Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm

Peter Lammich

Technische Universität München, lammich@in.tum.de

Abstract. We present an Isabelle/HOL formalization of Gabow's algorithm for finding the strongly connected components of a directed graph. Using data refinement techniques, we extract efficient code that performs comparable to a reference implementation in Java. Our style of formalization allows for reusing large parts of the proofs when defining variants of the algorithm. We demonstrate this by verifying an algorithm for the emptiness check of generalized Büchi automata, reusing most of the existing proofs.

1 Introduction

A strongly connected component (SCC) of a directed graph is a maximal subset of mutually reachable nodes. Finding the SCCs is a standard problem from graph theory with applications in many fields [27, Chap. 4.2].

There are several algorithms to partition the nodes of a graph into SCCs, the main ones being the Kosaraju-Sharir algorithm [28], Tarjan's algorithm [29], and the class of path-based algorithms [25,22,7,4,9].

In this paper, we present the verification of Gabow's path-based SCC algorithm [9] within the theorem prover Isabelle/HOL [24]. Using refinement techniques and efficient verified data structures, we extract Standard ML (SML) [21] code from the formalization. Our verified algorithm has a performance comparable to a reference implementation in Java, taken from Sedgewick and Wayne's textbook on algorithms [27, Chap. 4.2].

Our main interest in SCC algorithms stems from the fact that they can be used for the emptiness check of generalized Büchi automata (GBA), a problem that arises in LTL model checking [30,10,6]. Towards this end, we extend the algorithm to check the emptiness of generalized Büchi automata, reusing many of the proofs from the original verification.

Contributions and Related Work Up to our knowledge, we present the first mechanically verified SCC algorithm, as well as the first mechanically verified SCC-based emptiness check for GBA. Path-based algorithms have already been regarded for the emptiness check of GBAs [26]. However, we are the first to use the data structure proposed by Gabow [9].[1] Finally, our development is

[1] Although called Gabow-based algorithm in [26], a union-find data structure is used to implement collapsing of nodes, while Gabow proposes a different data structure [9, pg. 109]

a case study for using the Isabelle/HOL Monadic Refinement and Collection Frameworks [14,19,17,18] to engineer a verified, efficient implementation of a quite complex algorithm, while keeping proofs modular and reusable.

This development is part of the Cava project [8] to produce a fully verified LTL model checker. The current Cava model checker converts GBAs to standard Büchi automata, and uses an emptiness check based on nested depth first search [5,23]. Using GBAs directly typically yields smaller search spaces, thus making tractable bigger models and/or more complex properties [6].

The Isabelle source code of the formalization described in this paper is publicly available [15].

Outline The rest of this paper is organized as follows: In Section 2, we recall Gabow's SCC algorithm and present our extension for the emptiness check of generalized Büchi automata. Moreover, we briefly introduce the Isabelle/HOL Refinement and Collection Frameworks. Section 3 presents the formalization of the abstract algorithm, Section 4 presents the refinement to Gabow's data structure, and Section 5 presents the refinement to executable code. Finally, Section 6 reports on performance benchmarks and Section 7 contains conclusions and directions of future work.

2 Preliminaries

In this section, we present the preliminaries of our formalization. Subsection 2.1 recalls Gabow's algorithm and GBAs. Subsection 2.2 outlines our verification approach based on the Isabelle/HOL Refinement and Collection Frameworks.

2.1 Path-Based Strongly Connected Component Algorithms

Let $G = (V, E)$ be a finite directed graph over nodes V and edges $E \subseteq V \times V$. A *strongly connected component* (SCC) is a maximal set of nodes $U \subseteq V$, such that all nodes in U are *mutually reachable*, i. e. for all $u, v \in U$, there is a directed path from u to v.

A *path based SCC algorithm* is a depth first search (DFS) through the graph that, whenever it finds a back edge, contracts all nodes in the cycle closed by this edge [9]. To distinguish contracted nodes from nodes of the original graph, we refer to the former ones as *c-nodes*.

