficiently implemented. This greatly simplifies the proofs, and makes verification of more complex algorithms manageable in the first place.

On the abstract level, we provide a very detailed base state, containing the search stack, timing information of the nodes, and sets of visited back, cross, and tree edges. On this detailed state, we provide a large library of standard invariants, which are independent of the extension state, and thus can be re-used for all correctness proofs.

For refinement, we distinguish two orthogonal issues: Structural refinement concerns the overall structure of the algorithm. To this end, our framework currently supports recursive and tail-recursive implementations. Data refinement concerns the representation of the state. It allows to refine to a concrete state with its content tailored towards the specific requirements of the parametrization. This includes projecting away parts of the state that are not needed by the actual algorithm, as well as representing the state by efficient

data structures. Here, our framework supports some commonly used projections of the base state, and an integration with the Autoref-Tool [7]. This tool synthesizes a refined algorithm that uses efficient executable data structures provided by the Isabelle Collections Framework [10], and can be exported to executable code by the code generator of Isabelle/HOL [5].

The framework comes bundled with multiple instantiations, which mostly stem from requirements of the CAVA model checker [4]. We provide implementations for a cyclicity checker, a safety property model checker, the nested DFS variant of [16] and Tarjan's SCC algorithm.

The whole formalization is available online at `http://cava.in.tum.de/CPP15`.

Structure The structure of the paper follows roughly the layout from above. We start with an overview about related work in Section 3. In Section 4 we describe the generic framework of the parametrized DFS. After a primer on refinement in Section 5, the proof architecture and library invariants are covered in Section 6. Finally, in Section 7, we attend to the different refinement phases, before concluding the paper in Section 8.

3. Related Work

While depth-first search is a well-known and widespread algorithm, not much work has been done on its formal verification. A very basic stand-alone formalization was done by Nishihara and Minamide [14], where two variants of a basic DFS are given (one with explicit stack, one without) and their equality is shown. Furthermore a couple of basic invariants are proved and code export is possible. But there is neither parametrization (it can solely compute the set of reachable nodes) nor flexible representation of the graph: It is fixed as a list of pairs.

Another basic approach is given by Pottier [15], where DFS is formalized in Coq to prove correct Kosaraju's algorithm for computing the strongly connected components. This formalization also allows for program extraction, but does not allow easy extension for use in other algorithms.

We described a first approach to a DFS framework in [12], on which this paper is based. The availability of a more advanced refinement framework (cf. Section 5) allows for a cleaner, more general, and more elegant framework. One notable improvement is that the hook functions are now specified in the nondeterminism monad of the refinement framework. This way the refinement based approach can also be used to develop the hook functions, which was not possible in [12]. Another improvement is the introduction of the *most specific invariant* (cf. Section 6), opposed to the notion of *DFS-constructable* in [12], which allows for an easier process of proving invariants.

In contrast to our development in [9], where we provide a collection of abstract lemmas that help in proving correct similar algorithms, the framework described in this paper is parameterized with hook functions that are invoked on well-defined extension points. This approach is less general, but greatly reduces the effort of instantiating it for new algorithms, as only the functions for the extension points have to be specified, while in [9] the whole algorithm has to be re-written.

4. Generic Framework

In its most well-known formulation, depth-first search is a very simple algorithm: For each node v_0 from a given set *V0* of start nodes, we invoke the function *DFS*. This function, if it has not seen the node yet, recursively invokes itself for each successor of the node.

```
discovered = {}
foreach v0 in V0 do DFS v0

DFS u:
 if u ∉ discovered then
  discovered = discovered ∪ {u}
  foreach v ∈ E``{u} do
   DFS v
```

Note that we use E for the (fixed) set of edges of the graph and R `` $S$ for the image of a set under the relation $R$ (in particular, $E$``$\{v\}$ denotes the set of successors of the node v).

In this simple form, the algorithm can only be used to create the set of reachable nodes, i. e., *discovered*. However, our aim, as laid out before, is to cover DFS based algorithms in general. Therefore we need to develop another view of the algorithm:

1. The algorithm above was given in a recursive form. For a correctness proof, we need to establish invariants for the two foreach-loops, and a pair of pre- and postcondition for the recursive call. This quite complex proof structure impedes the design of our framework. Thus, we use an iterative formulation of DFS that only consists of a single loop. Correctness proofs are done via a single loop invariant.
2. The simple algorithm above only computes a set of discovered nodes. However, in general, one wants to build up a DFS forest with cross and back edges and discovered and finished times.
3. To generalize over different DFS-based algorithms, we provide a skeleton DFS algorithm, which is parameterized by hook functions that are called from well-defined *extension points*, and modify an opaque *extension state*. Moreover, we add an additional break condition, which allows to interrupt the search prematurely, before all reachable nodes have been explored.

The skeleton algorithm is defined as follows:

```
DFS_step:
  if stack = [] then
   choose v0 from V0 ∩ (UNIV − discovered)
   new_root v0; on_new_root v0
  else
   (u,V) = get_pending
   case V of
    None ⇒ finish u; on_finish u
  | Some v ⇒
     if v ∉ discovered then
      discover u v; on_discover u v
     else if v ∈ set stack then
      back_edge u v; on_back_edge u v
     else
      cross_edge u v; on_cross_edge u v

cond s:
 ¬ is_break ∧ (V0 ⊆ discovered ⟶ stack ≠ [])

DFS:
 init; on_init
 while cond do
  DFS_step
```

The step-function has five cases. In each case, we first perform a transformation on the base part of the state (e. g., *finish*), and then call the associated hook function (e. g., *on_finish*). Note that hook functions only modify the extension state. We now describe the cases in more detail: If the stack is empty, we choose a start node that has not yet been discovered (the condition guarantees that there is one). The *new_root*-function pushes this node on the stack and

marks it as discovered. Moreover, it declares all outgoing edges as pending.

