A Framework for Verified Depth-First Algorithms

René Neumann*
Technische Universität München
Garching, Germany
rene.neumann@in.tum.de

Abstract

We present a framework in Isabelle/HOL for formalizing variants of depth-first search. This framework allows to easily prove non-trivial properties of these variants. Moreover, verified code in several programming languages including Haskell, Scala and Standard ML can be generated.

In this paper, we present an abstract formalization of depth-first search and demonstrate how it is refined to an efficiently executable version. Further we use the emptiness-problem of Büchi-automata known from model checking as the motivation to present three Nested DFS algorithms. They are formalized, verified and transformed into executable code using our framework.

1 Introduction

Model-checking is a widespread technology that is used in the analysis of a multitude of systems (e.g. software, hardware, or communication protocols) [16]. To prove certain correctness properties of a system, its state space is exhaustively analyzed. One of the approaches – proposed by Vardi and Wolper [22, 23] – reduces model checking problems to operations on Büchi-automata [3]. But it requires a verified or at least trusted implementation of these operations to generate trustworthy results. Therefore the CAVA-project[1], which we are part of, is working on creating verified code for these operations so that it will be eventually possible to build a model checker out of this verified code. The code can then be used as a reference implementation by more efficient implementations, i.e. the results of the efficient model checker could be verified by our implementation. That way the efficient model checker could be considered "correct", with the same notion of "correctness" that is applied to programs after checking their results using tests.

One algorithm that is fundamental in automata-theoretic model checking (and graph-theory as a whole) is depth-first search (DFS)[21]. This algorithm is especially interesting, as it is used in several different areas and with different goals (finding cycles, finding a specific node, generating the set of reachable nodes, ...) and might even be restricted by special constraints. An example of such a constraint is the need to generate the automaton on the fly, as the model checking process yields huge automata where it in general is not feasible to generate the automaton upfront and hold it in memory. All these points have to be considered to create a flexible formalization that fits all the use cases. This flexibility and the different use cases result in non-trivial proofs of many properties. Experience shows that these proofs on paper are not always free of mistakes, and even if they are, the implementations do not necessarily need to be (see for example the findings of Schimpf et al. [19]). On the other hand, some of these proofs are shared between several algorithms and it is not economical to redo them each time. In this paper, we therefore present a general verified formalization of DFS resulting in a framework that allows the formalization of different variants of DFS. This lowers the barrier of formalizing and proving DFS-based algorithms.

*Supported by DFG project *Computergestützte Verifikation von Automatenkonstruktionen für Model Checking*
[1]Computer Aided Verification of Automata: http://cava.in.tum.de

An important problem in the field of model checking is checking the emptiness of Büchi-automata. Algorithms exist to solve this problem by means of DFS. Examples are Nested DFS [6] and algorithms to find strongly connected components (SCCs) of graphs [7, 8] – enhancements of Tarjan's original algorithm [21]. In this paper, we present how we embedded different variants of Nested DFS into our framework.

As mentioned earlier, CAVA strives to result in a verified and executable implementation. Hence, we do not only formalize DFS and Nested DFS in the interactive theorem prover Isabelle/HOL [17], but also use the power of its code generator [10] to produce verified functional code[2].

Related Work Lammich and Tuerk [15] formalized Hopcroft's algorithm to determinize NFAs as part of CAVA with the same goal of providing proven base-algorithms for model checking. Coming from the same motivation, Chou and Peled [4] and Ray et al. [18] verified other algorithms used in model checking, though without generating executable code. Furthermore Schimpf et al. [19] formalized the translation of LTL to BA, and also generate code.

Outline We first present an abstract view on how to formalize general DFS and give examples of the properties we verified. Then we present a specific use case of DFS (Nested DFS to find cycles in Büchi-automata) in Section 3, before showing how to bridge the gap to executable code in Section 4. The paper is ended with an outlook of our future plans on this topic.