The algorithm starts with the original graph and a path that consists of a single arbitrary node. In each step, an edge starting at the end of the path is selected. If it leads back to a c-node on the path, all c-nodes on the cycle formed by this back edge and the path are collapsed. If the edge leads to an unfinished c-node, this node is appended to the path. Otherwise, the edge is ignored. If all edges from the end of the path have already been considered, the last c-node is removed from the path and marked as finished. At this point, the last c-node represents an SCC of the original graph. If the path becomes empty, the algorithm is repeated for another unfinished node, until all nodes have been finished.

Implementation The problem when implementing this algorithm is to keep track of the collapsed nodes in the graph efficiently. Initially, general set merging algorithms were proposed for identifying the collapsed nodes [25,22]. The idea of [9] (variants of it are also used in [7,4]) is to represent the current path by two stacks: A stack S that contains nodes of the original graph, and a *boundary stack* B that contains indexes into the first stack, which represent the boundaries between the collapsed nodes. For example, to represent the path $[\{a, b\}, \{c\}, \{d, e\}]$, one uses $S = [a, b, c, d, e]$ and $B = [0, 2, 3]$. Moreover, the (partial) *index map* I maps each node on S to its index. I is also used to represent nodes that belong to finished c-nodes, by mapping them to a special value (e. g. a negative number).

Collapsing is always due to a back edge (u, v), where u is in the last c-node of the path. Thus, it can be implemented by looking up the index $I\ v$ of v, and then popping elements from B until its topmost element is less than or equal to $I\ v$. Appending a new c-node to the path is implemented by pushing it onto S and its index onto B. Removing the last c-node from the path is implemented by popping elements from S until its length becomes *top* B, and then popping the topmost element from B.

With this data structure, the algorithm runs in time $O(|V| + |E|)$, i. e. linear time in the size of the graph [9].

For our main purpose, i. e. LTL model checking, we generalize the algorithm to only consider the part of the graph that is reachable from a set of start nodes $V_0 \subseteq V$. This is easily achieved by only repeating the algorithm for unfinished nodes from V_0.

Generalized Büchi Automata Generalized Büchi Automata (GBA) [30] have been introduced as a generalization of Büchi automata (BA) [3] that allows for more efficient automata based LTL model checking [31].

A GBA is a finite directed graph (V, E) with a set of initial nodes $V_0 \subseteq V$, a finite set of *acceptance classes* C, and a map $F : V \to 2^C$. As we are only interested in emptiness, we need not consider an alphabet.

An *accepting run* is an infinite path starting at a node from V_0, such that a node from each acceptance class occurs infinitely often on that path. A GBA is *non-empty*, if it has an accepting run. As the GBA is finite, this is equivalent to having a reachable *accepting cycle*, i. e. a cyclic, finite path with at least one edge that contains nodes from all acceptance classes. Obviously, a graph has a reachable accepting cycle iff it has a reachable non-trivial SCC that contains nodes from all acceptance classes. Here, an SCC is called *non-trivial*, if there is at least one edge between its nodes.

To decide emptiness, we don't need to compute all SCCs first: As the c-nodes on the path are always subsets of SCCs, we can report ,,non-empty" already if the last c-node on the path contains nodes from all acceptance classes, after being collapsed (i. e. becoming non-trivial). This way, the algorithm reports non-emptiness as soon as it has seen all edges of an accepting cycle.

To implement this check efficiently, we store the set of acceptance classes for each c-node on the path. This information can be added to the B stack, or maintained as a separate stack. On collapsing, the sets belonging to the collapsed

c-nodes are joined. This adds a factor of $|C|$ to the run time. However, $|C|$ is typically small, such that the sets can be implemented efficiently, e. g. as bit vectors.

2.2 Refinement Based Program Verification in Isabelle/HOL

Our formalization is done in four main steps, using Isabelle/HOL [24]:

1. Verification of the abstract path-based algorithm.
2. Refinement to Gabow's data structure.
3. Refinement to efficient data structures (e. g. arrays, hash tables).
4. Extraction of Standard ML code.