If the stack is non-empty, the *get_pending*-function tries to select a pending edge starting at the node u on top of the stack. If there are no such edges left, the *finish*-function pops u off the stack. Otherwise, we have selected a pending edge (u,v). If the node v has not yet been discovered, the *discover*-function marks it as discovered, pushes it on the stack, and declares all its outgoing edges as pending. Otherwise, we distinguish whether v is on the stack, in which case we have encountered a back edge, or not, in which case we have encountered a cross edge. The corresponding basic functions *back_edge* and *cross_edge* have no effect on the stack or the set of discovered nodes.

Note that we have not given an explicit definition of any basic function (e. g., *finish*, *get_pending*), but only stated behavioral requirements. Similarly, we have not described the exact content of the state, but merely expected it to contain a stack, a set of discovered nodes, and a set of pending edges. We will first initialize this generic algorithm with a very detailed state (cf. Section 6.1) and corresponding operations, and then refine it to more suitable states and operations, based on the requirements of the parameterization (cf. Section 7.1).

We now describe two different show-cases on how to instantiate our framework to useful algorithms:

Example 4.1. A simple application of DFS is a cyclicity check, based on the fact that there is a back edge if and only if there is a reachable cycle. The state extension consists of a single flag *cyc*, which signals that a back edge has been encountered, and causes the algorithm to terminate prematurely. The hooks are implemented as follows, where omitted ones default to *skip*:

on_init: cyc = False (∗ *initially no cycle has been found* ∗)
is_break: cyc (∗ *break iff cycle has been found* ∗)
on_back_edge u v: cyc = True (∗ *cycle!* ∗)

Example 4.2. Another important family of DFS based algorithms is nested depth-first search [3, 6, 16], which is used in model checkers to find acceptance cycles in Büchi-automata.

A nested DFS algorithm consists of two phases, *blue* and *red*. The *blue* phase walks the graph to find accepting nodes. On backtracking from such a node it starts the *red* phase. This phase tries to find a cycle containing this accepting node – depending on the specific algorithm, it searches for a path to a node on the stack, or to the accepting node. In any case, the *red* phase does not enter nodes which were already discovered by another *red* search.

The idea behind *red* search is not a concept specific to nested DFS, but is of a more general nature: Find a non-empty path to a node with a certain property, possibly excluding a set of nodes. The latter set has to be closed (i. e., there must be no edges leaving it) and must not contain any node with the property in question. Using our DFS framework, we formalize this algorithm as *find_path1_excl V0 P X* for some set of start nodes *V0*, some property P and a set of nodes to exclude X. It returns either a path to a node with property P, or a new exclusion $X' = X \cup E^+ \,``V0$ that is also closed and does not contain a node with property P. Note that we use E^+ for the transitive closure of E.

For the following description of the nested DFS formalization, we assume *find_path1_excl* to be given.

The extension to the state needed for nested DFS consists of two parts: The lasso (i. e., an accepting cycle plus a reaching path from a start node) and all the nodes visited by *red* searches. Therefore the obvious hooks are

on_init: lasso = None; red = {}
is_break: lasso ≠ *None.*

The next hook to implement is *on_finish*, where the *red* phase (that is *find_path1_excl*) has to be run. We define the auxiliary function *run_red_dfs* as follows:

```
run_red_dfs u:
  case find_path1_excl {u} (λx. x ∈ set stack) red of
    Inl X' ⇒ (∗ no path, but new exclusion ∗)
              red = X'
  | Inr p ⇒ (∗ path ∗)
              lasso = make_lasso p
```

The hook is then defined as

on_finish u: **if** *accepting u* **then** *run_red_dfs u.*

For more recent optimizations of nested DFS, like cycle detection on back edges [16], some other hooks have to be instantiated, too.

5. The Isabelle Refinement Framework

In order to formalize algorithms such as depth-first search, it is advantageous to start with an abstract description of the algorithmic idea, on which the correctness proof can be done in a concise way. The abstract description usually includes nondeterminism and is not executable.

For example, the *get_pending*-function in our skeleton algorithm (cf. Section 4) does not specify an order in which pending edges are selected, i. e., any pending edge may be chosen nondeterministically. Moreover, the set type used for the successors of a node has no counterpart in common programming languages, e. g., there is no set datatype in Standard ML.

Once the abstract algorithm is proved correct, it is refined towards a fully deterministic, executable version, possibly via multiple refinement steps. Each refinement step is done in a systematic way that guarantees preservation of correctness. For example, one can implement the graph by adjacency lists, and process the pending edges in list order.

The refinement approach simplifies the formalization by separating the correctness proof of the abstract algorithmic ideas from the correctness proof of the concrete implementation. Moreover, it allows to re-use the same abstract correctness proof with different implementations.

In Isabelle, this approach is supported by the Isabelle Refinement and Collections Frameworks [10, 11], and the Autoref tool [7]. Using ideas of refinement calculus [1], the Isabelle Refinement Framework provides a set of tools to concisely express nondeterministic programs, reason about their correctness, and refine them (in possibly many steps) towards efficient implementations. The Isabelle Collections Framework provides a library of verified efficient data structures for standard types such as sets and maps. Finally, the Autoref tool automates the refinement to efficient implementations, based on user-adjustable heuristics for selecting suitable data structures to implement the abstract types.

In the following, we describe the basics of the Isabelle Refinement Framework. The result of a (possibly nondeterministic) algorithm is described as a set of possible values, plus a special result *FAIL* that characterizes a failing assertion:

datatype *'a nres = RES 'a set* | *FAIL*

On results, we define an ordering by lifting the subset ordering, *FAIL* being the greatest element.

RES X ≤ *RES Y iff X* ⊆ *Y*
| *m* ≤ *FAIL* | *FAIL* ≰ *RES X*

Note that this ordering forms a complete lattice, where *RES* {} is the bottom, and *FAIL* is the top element. The intuitive meaning of $m \leq m'$ is that all possible values of m are also possible for m'. We

say that *m refines m'*. In order to describe that all values in *m* satisfy a condition Φ, we write $m \leq \textbf{spec}\ x.\ \Phi\ x$ (or shorter: $m \leq \textbf{spec}\ \Phi$), where $\textbf{spec}\ x.\ \Phi\ x \equiv RES\ \{x.\ \Phi\ x\}$.