2 A general DFS framework

2.1 Algorithms

Algorithm 1 Simple DFS

```
1: visited ← ∅
2: procedure DFS x
3:     if x ∉ visited then
4:         visited ← visited ∪ {x}
5:         for all e ∈ succs x do
6:             DFS e
7: DFS start
```

Depth-first search is, in its most well-known formulation, a very simple algorithm (see Algorithm 1): We iterate through all nodes reachable from a given starting node[3] visiting the successors of a node in a non-deterministic order. The call-stack of the procedure also serves implicitly as the path from the starting node to the current node. In this form the algorithm can only be used to create the set of reachable nodes (= *visited*), which does not fulfill our requirements stated in Section 1, in particular the ability to support all use cases of DFS.

Therefore we enhance this simple algorithm to modify an opaque state σ (see Alg. 2) via the three functions ACTION (called when visiting a node), REMOVE (called when we omit a node because it has been visited already) and POST (called when having visited all successors of the

[2]The Isabelle theory files and the generated code can be downloaded from `http://cava.in.tum.de/downloads`.

[3]We do not consider depth-first forests like it is done in other places [12, 5], because their direct use complicates the proofs though the gains are neglegible.

Algorithm 2 Simple DFS with state augmentation

```
1:  visited ← ∅; σ ← σ₀
2:  procedure DFS x
3:      if COND σ then
4:          if x ∉ visited then
5:              visited ← visited ∪ {x}; σ ← ACTION σ x
6:              for all e ∈ SUCCS x do
7:                  DFS e
8:          else
9:              σ ← REMOVE σ x
10:         σ ← POST σ x
11: DFS start
```

node and backtracking to its parent). Additionally, we introduce a fourth function COND (cf. line 3) that serves as a flag signalling an abortion of the search. With this augmentation, the search can be utilized for different use cases by implementing these functions accordingly.

Example To search for a specific node, σ could be implemented as a boolean flag. This would be set to **true** in ACTION if the node is encountered. And by letting COND be $\lambda\,\sigma.\,\neg\sigma$, the loop would be aborted if the node has been found.

Algorithm 3 Functionalized DFS with state augmentation

```
1:  function DFS-STEP R s
2:      let x :: xs = stack s and w :: ws = wl s in
3:      if w = ∅ then                                          ▷ backtrack
4:          s(|stack ← xs, σ ← POST (σ s) s x, wl ← ws|)
5:      else
6:          choose e ∈ w and let w' = w − {e} in
7:          if e ∈ visited s then                              ▷ already visited
8:              s(|σ ← REMOVE (σ s) s e, wl ← w' :: ws|)
9:          else                                               ▷ visit new node
10:             s(|stack ← e :: x :: xs, wl ← (SUCCS e \ R) :: w' :: ws,
11:               σ ← ACTION (σ s) s e, visited ← visited s ∪ {e}|)

12: function DFS-START R x
13:     (|stack ← [x], visited ← {x}, wl ← [SUCCS x \ R], σ ← (σ₀ x)|)

14: function DFS R start
15:     while (λ s. stack s ≠ [] ∧ COND (σ s)) (DFS-STEP R) (DFS-START R start)
```

Though reasoning about imperative algorithms is possible in Isabelle/HOL [2], reasoning about functional programs is preferred. Hence, the imperative DFS algorithm is rewritten as functions (see Alg. 3), where we encapsulate each step of the search into an explicit DFS-state s that is modified inside a while-loop. Though the concept of while-loops is primarily known from the imperative world, they are used here for conceptual and technical reasons, in particular to uncouple the modification of states from the recursion. But they do not add expressiveness and

can easily be replaced by direct recursive calls inside DFS-STEP.

Besides this change, we also make the stack and a list of working sets (wl) explicit by adding them as part of the DFS-state. The latter contains the set of those successors of a particular node that still need to be processed. This list of working sets became necessary when we dropped the **forall**-construct, which has been hiding this kind of bookkeeping.

As can be seen in line 6 the non-deterministic choice is still preserved and we will outline in Section 4 the way of implementing this construct. For this section, we consider DFS-STEP to return a set containing exactly the results generated for each possible $e \in w$.