The key advantage of this approach is that proofs in one step are not influenced by proofs in the other steps, which greatly reduces the complexity of the whole development, and makes more complex developments possible in the first place.

With its refinement calculus [1] that is based on a nondeterminism monad [32], the Monadic Refinement Framework [19,17] provides a concise way to phrase the algorithms and refinements in Steps 1–3. The Isabelle Collection Framework [14,16] contributes the efficient data structures. Moreover, we use the Autoref tool [18] to add some automation in Step 3. Finally, we use the Isabelle/HOL code generator [11,12] in Step 4. In the following, we briefly recall these techniques.

Isabelle/HOL Isabelle/HOL [24] is a theorem prover for higher order logic. The listings contained in this paper are actual Isabelle/HOL source, sometimes slightly polished for better readability. We quickly review some non-standard syntax used in this paper: Functions are defined by sequential pattern matching, using $\equiv$ as defining equation operator. Theorems are written as $[\![P_1,\ldots,P_n]\!] \Longrightarrow Q$, which is syntactic sugar for $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$.

Program Refinement The Monadic Refinement Framework represents programs as a monad over the type *'a nres* = **res** *'a set* | **fail**. A result **res** X means that the program nondeterministically returns a value from the set X, and the result **fail** means that an assertion failed. The subset ordering is lifted to results:

res $X \leq$ **res** $Y \equiv X \subseteq Y$ | _ $\leq$ **fail** $\equiv$ *True* | **fail** $\leq$ _ $\equiv$ *False*

Intuitively, $m \leq m'$ (m *refines* m') means that all possible values of m are also possible values of m'. Note that this ordering yields a complete lattice on results, with smallest element **res** $\{\}$ and greatest element **fail**. The monad operations are then defined as follows:

return $x \equiv$ **res** $\{x\}$
bind (**res** X) $f \equiv$ *Sup* $\{f\, x \mid x{\in}X\}$ | **bind fail** $f \equiv$ **fail**

Intuitively, **return** x is the result that contains the single value x, and **bind** m f is sequential composition: Choose a value from m, and apply f to it.

As a shortcut to specify values satisfying a given predicate Φ, we define **spec** $\Phi \equiv$ **res** $\{x \mid \Phi\ x\}$. Moreover, we use a Haskell-like do-notation, and define a shortcut for assertions:

assert $\Phi \equiv$ **if** Φ **then return** () **else fail**

Recursion is defined by a fixed point:

rec $B\ x \equiv$ **do** {**assert** (*mono* B)*; gfp* $B\ x$}

As we use the greatest fixed point, a non-terminating recursion causes the result to be **fail**. This matches the notion of total correctness. Note that we assert monotonicity of the recursive function's body B, which typically follows by construction [13]. On top of the **rec** primitive, we define loop constructs like **while** and **foreach**.

In a typical development based on stepwise refinement, one specifies a series of programs $m_1 \leq \ldots \leq m_n$, such that m_n has the form **spec** Φ, where Φ is the specification, and m_1 is the final implementation. In each refinement step (from m_{i+1} to m_i), some aspects of the program are refined.

Example 1. Given a finite set S of sets, the following specifies a set r that contains at least one element from every non-empty set in S:

$sel_3\ S \equiv$ **do** {**assert** (*finite* S)*;* (**spec** $r.\ \forall s \in S.\ s \neq \{\} \longrightarrow r \cap s \neq \{\}$)}

This specification can be implemented by iteration over the outer set, adding an arbitrary element from each non-empty inner set to the result:

```
sel₂ S ≡ do {
  assert (finite S);
  foreach S (λs r.
    if s={} then return r else do {x←spec x. x∈s; return (insert x r)}
  ) {} }
```

Using the verification condition generator (VCG) of the monadic refinement framework, it is straightforward to show that sel_2 is a refinement of sel_3:

```
lemma sel₂ S ≤ sel₃ S
  unfolding sel₂_def sel₃_def
  by (refine_rcg foreach_rule[where I=λit r. ∀s∈S − it. s≠{} ⟶ r∩s≠{}])
    auto
```

As constructs used in monadic programs are monotonic, sel_3 can be replaced by sel_2 in a bigger program, yielding a correct refinement.