Example 5.1. Let *cyc_checker E V0* be an algorithm that checks a graph over edges *E* and start nodes *V0* for cyclicity. Its correctness is described by the following formula, stating that it should return *true* if and only if the graph contains a cycle reachable from *V0*, which is expressed by the predicate *cyclic*:

$cyc\_checker\ E\ V0 \leq \textbf{spec}\ r.\ r = cyclic\ E\ V0$

Now let *cyc_checker_impl* be an efficient implementation[1] of *cyc_checker*. For refinement, we have to prove:

$cyc\_checker\_impl\ E\ V0 \leq cyc\_checker\ E\ V0.$

Note that, by transitivity, we also get that the implementation is correct:

$cyc\_checker\_impl\ E\ V0 \leq \textbf{spec}\ r.\ r = cyclic\ E\ V0$

To express nondeterministic algorithms, the Isabelle Refinement Framework uses a monad over nondeterministic results. It is defined by

$\textbf{return}\ x \equiv RES\ \{x\}$
$\textbf{bind}\ FAIL\ f \equiv FAIL \mid \textbf{bind}\ (RES\ X)\ f \equiv RES \bigcup x{\in}X.\ f\ x$

Intuitively, **return** *x* returns the deterministic outcome *x*, and **bind** *m f* is sequential composition, which describes the result of nondeterministically choosing a value from *m* and applying *f* to it. In this paper, we write *x=m; f x* instead of **bind** *m f*, to make program text more readable.

Recursion is described by a least fixed point, i. e., a function *F* with recursion equation $F\ x = B\ F\ x$ is described by $lfp\ (\lambda F\ x.\ B\ F\ x)$. To increase readability, we write a recursive function definition as *F x: B F x*. Based on recursion, the Isabelle Refinement Framework provides **while** and **foreach** loops. Note that we agree on a partial correctness semantics in this paper,[2] i. e., infinite executions do not contribute to the result of a recursion.

Another useful construct are assertions:

$\textbf{assert}\ \Phi \equiv \textbf{if}\ \Phi\ \textbf{then return}\ ()\ \textbf{else}\ FAIL$

An assertion generates an additional proof obligation when proving a program correct. However, when refining the program, the condition of the assertion can be assumed.

Example 5.2. The following program removes an arbitrary element from a non-empty set. It returns the element and the new set.

select s:
assert $s \neq \{\}$;
$x = \textbf{spec}\ x.\ x{\in}s$;
return $(x, s - \{x\})$

The assertion in the first line expresses the precondition that the set is not empty. If the set is empty, the result of the program is *FAIL*. The second line nondeterministically selects an element from the set, and the last line assembles the result: A pair of the element and the new set.

Using the verification condition generator of the Isabelle Refinement Framework, it is straightforward to prove the following lemma, which states that the program refines the specification of the correct result:

$s{\neq}\{\} \Longrightarrow select\ s \leq \textbf{spec}\ (x,s').\ \{x{\in}s \wedge s'{=}s{-}\{x\}\}$
unfolding *select_def* **by** *refine_vcg auto*

Typically, a refinement also changes the representation of data, e. g., a set of successor nodes may be implemented by a list. Such a *data refinement* is described by a relation *R* between concrete and abstract values. We define a *concretization function* $\Downarrow R$, that maps an abstract result to a concrete result:

$\Downarrow R\ FAIL \equiv FAIL$
$\mid\ \Downarrow R\ (RES\ X) \equiv \{c.\ \exists x{\in}X.\ (c,x){\in}R\}$

Intuitively, $\Downarrow R\ m$ contains all concrete values with an abstract counterpart in *m*.

Example 5.3. A finite set can be implemented by a duplicate-free list of its elements. This is described by the following relation:

$ls\_rel \equiv \{(l,s).\ set\ l = s \wedge distinct\ l\}$

The *select*-function from Example 5.2 can be implemented on lists as follows:

select' l:
assert $l{\neq}[]$;
$x = hd\ l$;
return $(x, tl\ l)$

Again, it is straightforward to show that *select'* refines *select*:

$(l,s){\in}ls\_rel \Longrightarrow select'\ l \leq \Downarrow(Id{\times}ls\_rel)\ (select\ s)$
unfolding *select'_def select_def*
by $(refine\_vcg)\ (auto\ simp:\ ls\_rel\_def\ neq\_Nil\_conv)$

Intuitively, this lemma states that, given the list *l* is an implementation of set *s*, the results of *select* and *select'* are related by $Id{\times}ls\_rel$, i. e., the first elements are equal, and the second elements are related by *ls_rel*.

Note that the assertion in the abstract *select*-function is crucial for this proof to work: For the empty set, we have $select\ \{\} = FAIL$, and the statement holds trivially. Thus, we may assume that $s{\neq}\{\}$, which implies $l{\neq}[]$, which, in turn, is required to reason about *hd l* and *tl l*. The handling of assertions is automated by the verification condition generator, which inserts the assumptions and discharges the trivial goals.

6. Proof Architecture

Recall that we have phrased the DFS algorithm as a single loop of the form:

init; **while** *cond* **do** *step*

Using the monad of the refinement framework, this is implemented by explicitly threading through the state, i. e.,

$\textbf{let}\ s = init;\ (\textbf{while}\ (\lambda s.\ cond\ s)\ \textbf{do}\ (\lambda s.\ step\ s))\ s.$

For better readability, we introduce the convention to omit the state parameter whenever it is clear which state to use.

Properties of the DFS algorithm are shown by establishing invariants, i. e., predicates that hold for all reachable states of the DFS algorithm.

The standard way to establish an invariant is to generalize it to an inductive invariant, and show that it holds for the initial state, and is preserved by steps of the algorithm. When using this approach naïvely, we face two problems:

1. The invariant to prove an algorithm correct typically is quite complicated. Proving it in one go results in big proofs that tend to get unreadable and hard to maintain. Moreover, there are many basic invariants that are used for almost all DFS-based

[1] For this example, we omit data-refinement of the parameters *E* and *V0*.