Note that in the same step we introduce a set R that restricts the generation of successors insofar as nodes in R are excluded from visiting. For the rest of this paper we make R implicit though, to help readibility.

The Algorithm 3 does not allow to prove important properties, especially those concerning the time at which nodes are discovered or backtracked from. Therefore, we further extend it by adding more fields to the DFS-state and manipulating them accordingly (code omitted for brevity): We add a counter to represent time, and also two maps ϕ and δ, assigning timestamps to each finished (ϕ) and discovered (δ) node. As a bit of notation we further write *discovered* s instead of $dom(\delta\ s)$ and equivalently *finished* s for $dom(\phi\ s)$, thus *discovered* resembles the former *visited*.

In the introduction in Section 1 we mentioned the necessary ability of generating the automaton on the fly. This is addressed by handling the function returning the successors (SUCCS) as an argument to the DFS algorithm, similiar to ACTION, etc. This allows the user of this DFS library to provide any graph model that fits his needs as long as he is able to provide the needed soundness proofs.

2.2 Properties

From the bare algorithms we move on to constructs that allow easy and elegant proofs. We start by lifting the condition of the while-loop and its body into an explicit predicate:

$$\text{DFS-NEXT}\ s\ s' :\Longleftrightarrow \mathit{stack}\ s \neq [] \wedge \text{COND}\ (\sigma\ s) \wedge s' \in \text{DFS-STEP}\ s$$

This can further be expanded to yield the set of all DFS-states that can be generated from the starting state:

$$\text{DFS-CONSTR-FROM}\ x := \text{DFS-NEXT}^*\ (\text{DFS-START}\ x)$$

Thanks to this construction, we can use induction to prove predicates over all possible DFS-states. Examples of such predicates are ($v \rightarrow^*_{\setminus R} w$ denotes reachability over $V \setminus R$):

$$\forall x\ s\ v.\ s \in \text{DFS-CONSTR-FROM}\ x \Longrightarrow v \in \mathit{finished}\ s \Longrightarrow x \rightarrow^*_{\setminus R} v$$
$$\forall x\ s\ v\ w.\ s \in \text{DFS-CONSTR-FROM}\ x \Longrightarrow w \not\rightarrow^* v \Longrightarrow v \not\rightarrow^* w \Longrightarrow$$
$$v \in \mathit{finished}\ s \Longrightarrow w \in \mathit{finished}\ s \Longrightarrow \delta\ s\ w > \phi\ s\ v \vee \phi\ s\ w < \delta\ s\ v$$

The second lemma (together with the fact that $\forall v.\ \delta\ s\ v < \phi\ s\ v$ holds) already is the proof of an important property: The intervals created by the discovery and the finishing time of two nodes are disjoint if neither can reach the other in the graph. We moreover formalize and proof crucial theorems (Parenthesis Theorem, White-path Theorem) regarding depth-first search trees that Cormen et al. [5, pp. 606–608] formulate[4].

[4] In the Isabelle theories, these proofs can be found in the `TreeDFS`-theory.

Algorithm 4 Nested DFS by Courcoubetis et al. [6]

```
1:  procedure NESTED-DFS
2:      DFS-BLUE start
3:  procedure DFS-BLUE s
4:      s.blue ← true
5:      for all t ∈ SUCCS s do
6:          if ¬ t.blue then
7:              DFS-BLUE t
8:      if s ∈ 𝒜 then                ▷ s is accepting
9:          DFS-RED s s
10: procedure DFS-RED seed s
11:     s.red ← true
12:     for all t ∈ SUCCS s do
13:         if ¬ t.red then
14:             DFS-RED seed t
15:         else if t = seed then
16:             report cycle
```

We now define two more predicates reflecting the need to express properties about the state of a finished search:

$$\text{DFS-FINISHED } x\ s :\Longleftrightarrow s \in \text{DFS-CONSTR-FROM } x \wedge \neg\exists\, s'.\ \text{DFS-NEXT } s\ s'$$
$$\text{DFS-COMPLETED } x\ s :\Longleftrightarrow \text{DFS-FINISHED } x\ s \wedge \text{COND } (\sigma\ s)$$