Data Refinement In a typical refinement based development, one also wants to refine the representation of data. For example, we need to refine the abstract path by Gabow's data structure. A data refinement is specified by a single-valued *refinement relation* between concrete and abstract values. Equivalently, it can be expressed by an *abstraction function* from concrete to abstract values and an invariant on concrete values. A prototypical example is implementing sets by

distinct lists, i. e. lists that contain no duplicate elements. Here, the refinement relation $\langle R \rangle list\_set\_rel$ relates a distinct list to the set of its elements, where the elements are related by R.

Given a refinement relation R, we define the function $\Downarrow_R$ to map results over the abstract type to results over the concrete type:

$$\Downarrow_R (\textbf{res } A) \equiv \textbf{res } \{c \mid \exists a \in A.\ (c,a) \in R\} \quad | \quad \Downarrow_R \textbf{fail} \equiv \textbf{fail}$$

Thus, $m_1 \leq \Downarrow_R m_2$ states that m_1 is a refinement of m_2 w. r. t. the refinement relation R, i. e. all concrete values in m_1 correspond to abstract values in m_2.

The Monadic Refinement Framework implements a refinement calculus [1] that is used by the VCG for refinement proofs. Moreover, the Autoref tool [18] can be used to automatically synthesize the concrete program and the refinement proof, guided by user-adjustable heuristics to find suitable implementations of abstract data types. For the algorithm sel_2 from Example 1, Autoref generates the implementation

$sel_1\ Xi \equiv foldl$
  $(\lambda\sigma\ x.\ \textbf{if}\ is\_Nil\ x\ \textbf{then}\ \sigma\ \textbf{else let}\ xa = hd\ x\ \textbf{in}\ glist\_insert\ op = xa\ \sigma)\ []\ Xi$

and proves the theorem

$(Xi1,\ X1) \in \langle\langle Id\rangle list\_set\_rel\rangle list\_set\_rel \Longrightarrow$
  $\textbf{return}\ (sel_1\ Xi1) \leq \Downarrow_{\langle Id\rangle list\_set\_rel}\ (sel_2\ X1)$

By default, Autoref uses the Isabelle Collection Framework [14,16], which provides a large selection of verified collection data structures.

Code Generation After the last refinement step, one typically has arrived at a deterministic program inside the executable fragment of Isabelle/HOL. The code generator [11,12] extracts this program to Standard ML code. For example, it generates the following ML function for sel_1:

```
fun sel1 xi =
  List.foldl (fn sigma => fn x =>
    (if Autoref_Bindings_HOL.is_Nil x then sigma
     else let val xa = List.hd x;
         in Impl_List_Set.glist_insert Arith.equal_nat xa sigma
         end)) [] xi;
```

3 Abstract Algorithm

In this section, we describe our formalization of the abstract path based algorithm for finding SCCs. The goal is to formalize two variants of the algorithm, one for computing a list of SCCs, and another for emptiness check of GBAs, while sharing common parts of the proofs. For this purpose, we first define a skeleton algorithm that maintains the path through the graph, but does not store the found SCCs. This skeleton algorithm helps us finding invariants that hold in all

```
skeleton ≡ do {
  let D = {};
  foreach^{outer_invar} V_0 (λv_0 D_0. do {
    if v_0∉D_0 then do {
      let (p,D,pE) = initial v_0 D_0;

      (p,D,pE) ← while^{invar v_0 D_0} (λ(p,D,pE). p ≠ []) (λ(p,D,pE). do {
        (vo,(p,D,pE)) ← select_edge (p,D,pE);
        case vo of
          Some v ⇒ do {
            if v ∈ ⋃set p then return (collapse v (p,D,pE))
            else if v∉D then return (push v (p,D,pE))
            else return (p,D,pE)
          }
        | None ⇒ return (pop (p,D,pE))
      }) (p,D,pE);
      return D
    } else
      return D_0
  }) D
}
```