[2] The Isabelle Refinement Framework supports both, partial and total correctness semantics. However, the code generator of Isabelle/HOL currently only guarantees partial correctness of the generated code.

algorithms. These have to be re-proved for every algorithm, which is contrary to our intention to provide a framework that eliminates redundancy. Thus, we need a solution that allows us to establish invariants incrementally, re-using already established invariants to prove new ones.

2. Our refinement framework allows for failing assertions. If the algorithm may reach a failing assertion, we cannot establish any invariants. Thus we can only establish invariants of the base algorithm under the assumption that the hook functions do not fail. However, we would like to use invariants of the base algorithm to show that the whole algorithm is correct, in particular that the hook functions do not fail. Thus, we need a solution that allows us to establish invariants for the non-failing reachable states only, and a mechanism that later transfers these invariants to the actual algorithm.

In the following, we describe the proof architecture of our DFS framework, which solves the above problems. First, we define the operator[3] $\leq_n$ by $m \leq_n m' \equiv m \neq FAIL \longrightarrow m \leq m'$.

Thus, $m \leq_n$ **spec** Φ means, that m either fails or all its possible values satisfy Φ. With this, an inductive invariant of the non-failing part of the algorithm can be conveniently expressed as

$$is\_ind\_invar\ P \equiv init \leq_n \textbf{spec}\ P \\ \wedge (\forall s.\ P\ s \wedge cond\ s \longrightarrow step\ s \leq_n \textbf{spec}\ P).$$

It is straightforward to show that there exists a *most specific invariant I*, i. e., an inductive invariant that implies all other inductive invariants:

$$is\_ind\_invar\ I\ \textbf{and}\ [\![is\_ind\_invar\ P; I\ s]\!] \Longrightarrow P\ s$$

In order to establish invariants of the algorithm, we show that they are inductive invariants when combined with *I*. This leads to the following rule, which shows consequences of the most specific invariant:

lemma *establish_invar:*
assumes $init \leq_n$ **spec** P
assumes $\bigwedge s.\ [\![cond\ s; I\ s; P\ s]\!] \Longrightarrow step\ s \leq_n$ **spec** P
shows $I\ s \Longrightarrow P\ s$

When discharging the proof-obligation for a step, introduced by the second premise of this rule, we may assume $I\ s$, and thus re-use invariants that we have already proved.

In order to use invariants to show properties of the algorithm, we use the fact that at the end of a loop, the invariant holds and the condition does not:

$$(init;\ \textbf{while}\ cond\ \textbf{do}\ step) \leq_n \textbf{spec}\ s.\ I\ s \wedge \neg\ cond\ s$$

Finally, we use the following rule to show that the algorithm does not fail:

lemma *establish_nofail:*
assumes $init \neq FAIL$
assumes $\bigwedge s.\ [\![cond\ s; I\ s]\!] \Longrightarrow step\ s \neq FAIL$
shows $(init;\ \textbf{while}\ cond\ \textbf{do}\ step) \neq FAIL$

To simplify re-using and combining of already established invariants, we define a locale *DFS_invar*, which fixes a state s and assumes that the most specific invariant holds on s. Whenever we have established an invariant P, we also prove $P\ s$ inside this locale. In a proof to establish an invariant, we may interpret the locale, to have the already established invariants available.

[3] $\leq_n$ is not a partial order. However, it is reflexive, and fulfills a restricted transitivity law: $a \leq_n RES\ X \leq_n c \Longrightarrow a \leq_n c$

Example 6.1. In our parameterized DFS framework, we provide a version of *establish_invar* that splits over the different cases of *step*, and is focused on the hook functions:

lemma *establish_invar:*
assumes *init: on_init* $\leq_n$ **spec** *x. P* (*empty_state x*)
assumes *new_root:* $\bigwedge$*v0 s s'. pre_on_new_root v0 s s'*
$\Longrightarrow$ *on_new_root v0 s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
assumes *finish:* $\bigwedge$*u s s'. pre_on_finish u s s'*
$\Longrightarrow$ *on_finish u s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
assumes *cross_edge:* $\bigwedge$*u v s s'. pre_on_cross_edge u v s s'*
$\Longrightarrow$ *on_cross_edge u v s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
assumes *back_edge:* $\bigwedge$*u v s s'. pre_on_back_edge u v s s'*
$\Longrightarrow$ *on_back_edge u v s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
assumes *discover:* $\bigwedge$*u v s s'. pre_on_discover u v s s'*
$\Longrightarrow$ *on_discover u v s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
shows *is_invar P*

Here, *is_invar P* states that P is an invariant, *s'*(|*more* := *x*|) is the state *s'* with the extension part updated to x, and the *pre_*-predicates define the preconditions for the calls to the hook functions. For example, we have

$$pre\_on\_finish\ u\ s\ s' \equiv DFS\_invar\ s \wedge cond\ s \\ \wedge\ stack\ s \neq [] \wedge u = hd\ (stack\ s) \\ \wedge\ pending\ s\ ``\ \{u\} = \{\} \wedge s' = finish\ u\ s.$$

That is, the invariant holds on state s and s has no more pending edges from the topmost node on the stack. The state *s'* emerged from s by executing the *finish*-operation on the base DFS state.

A typical proof of an invariant P has the following structure:

lemma *P_invar: is_invar P*
proof (*induction rule: establish_invar*)
case (*discover u v s s'*)
then interpret *DFS_invar s* **by** *simp*
show *on_discover u v s'* $\leq_n$ **spec** *x. P* (*s'*(|*more* := *x*|))
...
next
...
qed
lemmas (**in** *DFS_invar*) *P* = *P_invar*[*THEN xfer_invar*]

The proof of the first lemma illustrates how the proof language Isar is used to write down a human-readable proof. The different cases that we have to handle correspond to the assumptions of the lemma *establish_invar*. The **interpret** command makes available all definitions and facts from the locale *DFS_invar*, which can then be used to show the statement. The second lemma just transfers the invariant to the *DFS_invar* locale, in which the fact *P s* is now available by the name *P*.