Intuitively, DFS-FINISHED $x\ s$ holds iff s is the final DFS-state starting from x. DFS-COMPLETED $x\ s$ then adds the additional constraint, that the search has exhaustively discovered all reachable states, i.e. it has not been aborted beforehand. Interesting properties that can be proven now are for example:

$$\forall x\ s.\ \text{DFS-COMPLETED } x\ s \Longrightarrow \mathit{finished}\ s = \mathit{discovered}\ s$$
$$\forall x\ s.\ \text{DFS-COMPLETED } x\ s \Longrightarrow \mathit{finished}\ s = \{v \mid v = x \vee x \rightarrow^+_{\setminus R} v\}$$
$$\forall x\ s.\ s \in \text{DFS-CONSTR-FROM } x \Longrightarrow \mathit{stack}\ s = [] \Longrightarrow \text{DFS-COMPLETED } x\ s$$

3 Case Study: Nested DFS

Nested DFS is an approach for checking emptiness of Büchi-automata: The BA is empty if and only if there is no cycle (reachable from the starting node) that contains at least one accepting state (for the rest of the section we will use $\mathcal{A}$ to denote the set of accepting states). Hence the general procedure of Nested DFS is to use a first DFS run (the "blue" one) to find the accepting nodes and then run another DFS (the "red" one) from each of these nodes to find a path back to that node – resulting in a cycle.

The first Nested DFS proposal is due to Courcoubetis et al. [6] and is presented in Alg. 4. Here the red search tries to find a path directly to the node it started from (*seed* in DFS-RED).

Note that the knowledge whether a node has been searched in a red DFS is shared globally by having the boolean *red* directly attached to the node (cf. line 11). This avoids searching subtrees that have been searched already in previous runs of the red DFS. Without this behavior a plain DFS with linear runtime (in the size of the edges) is started for each accepting state. As the number of them is also linear, the resulting runtime for the whole algorithm would be quadratic. But with the approach shown in Alg. 4 the result is an overall linear runtime for the whole search: We visit an edge two times in each colored DFS, once upon reaching the node and once upon backtracking.

Please also note that this omission of nodes visited in other red searches has the crucial prerequisite of triggering the red search while backtracking (see line 9). Running the red part directly in the discovery phase would result in an incomplete algorithm as possible cycles might be missed. Proving this is non-trivial (on paper). But as the ability to restrict the search by a certain set is built into the general framework of DFS (the parameter R in the algorithms of Section 2.1), it does not pose a problem in our formalization.

Enhancements of the basic Nested DFS are for example given by Holzmann et al. [11], where the red search succeeds if a path to the stack of the blue one has been found, or by Schwoon and Esparza [20], where also the blue search is utilizied for cycle detection. See the latter paper for some elaborations about different implementations of Nested DFS.

As it turns out, the differences between the algorithms are rather small. Hence, it is sensible to show general results over a parametrized core component first, and implement the differences thereafter. This also allows to add other implementations of Nested DFS without having to start from scratch. By using the general DFS-body described in the previous section we get most of the important properties for free, most importantly statements of reachability.

The algorithm of the red phase can be abstracted to one that tries to find a non-empty path into a given set ss. We also allow to pass a (possibly empty) set of nodes R that the search must not visit. Using the definitions of Section 2.1 we get the following implementation of a DFS, where σ is just a boolean value signalling whether a path into ss has been found yet:

$$\begin{aligned}\textsc{SubDFS}\ ss\ R = (\!|\ &\textsc{cond} = \lambda\,\sigma.\,\neg\sigma,\ \sigma_0 = \lambda\,x.\,\textbf{false},\ \textsc{restrict} = R,\\ &\textsc{action} = \lambda\,\sigma\ s\ e.\,\sigma \vee e \in ss,\ \textsc{post} = \lambda\,\sigma\ s\ e.\,\sigma\\ &\textsc{remove} = \lambda\,\sigma\ s\ e.\,\sigma \vee (e = \text{start}\ s \wedge e \in ss)\ |\!)\end{aligned}$$

The REMOVE specification is required, because the starting node of this search might be in the set ss but it is not sufficient to check this at the start, as the path should be non-empty. This can only be ensured if there is another reachable node of which the starting node is a successor. And it is not covered by ACTION as this method is only called for non-visited nodes.