Listing 1.1: Skeleton of a path-based algorithm

path-based algorithms, and can be used as a starting point for defining the actual algorithms. Listing 1.1 displays the code of the skeleton algorithm.[2] It formalizes the path-based algorithm sketched in Section 2.1: First, we initialize the set D of finished nodes. Then, we iterate over the root nodes V_0 of the graph, and for each unfinished one, we start the inner loop of the search algorithm, which runs until the path becomes empty again. In the inner loop, we additionally keep track of the current path p and a set pE of *pending edges*, i. e. edges that have not yet been explored and start from nodes on the path. For better manageability of the proofs, we have defined constants for the basic operations:

```
initial v_0 D_0 ≡ ([{v_0}], D_0, E ∩ {v_0} × UNIV)
select_edge (p,D,pE) ≡ do {
    e ← select (pE ∩ last p × UNIV);
    case e of
      None ⇒ return (None,(p,D,pE))
    | Some (u,v) ⇒ return (Some v, (p,D,pE − {(u,v)}))
  }
collapse v (p,D,pE) ≡ let i=idx_of p v in (take i p @ [⋃set (drop i p)],D,pE)
  where idx_of p v ≡ THE i. i<length p ∧ v∈p!i
push v (p, D, pE) ≡ (p @ [{v}], D, pE ∪ E ∩ {v} × UNIV)
pop (p, D, pE) ≡ (butlast p, last p ∪ D, pE)
```

[2] Shortened a bit by removing some **assert**-statements.

These constants are defined over the whole state (p, D, pE) of the inner loop, even if they only work on parts of it. This allows for nicer refinement proofs, as operations on the abstract state are refined to operations on the concrete state, without exposing the inner structure of the states, which differs on the concrete and abstract domain. Note that this also results in a more modular correctness proof, as invariant preservation can be shown separately for each operation.

We briefly explain the operations: The *initial* operation initializes the state for the inner loop with a path that consists of the single node v_0. The *select_edge* operation checks if there is a pending edge from the end of the path. If there is no such edge, it returns *None* and does not change the state. Otherwise, it removes the edge from the set of pending edges, and returns its target node. The operation *collapse* first determines the index of the node on the path, and then collapses the corresponding suffix of the path. The operation *push* appends a new node to the path, and the operation *pop* removes the last node from the path.

3.1 Invariants

Correctness of while and foreach loops is proved by establishing a loop invariant. Moreover, we have to show that the body of a while loop transforms states within a well-founded relation, and that the set iterated over by a foreach loop is finite.

We specify invariants for the skeleton algorithm and show that they are preserved by the operations inside the loop. The invariants and the preservation lemmas are then reused for the actual algorithms. In Listing 1.1, the loops are annotated with their invariants, such that the VCG sees them.

The invariant of the outer loop depends on the nodes *it* still to be iterated over, and on the finished nodes *D*. It states that (1) we only iterate over start nodes, (2) nodes that we have already iterated over are finished, (3) finished nodes are reachable, and (4) edges from finished nodes lead to finished nodes again. The invariant is formalized using the locale mechanism of Isabelle/HOL [2]:

locale *outer_invar* = *digraph_loc* + **fixes** *it* **and** *D*
  assumes *1:* $it \subseteq V_0$ **and** *2:* $V_0 - it \subseteq D$ **and** *3:* $D \subseteq E^* ``V_0$ **and** *4:* $E``D \subseteq D$

Here, *digraph_loc* defines a finite directed graph, represented by its edge relation E and a set of initial nodes V_0. The set V of nodes is implicitly fixed to the universal set *UNIV*, and thus not explicitly mentioned in the formalization. Moreover, r^* denotes the reflexive transitive closure of a relation r, and $r``s = \{y.\ \exists x \in s.\ (x,y) \in r\}$ denotes the image of a set s under a relation r.