Note that this Isar proof scheme is only suited for invariants with complex proofs. Simpler invariant proofs can often be stated on a single line. For example, finiteness of the discovered edges is proved as follows:

lemma *is_invar* (*λs. finite* (*edges s*))
by (*induction rule: establish_invar*) *auto*

6.1 Library of Invariants

In the previous section we have described the proof architecture, which enables us to establish invariants of the depth-first search algorithm. In this section, we show how this architecture is put to use.

We define an *abstract* DFS algorithm, which is an instance of the generic algorithm presented in Section 4. Its state contains discovery and finished times of nodes, and a search forest with additional back and cross edges. In detail, the state consists of:

stack search stack

pending set of pending edges, i. e., the workset

δ partial function mapping each node to its discovery time, i. e., *dom* δ denotes the set of discovered nodes

φ partial function mapping each node to its finishing time, i. e., *dom* φ denotes the set of finished nodes

tree the search tree, i. e., the edges leading to the discovery of new nodes

back_edges the set of edges going back onto the stack

cross_edges the set of edges going to an already finished node

time current time

The abstract basic operations (e. g., *finish*, *get_pending*) are defined accordingly, fulfilling the requirements of the generic framework.

Based on this, we provide a variety of invariants, which use the information in the state at different levels of detail. Note that these invariants do not depend on the extension part of the state, and thus can be proven independently of the hook functions, which only update the extension part. Further note that we present them as they occur in the locale *DFS_invar*, which fixes the state and assumes that the most specific invariant holds (cf. Section 6).

For the sets *dom* δ of discovered and *dom* φ of finished nodes, we prove, among others, the following properties:

lemma *stack_set_def: set stack* = *dom* δ − *dom* φ
lemma *finished_closed:* E``*dom* $\varphi \subseteq$ *dom* δ
lemma *nc_finished_eq_reachable:*
$\neg$*cond* $\wedge$ $\neg$*is_break* $\Longrightarrow$ *dom* φ = E^*``*V0*

The first lemma states that the nodes on the stack are exactly those that have already been discovered, but not yet finished. The second lemma states that edges from finished nodes lead to discovered nodes, and the third lemma states that the finished nodes are exactly the nodes reachable from *V0* when the algorithm terminates without being interrupted.

We also prove more sophisticated properties found in standard textbooks (e. g., [2, pp. 606–608]), like the Parenthesis Theorem (the discovered/finished intervals of two nodes are either disjoint or the one is contained in the other, but there is no overlap) or the White-Path-Theorem (a node v is reachable in the search tree from a node u iff there is a white path from v to u, i. e., a path where all nodes are not yet discovered when v is).

lemma *parenthesis:*
assumes $v \in$ *dom* φ **and** $w \in$ *dom* φ
and $\delta\ v < \delta\ w$
shows $(* $ *disjoint* $*)\ \varphi\ v < \delta\ w$
$\vee\ (* v$ *contains* $w *)\ \varphi\ w < \varphi\ v$

lemma *white_path:*
assumes $v \in E^*$``*V0* **and** $w \in E^*$``*V0*
and $\neg$*cond* $\wedge$ $\neg$*is_break*
shows *white_path v w* $\longleftrightarrow$ $(v,w) \in$ *tree**

The Parenthesis Theorem is important to reason about paths in the search tree, as it allows us to gain insights just by looking at the timestamps:

lemma *tree_path_iff_parenthesis:*
assumes $v \in$ *dom* φ **and** $w \in$ *dom* φ
shows $(v,w) \in$ *tree*$^+$
$\longleftrightarrow \delta\ v < \delta\ w \wedge \varphi\ v > \varphi\ w$

From the location of two nodes in the search tree, we can deduce several properties of those nodes (e. g., the $\rightarrow$ direction of *tree_path_iff_parenthesis*). This can be used, for example, to show properties of back edges, as

lemma *back_edge_impl_tree_path:*
$[\![(v,w) \in$ *back_edges*; $v \neq w\ ]\!] \Longrightarrow (w,v) \in$ *tree*$^+$.

It can also be used to establish invariants about the root of a strongly connected component, i. e., the node of an SCC with the highest position in the tree, because

lemma *scc_root_scc_tree_trancl:*
$[\![$ *scc_root v scc*; $x \in$ *scc*; $x \in$ *dom* δ; $x \neq v\ ]\!]$
$\Longrightarrow (v,x) \in$ *tree*$^+$.

Utilizing the knowledge about the search tree, we can then show that a node of an SCC is its root iff it has the minimum discovery time of the SCC. This is an important fact, for example in the proof of Tarjan's SCC Algorithm.

Example 6.2. The idea of cycles in the set of reachable edges is independent of any DFS instantiation. Therefore we can provide invariants about the (a)cyclicity of those edges in the general library, the most important one linking acyclicity to the existence of back edges:

lemma *cycle_iff_back_edges:*
acyclic edges $\longleftrightarrow$ *back_edges* = $\{\}$

Here, *edges* is the union of all tree, cross, and back edges.

The $\rightarrow$ direction follows as an obvious corollary of the lemma *back_edge_impl_tree_path* shown above. The $\leftarrow$ direction follows from the fact that *acyclic* (*tree* $\cup$ *cross_edges*), the proof of which uses the Parenthesis Theorem.

Moreover, we need the fact that at the end of the search *edges* is the set of all reachable edges:

lemma *nc_edges_covered:*
assumes $\neg$*cond s* **and** $\neg$*is_break s*
shows $E \cap (E^*$``*V0*$) \times$ *UNIV* = *edges s*

With those facts from the library, we recall the definition of the cyclicity checker in our framework as presented in Example 4.1. Let *cycc* be that instantiation.