For this application of a DFS it can be shown that it fullfills its specification:

$$\forall x\ s.\ \text{DFS-FINISHED}\ x\ s \Longrightarrow \sigma\ s \leftrightarrow \exists\, v.v \in ss \wedge x \rightarrow^+_{\setminus R} v$$

Now, having the red DFS, we continue with the blue part, whose specification is: On backtracking from an accepting node run the red DFS, while sparing all nodes already visited in a red DFS. Therefore, our opaque state σ now contains two parts: a boolean value signalling the discovery of a cycle (b) and a set of all nodes visited through red depth-first searches (F).

$$\begin{aligned}\textsc{NestedDFS}\ M = (\!|\ &\textsc{cond} = \lambda\,(b,F).\,\neg b,\ \sigma_0 = \lambda\,x.\,(\textbf{false},\emptyset),\ \textsc{restrict} = \emptyset,\\ &\textsc{action} = \lambda\,\sigma\ s\ e.\,\sigma,\ \textsc{remove} = \lambda\,\sigma\ s\ e.\,\sigma\\ &\textsc{post} = \lambda\,(b,F)\ s\ e.\,\textbf{if}\ b \vee e \notin \mathcal{A}\ \textbf{then}\ (b,F)\ \textbf{else}\\ &\quad \textbf{let}\ \mathit{sub} = \textsc{SubDFS}\ (M\ s)\ F\ \textbf{in}\\ &\quad \textbf{if}\ \sigma\ \mathit{sub}\ \textbf{then}\ (\textbf{true},F)\ \textbf{else}\ (\textbf{false},F \cup \mathit{finished}\ \mathit{sub})\ |\!)\end{aligned}$$

Note that NESTEDDFS is parametrized over some M that returns the set into which SUBDFS tries to find a path. Hence we get the implementation of Courcoubetis et al. [6] with NESTEDDFS $(\lambda\,s.\,\{\texttt{head}\ (\mathit{stack}\ s)\})$ and the implementation of Holzmann et al. [11] with NESTEDDFS $(\lambda\,s.\,\mathit{set}\ (\mathit{stack}\ s))$.

Since we can show the general correctness property

$$\forall x\, s.\ (\forall s.\, \texttt{head}\ (stack\ s) \in M\ s \wedge M\ s \subseteq set\ (stack\ s)) \Longrightarrow \text{DFS-FINISHED}\ x\ s \Longrightarrow \texttt{fst}(\sigma\ s) \leftrightarrow \exists v \in \mathcal{A}.\, v \rightarrow^+ v \wedge x \rightarrow^* v$$

we immediately get the correctness of these two implementations.

We can further enhance this to also give a formalization of the Nested DFS algorithm due to Schwoon and Esparza [20], as this is the algorithm of Holzmann et al. with an additional check in the blue phase: If we encounter a node that is already on the stack and either that node or our parent node (i.e. the head of the stack) is accepting, we have found a cycle. Thus we replace the noop-implementation of REMOVE by the following one:

$$\lambda\,(b, F)\ s\ e.\, (b \vee (e \in set(stack\ s) \wedge (e \in \mathcal{A} \vee \texttt{head}\ (stack\ s) \in \mathcal{A})), F)$$

The results of the previous formalizations can also be used here – all that remains to be shown is that the new REMOVE implementation is well-behaved.

In this section, we showed that the parametrization makes it possible to do step-wise formalization of DFS-applications. This allows to focus on the immediate problems and properties, while being able to use theorems about the lower layers in a black-box-like style. It also allows to verify different variants without needing to redo basic proofs. Though lines of code are not a very reliable measure, they give some hints about dimensions: Specifying and proving properties of the parameterized blue and red DFS took about 700 lines (this does not include the "raw" DFS parts), while instantiating them for the two variants is around 30 lines each. The more complex formalization of the Schwoon-Esparza-algorithm took another 250 lines.