The invariant *invar* v_0 D_0 (p,D,pE) of the inner loop is more complex. We only sketch its main idea here, and refer the reader to the actual formalization [15] for more details. The main parts of the inner loop's invariant are:

(1) Edges from finished nodes lead to finished nodes; nodes on the path are not finished; non-pending edges from the path lead either to nodes on the path or to finished nodes.
(2) Only pending edges may go back on the path.
(3) The nodes inside a c-node on the path are mutually reachable.

(1) is a standard invariant for DFS. (2) ensures that all cycles that have already been seen are collapsed, and (3) ensures that the c-nodes on the path are always subsets of SCCs. In particular, when a c-node is popped from the path, it has no pending edges left. Then, it easily follows from the invariant that this c-node is a maximal set of mutually reachable nodes, i. e. an SCC.

We now apply the VCG to the skeleton algorithm, after unfolding the definition of *select_edge*. This leaves us with proof obligations to show invariant preservation for each of the operations in the loop. These are proved as separate lemmas, to be reused later. For example, we prove for the pop operation:

invar_pop: $[\![invar\ v_0\ D_0\ (p,\ D,\ pE);\ p \neq [];\ pE \cap last\ p \times UNIV = \{\}]\!]$
$\Longrightarrow invar\ v_0\ D_0\ (pop\ (p,\ D,\ pE))$

To show termination of the inner loop, we define an edge to be *visited* if it is not pending and starts at a finished node, or at a node on the path. We then show that, in each step, either the set of visited edges grows, or it remains the same and the path length decreases. Technically, we define a well-founded relation over the state of the while loop and show that the operations are compatible with it. These verification conditions are also proved as separate lemmas.

3.2 Computing the SCCs

In a next step, we extend the skeleton algorithm to actually compute a list of SCCs of the graph. We define the algorithm *compute_SCC* by replacing the statement **return** (*pop* (*p,D,pE*)) in Listing 1.1 with **return** (*last p#l,pop* (*p,D,pE*)), and pass the list l through the inner and outer loop, initializing it to the empty list.

In order to specify the intended result, we first define a strongly connected component as a maximal mutually connected set of nodes:

$is\_scc\ E\ U \equiv U \times U \subseteq E^* \wedge (\forall U'.\ U' \supset U \longrightarrow \neg\ (U' \times U' \subseteq E^*))$

Then, we define the intended result as a list that covers all reachable nodes and contains SCCs in (reverse) topological order:

compute_SCC_spec $\equiv$ **spec** l.
$\bigcup set\ l = E^* ``V_0 \wedge (\forall U \in set\ l.\ is\_scc\ E\ U)$
$\wedge\ (\forall i\ j.\ i<j \wedge j<length\ l \longrightarrow l!j \times l!i \cap E^* = \{\})$

Next, we extend the invariant of the skeleton algorithm. The invariant extension is the same for the inner and outer invariant, and states that the list computed so far (1) covers exactly the finished nodes and (2) contains SCCs in (3) reverse topological order. The new invariants can be elegantly defined using the locale mechanism of Isabelle/HOL:

locale *cscc_invar_ext* = *digraph_loc* + **fixes** $l\ D$
assumes *1:* $\bigcup set\ l = D$ **and** *2:* $\forall U \in set\ l.\ is\_scc\ E\ U$
and *3:* $\bigwedge i\ j.\ [\![i<j;\ j<length\ l]\!] \Longrightarrow l!j \times l!i \cap E^* = \{\}$

locale *cscc_outer_invar* = *outer_invar* + *cscc_invar_ext*
locale *cscc_invar* = *invar* + *cscc_invar_ext*

In order to prove the algorithm correct, we have to show that the extended invariant is preserved. We add the following rule to the VCG:

$$\textit{cscc\_invarI: } [\![\textit{invar } v_0\ D_0\ s;\ \textit{invar } v_0\ D_0\ s \Longrightarrow \textit{cscc\_invar\_ext } (l,\ s)]\!] \\ \Longrightarrow \textit{cscc\_invar } v_0\ D_0\ (l,\ s)$$

We also add the analogous rule *cscc_outer_invarI* for the outer loop's invariant. These rules split a proof of the extended invariant into a proof of the original invariant and a proof of the invariant extension.