As the *cyc* flag is set when a back edge is encountered, the following invariant is easily proved:

lemma *i_cyc_eq_back:*
is_invar (λ*s. cyc s* $\longleftrightarrow$ *back_edges s* $\neq \{\}$)
apply (*induct rule: establish_invar*)
apply (*simp_all add: cond_def cong: cyc_more_cong*)
apply (*simp add: empty_state_def*)
done

This happens to be the only invariant that needs to be shown for the correctness proof. Using the invariants mentioned above, we easily get the following lemma inside the locale *DFS_invar*, i. e., under the assumption *I s*:

lemma (**in** *DFS_invar*) *cycc_correct_aux:*
assumes $\neg$*cond s*
shows *cyc s* $\longleftrightarrow$ $\neg$*acyclic* ($E \cap (E^*$``*V0*$) \times$ *UNIV*)

Intuitively, this lemma states that the *cyc* flag is equivalent to the existence of a reachable cycle upon termination of the algorithm. Finally, we gain the correctness lemma of the cyclicity checker as an easy consequence:

cycc E V0 $\leq$ **spec** *s.*
cyc s $\longleftrightarrow$ $\neg$*acyclic* ($E \cap (E^*$``*V0*$) \times$ *UNIV*).

7. Refinement

In Section 6, we have described the abstract DFS framework. We phrased the algorithm as a step-function on a state that contained detailed information. In order to implement an actual DFS-algorithm, most of this information is typically not required, and the required information should be represented by efficient data structures. Moreover, we want to choose between different implementation styles, like recursive or tail-recursive.

For this, our framework splits the refinement process into three phases: In the projection phase, we get rid of unnecessary information in the state. In the structural refinement phase, we choose the implementation style. Finally, in the code generation phase, we choose efficient data structures to represent the state, and extract executable code from the formalization.

Although the refinements are applied sequentially, our design keeps the phases as separated as possible, to avoid a cubic blowup of the formalization in the number of different states, implementation styles and efficient data structures.

7.1 Projection

To get a version of the algorithm over a state that only contains the necessary information, we use data refinement: We define a relation between the original *abstract* state and the reduced *concrete* state, as well as the basic operations on the concrete state. Then, we show that the operations on the concrete state refine their abstract counterparts. Using the refinement calculus provided by the Isabelle Refinement Framework, we easily lift this result to show that the concrete algorithm refines the abstract one.

In order to be modular w. r. t. the hook operations, we provide a set of standard implementations together with their refinement proofs, assuming that we have a valid refinement for the hooks. As for the abstract state, we also use extensible records for the concrete state. Thus, we obtain templates for concrete implementations, which are instantiated with a concrete data structure for the extension part of the state, a set of concrete hook operations, and refinement theorems for them.

Example 7.1. For many applications, such as the cyclicity checker from Example 4.1, it suffices to keep track of the stack, the pending edges, and the set of discovered nodes. We define a state type

record *'v simple_state =*
stack :: ('v × 'v set) list, on_stack :: 'v set, visited :: 'v set

and a corresponding refinement relation

$(s',s) \in R_X \equiv$
stack s' = map (λu. (u,pending s `` {u})) (stack s) ∧
on_stack s' = set (stack s) ∧
visited s' = dom (δ s) ∧
(more s', more s) ∈ X.

Note that we store the pending edges as part of the stack, and provide an extra field *on_stack* that stores the set of nodes on the stack. This is done with the implementation in mind, where cross and back edges are identified by a lookup in an efficient set data structure and the stack may be projected away when using a recursive implementation. Moreover, we parameterize the refinement relation with a relation X for the extension state.

Next, we define a set of concrete operations. For example, the concrete *discover* operation is defined as:

discover' u v:
stack = (v, E``{v}) # stack
on_stack = insert v on_stack
visited = insert v visited

It is straightforward to show that *discover'* refines the abstract *discover*-operation:

lemma *discover_refine:*
assumes $(s',s) \in R_X$
shows *discover' u v s' ≤⇓R_X discover u v s*

Assuming refinement of all hook operations, we get refinement of the abstract algorithm:

lemma *refine:*
assumes *on_init' ≤⇓X on_init*
and *⋀v0 s0 s s'. ⟦pre_on_new_root v0 s0 s; (s',s)∈R_X⟧*
⟹ on_new_root' s' ≤⇓X on_new_root s
and ...
shows *DFS' ≤⇓R_X DFS*

where *DFS'* is the DFS-algorithm over the concrete operations.

We also provide further implementations, which both require the hooks for back and cross edges to have no effect on the state. Thus the corresponding cases can be collapsed and there is no need to implement the *on_stack* set. As an additional optimization we pre-initialize the set of visited nodes to simulate a search with some nodes excluded[4]. As an example, this is used in the inner DFS of the nested DFS algorithm (cf. Example 4.2).

For the cyclicity checker, we define the concrete state by extending the *simple_state* record:

record *'v cycc_state_impl = 'v simple_state +*
cyc :: bool

The extension state will be refined by identity, i. e., the refinement relation for the concrete state is R_{Id}. We also define a set of concrete hook operations (which look exactly like their abstract counterparts)

on_init_impl: cyc = False
is_break_impl: cyc
on_back_edge_impl u v: cyc = True

It is trivial to show that these refine their abstract counterparts w. r. t. R_{Id}. Once this is done, the DFS framework gives us a cyclicity checker over the concrete state, and a refinement theorem:

cycc_impl ≤⇓R_Id cycc

7.2 Structural Refinement

Another aspect of refinement is the structure of the algorithm. Up to now, we have represented the algorithm as a while-loop over a step-function. This representation greatly simplifies the proof architecture, however, it is not how one would implement a concrete DFS algorithm[5]. We provide two standard implementations: A tail-recursive one and a recursive one. The tail-recursive implementation uses only while and foreach loops, maintaining the stack explicitly, while the recursive implementation uses a recursive function and requires no explicit stack.

We are interested in making the structural refinement of the algorithm independent of the projection, such that we can combine different structural refinements with different projections, without doing a quadratic number of refinement proofs. For this purpose we formalize the structural refinements in the generic setting (cf. Section 4) first. Depending on the desired structure, we have to add some minimal assumptions on the state and generic operations, as will be detailed below. The resulting generic algorithms are then instantiated by the concrete state and operations from the projection phase, thereby discharging the additional assumptions.