4 Implementation

The mechanized proofs of DFS presented in the previous sections are not enough to create verified model checkers: We also need to provide code that has the properties we proved. For tasks like this, Isabelle/HOL provides a code-generator [10, 9] which allows to convert functional definitions into code in a functional language (SML, OCaml, Haskell, Scala). But as the definitions from the previous sections might include constructs of higher-order logic like quantification or non-determinism, they are not necessarily expressible in a (deterministic) program as is. Thus it is not sufficient to solely pass them on to the code-generator.

Moreover abstract implementations of data structures like sets, as they are provided by Isabelle, are easier to use in proofs, but tend to be inefficient for executable code. Therefore it is necessary to derive a definition that fulfills the same properties as the more abstract definition, but replaces the offending data structures and concepts by better fitted alternatives. The same holds for the algorithms itself: It is, for instance, more efficient to directly modify the set of visited nodes of the red DFS, the set F in the previous section, directly in place instead of performing rather expensive unions.

This is achieved using a *refinement framework* by Lammich [15], that allows refinement [1] on functions given in a monadic form [24]. Here a refinement is a relation $\leq$ on two non-deterministic programs S and S' such that $S \leq S'$ holds if and only if all possible results of S are also results of S'.[5] Data refinement is the special case, where the structure of the program is kept but data structures are replaced – for example abstract sets by concrete lists. The refinement framework provides the ability to annotate abstraction relations $R \subseteq c \times a$ that relate concrete values c

[5] We do not consider the special cases *FAIL* and *SUCCEED* here. Please refer to the work of Lammich [15].

with abstract values a. This is then written $S \leq \Downarrow R\ S'$, expressing that if some x is a result of S, there exists some x' such that $(x, x') \in R$ and x' is a result of S'.

Example It is possible to replace the specification "some $x \in X$" by the `head`-operation on a list xs where $set\ xs = X$. In terms of the refinement this is expressed by

$$set\ xs = X \Longrightarrow \mathsf{RETURN}\ \texttt{head}\ xs \leq\ \mathsf{SPEC}\ \lambda x.\, x \in X$$

Lammich provides one more framework named the Isabelle Collections Framework (ICF) [13, 14]. This framework allows to create the proofs just with an interface of the operations (called *locales* in Isabelle) and later "plugging-in" different concrete implementations of abstract data structures that satisfy this interface. Furthermore, the ICF already ships with different implementations allowing to change the concrete data structures without much hassle.

With these ingredients, we provide a modified version of DFS-STEP called DFS-STEP-IMPL where operations on sets and maps are replaced by their counterparts of the ICF and the operations on the opaque state σ are replaced by user-specified implementations of the four operations (ACTION by ACTION-IMPL, COND by COND-IMPL and so on). The same is done with SUCCS that is replaced by SUCCS-IMPL. Under the assumptions that the user has proven these implementations to be correct (where α_σ is the abstraction relation on the opaque states and α the abstraction relation on the DFS-states):

$$\mathsf{RETURN}\ \textsc{action-Impl}\ (\sigma\ s)\ s\ x \leq \Downarrow \alpha_\sigma\ \mathsf{RETURN}\ \textsc{action}\ (\alpha_\sigma(\sigma\ s))\ (\alpha\ s)\ x$$
$$\ldots$$
$$\textsc{cond-Impl}\ (\sigma\ s) \leftrightarrow \textsc{cond}\ (\alpha_\sigma(\sigma\ s))$$
$$set(\textsc{succs-Impl}\ x) = \textsc{succs}\ x$$

we show that

$$\mathsf{RETURN}\ \text{DFS-STEP-IMPL}\ s \leq \Downarrow \alpha\ \text{DFS-STEP}\ (\alpha\ s).$$