The following listing depicts the pseudocode for the tail-recursive implementation, using the basic DFS operations:

[4] This is an optimization that saves one membership query per node.

[5] For example, checking the loop condition would require iteration over all root nodes each time.

```
tailrec_DFS:
 init; on_init
 foreach v0 in V0 do
  if is_break then break
  if not discovered v0 then
   new_root v0; on_new_root v0
   while (stack != [] ∧ ¬ is_break) do
    (u,V) = get_pending
    case V of
     None ⇒ finish u; on_finish u
    | Some v ⇒ {
       if discovered v then
        if finished v then
         cross_edge u v; on_cross_edge u v
        else
         back_edge u v; on_back_edge u v
       else
        discover u v; on_discover u v
      }
```

This implementation iterates over all root nodes. For each root node, it calls *new_root* and then executes steps of the original algorithm until the stack is empty again. Note that we effectively replace the arbitrary choice of the next root node by the outer foreach-loop. In order for this implementation to be a refinement of the original generic algorithm, we have to assume that 1) the stack is initially empty, such that we can start with choosing a root node, and 2) the same root node cannot be chosen twice, so that we are actually finished when we have iterated over all root nodes. In order to ensure 2), we assume that *new_root* sets the node to discovered, and no operation can decrease the set of discovered nodes.

With these assumptions, we can use the infrastructure of the Isabelle Refinement Framework to show that the algorithm *tailrec_DFS* refines the original *DFS*.

The next listing depicts the pseudocode for the recursive implementation:

```
recursive_DFS:
 init; on_init
 foreach v0 in V0 do
  if is_break then break
  if not discovered v0 then
   new_root v0; on_new_root v0
   inner_dfs v0

inner_dfs u:
 foreach v in E``{u} do {
  if is_break then break
  choose_pending u (Some v)
  if discovered v
   if finished v then
    cross_edge u v; on_cross_edge u v
   else
    back_edge u v; on_back_edge u v
  else
   discover u v; on_discover u v;
   if ¬ is_break then inner_dfs v
 }
 choose_pending u None;
 finish u; on_finish u
```

As in the tail-recursive implementation, we iterate over all root nodes. For each root node, we call the recursive function *inner_dfs*. Intuitively, this function handles a newly discovered node: It iterates over its successors, and for each successor, it decides whether it induces a cross or back edge, or leads to a newly discovered node. In the latter case, *inner_dfs* is called recursively on this newly discovered node. Finally, if all successor nodes have been processed, the node is finished.

Intuitively, this implementation replaces the explicit stack of the original algorithm by recursion.

Apart from the assumptions 1) and 2) from *tailrec_DFS*, we need some additional assumptions to show that this implementation refines the original algorithm: 3) The operation *new_root v0* initializes the stack to only contain *v0*, and the pending edges to all outgoing edges of *v0*; the operation *discover u* pushes *u* on the stack and adds its outgoing edges to the set of pending edges; the *finish*-operation pops the topmost node from the stack. 4) The *get_pending*-operation of the original algorithm must have the form of selecting a pending edge from the top of the stack, if any, and then calling the operation *choose_pending* for this edge, where *choose_pending* removes the edge from the set of pending edges.

With these assumptions we show that *recursive_DFS* refines the original *DFS* algorithm. Note that the refinement proof requires the state to contain a stack, which is however not used by the recursive algorithm. Provided that the parameterization does not require a stack either, we can add an additional data refinement step to remove the stack. For convenience, we combine this step with the automatic refinement to efficient data structures, which is described below.

Note that these assumptions are natural for any set of operations on a DFS state. The advantage of this formulation is its independence from the actual operations. Thus, the same formalization can be used to derive implementations for all states and corresponding operations, which reduces redundancies, and even makes proofs more tractable, as it abstracts from the details of a concrete data structure to its essential properties.

Example 7.2. Recall the simple state from Example 7.1. The simple implementation satisfies all assumptions required for the tail-recursive and recursive implementation, independent of the parameterization. Thus, upon refining an algorithm to *simple_state*, we automatically get a tail-recursive and a recursive implementation, together with their refinement theorems. In case of the cyclicity checker, we get:

$cycc\_tr\_impl \leq \Downarrow R_{Id}\ cycc$ **and**
$cycc\_rec\_impl \leq \Downarrow R_{Id}\ cycc$

7.3 Code Generation

After projection and structural refinement have been done, the algorithm is still described in terms of quite abstract data structures like sets and lists. In a last refinement step, these are refined to efficiently executable data structures, like hash-tables and array-lists. To this end, the Isabelle Collections Framework [10] provides a large library of efficient data structures and generic algorithms, and the Autoref-tool [7] provides a mechanism to automatically synthesize an efficient implementation and a refinement theorem, guided by user-configurable heuristics.

Note that we do this last refinement step only after we have fully instantiated the DFS-scheme. This has the advantage that we can choose the most adequate data structures for the actual algorithm. The fact that the refinements for the basic DFS operations are performed redundantly for each actual algorithm does not result in larger formalizations, as it is done automatically.

Example 7.3. In order to generate an executable cyclicity checker, we start with the constant *cycc_tr_impl*, which is the tail-recursive version of the cyclicity checker, using the *simple_state* (cf. Example 7.2). The state consists of a stack, an on-stack set, a visited set, and the *cyc*-flag. Based on this, we define the cyclicity checker by

```
cyc_checker E V0:
 s = cycc_tr_impl E V0;
 return (cyc s).
```

To generate executable code, we first have to write a few lines of canonical boilerplate to set up Autoref to work with the extension state of the cyclicity checker. The executable version of the algorithm is then synthesized by the following Isabelle commands:

```
schematic_lemma cycc_impl:
  fixes V0::'v::hashable set and E::('v×'v) set
  defines V ≡ Id :: ('v × 'v) set
  assumes [unfolded V_def,autoref_rules]:
    (succi,E)∈⟨V⟩slg_rel
    (V0',V0)∈⟨V⟩list_set_rel
  notes [unfolded V_def,autoref_tyrel] =
    TYRELI[where R=⟨V⟩dflt_ahs_rel]
    TYRELI[where R=⟨V × ⟨V⟩list_set_rel⟩ras_rel]
  shows nres_of (?c::?'c dres) ≤⇓?R (cyc_checker E V0)
  unfolding cycc_tr_impl_def[abs_def] cyc_checker_def
  by autoref_monadic

concrete_definition cycc_exec uses cycc_impl
export_code cycc_exec in SML
```

The first command uses the Autoref-tool to synthesize a refinement. The **fixes** line declares the types of the abstract parameters, restricting the node-type to be of the *hashable* typeclass. The next line defines a shortcut for the implementation relation for nodes, which is fixed to identity here. The assumptions declare the refinement of the abstract to the concrete parameters: The edges are implemented by a successor function, using the relator *slg_rel*, which is provided by the CAVA automata library [8]. The set of start nodes are implemented by a duplication-free list, using the relator *list_set_rel* from the Isabelle Collections Framework, which is roughly the same as *ls_rel* from Example 5.3.

Finally, the **notes**-part gives some hints to the heuristics: The first hint causes sets of nodes to be implemented by hash-tables. This hint matches the on-stack and visited fields of the state. The second hint matches the stack field, and causes it to be implemented by an array-list, where the sets of pending nodes are implemented by duplication-free lists of their elements. Again, the required datatypes and their relators *dflt_ahs_rel* and *ras_rel* are provided by the Isabelle Collections Framework.

Ultimately, the *autoref_monadic* method generates a refinement theorem of the shape indicated by the **show**-part, where *?c* is replaced by the concrete algorithm, and *?R* is replaced by the refinement relation for the result. The second command defines a new constant for the synthesized algorithm, and also provides a refinement theorem with the constant folded. As the generated algorithm only uses executable data structures, the code generator of Isabelle/HOL [5] can be used to generate efficient Standard ML code.

8. Conclusion and Future Work

In this paper, we have presented a framework that supports a stepwise refinement development approach of DFS-based algorithms. On the abstract level, we have a generic formalization of DFS, which is parametrized by hook functions that operate on an opaque extension state. Properties of the algorithm are proved via invariants. To establish new invariants, only their preservation by the hook functions has to be shown. Moreover, invariants can be established incrementally, i. e., already proven invariants can be used when establishing new ones. To this end, our framework provides a large library of parametrization-independent standard invariants, which greatly simplify the correctness proofs of actual instantiations of the framework. For example, the cyclicity checker (cf. Example 4.1) required only one additional invariant with a straightforward 3-line proof.

Furthermore, the framework allows to refine both, data-structures and algorithm-structure, where the latter is (in general) independent of the actual instantiation. The data-refinement, as shown in this paper, is the prerequisite for the aforementioned library of invariants, as it allows to project a detailed abstract state to a small concrete state. This way, it is possible to have proof-supporting information without the necessity to actually gather it at runtime.

The framework supports various default concrete states. Using them only requires a refinement proof of the hook-functions.

To show the usability of the presented framework, we have formalized several examples from easy (Cyclicity Checker) to more advanced (Tarjan's SCC algorithm). In this paper, we presented the development of the Cyclicity Checker and the approach for formalizing a nested DFS algorithm.

The main contribution of this paper is the DFS-framework itself, and its design approach, which is not limited to DFS algorithms. The first general design principle is the technique of incrementally establishing invariants, which allows us to provide a standard library of invariants, which are independent of the actual instantiation. In Isabelle/HOL, this technique is elegantly implemented via locales.

The second general principle is to provide an algorithm over a detailed state at the abstract level, and then use refinement to project away the unused parts of the state for the implementation. This allows us to have a common abstract base for all instantiations.

Finally, we provide different implementation styles for the same algorithm, in a way that is independent of the concrete data structures, only making some basic assumptions. This allows us to decouple the data refinement and the structural refinement.

8.1 Future Work

An interesting direction of future work is to extend the framework to more general classes of algorithms. For example, when dropping the restriction that pending edges need to come from the top of the stack, one gets a general class of search algorithms, also including breadth-first search, and best-first search.

Currently, our framework only supports an invariant-based proof style. However, in many textbooks, proofs about DFS algorithms are presented by arguing over the already completed search forest. This proof style can be integrated in our framework by (conceptually) splitting the DFS algorithm into two phases: The first phase creates the DFS forest, only using the base state, while the second phase recurses over the created forest and executes the hook functions. It remains future work to elaborate this approach and explore whether it results in more elegant proofs.

References

[1] R.-J. Back and J. von Wright. *Refinement Calculus — A Systematic Introduction*. Springer, 1998.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[3] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[4] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable LTL model checker. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

[5] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *FLOPS*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.

[6] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proc. of the 2nd SPIN Workshop*, volume 32 of *Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1997.

[7] P. Lammich. Automatic data refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 84–99. Springer, 2013.

[8] P. Lammich. The CAVA automata library. In *Isabelle Workshop 2014*, 2014.

[9] P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In G. Klein and R. Gamboa, editors, *ITP*, volume 8558 of *LNCS*, pages 325–340. Springer, 2014.

[10] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *LNCS*, pages 339–354. Springer, 2010.

[11] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In L. Beringer and A. Felty, editors, *ITP*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

[12] R. Neumann. A framework for verified depth-first algorithms. In A. McIver and P. Höfner, editors, *Proc. of the Workshop on Automated Theory Exploration (ATX 2012)*, pages 36–45. EasyChair, 2012.

[13] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[14] T. Nishihara and Y. Minamide. Depth first search. *Archive of Formal Proofs*, June 2004. `http://afp.sf.net/entries/Depth-First-Search.shtml`, Formal proof development.

[15] F. Pottier. Depth-first search and strong connectivity in Coq, 2014. `http://gallium.inria.fr/~fpottier/publis/fpottier-dfs-scc.pdf`.

[16] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.

[17] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.