This intuitively describes that the implementation of DFS-STEP returns a result whose abstraction is a possible result of DFS-STEP. Therefore, all properties on the abstract level carry over into the concrete world. Of course, this also holds for the final while-loop:

$$\mathsf{RETURN}\ \text{DFS-IMPL}\ x \leq \Downarrow \alpha\ \text{DFS}\ x.$$

With the help of the refinement framework one can generate code such that

$$\mathsf{RETURN}\ \text{dfs-code}\ x \leq \Downarrow \alpha\ \mathsf{RETURN}\ \text{DFS-IMPL}\ x$$

and by transitivity it holds that

$$\mathsf{RETURN}\ \text{dfs-code}\ x \leq \Downarrow \alpha\ \text{DFS}\ x.$$

In particular, it can be shown that $\alpha(\text{dfs-code}\ x) \in \text{DFS-CONSTR-FROM}\ x$. Therefore it is guaranteed that all the properties shown for elements of DFS-CONSTR-FROM also hold on the result of the code-equation, and – after it has been converted into executable code with the help of the Isabelle code generator – also on the generated code[6].

[6] We provide generated and tested code in OCaml, SML, and Haskell. Please refer to the `README` distributed in the archive for details.

5 Future Work

In this paper we presented a formal framework for modelling, verifying, and implementing algorithms based on depth-first search. We showed the development of that framework from the well-known DFS algorithm to a while-loop with states. We then presented a show-case and implemented three different Nested DFS algorithms in that framework. Finally, we showed how this is then refined into executable code.

Our framework is a good starting point for other applications besides Nested DFS. One further application we want to formalize are the different SCC-algorithms [7, 8].

However, there are also some points that we would like to see included in the framework itself: Currently ACTION, etc. are defined as simple functions that do not support non-determinism and hence must return the same value for the same set of arguments. This is a drawback as it reduces the possibilities of what can be accomplished in them: For instance, it is not possible to return a counter-example in addition to the simple "yes/no" answer, because the counter-example depends on the series of non-deterministic successor-choices. Therefore, we want to embed them into the same non-deterministic monad the main DFS-loop is already in. We also expect certain proofs to become easier then.

Another addition would be proofs about the complexity of the operations, for example showing that the implementations of Nested DFS presented in the previous section are running in linear time. This addition would make the set of formalized properties more complete.

Finally, a proper benchmarking of the run-time of the generated code needs to be done. This benchmarking should also take into account different representations of sets.

References

[1] R.-J. J. Back. *On the correctness of refinement steps in program development.* PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[2] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming with Isabelle/HOL. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proc. of the 21th International Conference on Theorem Proving in Higher Order Logics (TPHOL)*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2008.

[3] J. R. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Proc. of the International Congress Logic, Methodology and Philosophy of Science 1960*, volume 44 of *Studies in Logic and the Foundations of Mathematics*, pages 1–11. Elsevier, 1966.

[4] C.-T. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, 23(3):265–298, 1999.

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

[6] 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.

[7] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In J. Wing, J. Woodcock, and J. Davies, editors, *Proc. Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271. Springer, 1999.

[8] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004.

[9] F. Haftmann. *Code Generation from Specifications in Higher Order Logic.* PhD thesis, Technische Universität München, 2009.

[10] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*. Springer, 2010.

[11] 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.

[12] D. J. King and J. Launchbury. Structuring depth-first search algorithms in Haskell. In *Proc. of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 344–354. ACM Press, 1995.

[13] P. Lammich. Collections Framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. `http://afp.sourceforge.net/entries/Collections.shtml`, 2009. Formal proof development.

[14] P. Lammich and A. Lochbihler. The Isabelle Collections Framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[15] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. Interactive Theorem Proving (ITP 2012), to appear, 2012.

[16] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer, 2001.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[18] S. Ray, J. Matthews, and M. Tuttle. Certifying compositional model checking algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J. S. Moore, editors, *4th International Workshop on the ACL2 Theorem Prover and its Applications*, 2003.

[19] A. Schimpf, S. Merz, and J.-G. Smaus. Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009.

[20] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

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

[22] M. Y. Vardi. Verification of concurrent programs: the automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1-2):79–98, 1991.

[23] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[24] P. Wadler. The essence of functional programming. In *Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 1–14. ACM Press, 1992.