As we already have proved lemmas for the verification conditions concerning *invar*, we only have to prove lemmas for the invariant extension. For example, for finishing a node, we prove

$$\textit{cscc\_invar\_pop: } [\![\textit{cscc\_invar } v_0\ D_0\ (l,\ p,\ D,\ pE);\ \textit{invar } v_0\ D_0\ (\textit{pop } (p,\ D,\ pE));\\ p \neq [];\ pE \cap \textit{last } p \times \textit{UNIV} = \{\}]\!] \\ \Longrightarrow \textit{cscc\_invar\_ext } (\textit{last } p\ \#\ l,\ \textit{pop } (p,\ D,\ pE))$$

The other operations, i. e. *collapse* and *push*, have not been modified at all, and also do not change the parts of the state that the invariant extension depends on. Thus, proving preservation of the invariant extension for these operations is straightforward. Moreover, the termination argument from the skeleton algorithm can be reused. Finally, we prove the theorem $\textit{compute\_SCC} \leq \textit{compute\_SCC\_spec}$, which states that the SCC algorithm behaves according to its specification. This is straightforward, using the VCG with the invariant preservation lemmas from the skeleton algorithm together with the new ones for the invariant extension.

While the formalization of the skeleton algorithm and the invariants requires about 1300 lines of proof text, the extension to compute SCCs requires only about 300 lines.

3.3 Emptiness Check for GBA

The extension to check for emptiness of GBAs is more complex, but is formalized in the same way. We sketch the extension here very briefly, and refer the reader to the actual formalization [15] for details.

Starting from the skeleton algorithm, we extend the collapse operation to check whether the collapsed c-node contains nodes from all acceptance classes. If so, we break the loop immediately and return the result for non-emptiness. It contains the two sets $\bigcup \textit{butlast } p$ and $\textit{last } p$, which can be used to reconstruct the accepting run: The path reaching the accepting cycle only contains nodes from the first set, the accepting cycle itself only contains nodes from the second set.

The invariant for the outer loop is extended to state that there is no accepting cycle within finished nodes. The invariant of the inner loop is extended to state that there is no accepting cycle over visited edges. The extension of the skeleton algorithm to GBA emptiness check requires about 700 lines of proof text.

4 Implementation using Gabow's Data Structure

In the last section, we described the verification of the abstract path based algorithm. In this section, we describe the refinement to Gabow's data structure, which was already sketched in Section 2.2.

We implement the stack S and the boundary stack B by lists of nodes and natural numbers, respectively. The index map I is implemented as a function from nodes to *node_state option*, where

node_state = DONE | STACK nat

Finished nodes are mapped to *Some DONE*, nodes on the stack are mapped to *Some (STACK j)*, where j is the index of the node in S, and nodes not yet seen are mapped to *None*.

We additionally use the *pending stack* P to store the pending edges. P contains entries of the form *nat* × *node set*. An entry *(j,succs)* means that the edges $\{S!j\} \times succs$ are pending. The pending stack only contains entries with non-empty second component, and the entries are always sorted by first component. Thus, the last entry *(j,succs)* of P contains the pending edges for the last node on S that has pending edges left. By comparing j to *last B*, one efficiently checks whether this node belongs to the last collapsed node on the path. Also, pushing a new node is efficiently implemented by pushing an entry for its successors, if any, onto P. The invariant for Gabow's data structure is, again, formalized as a locale, based on the locale GS, which fixes (S, B, P, I):

locale *GS_invar = GS +*
(∗1∗) **assumes** *sorted B* **and** *distinct B* **and** *set B* ⊆ {*0..<length S*}
(∗2∗) **and** *S≠[]* ⟹ *B≠[]* ∧ *B!0=0* **and** *distinct S*
(∗3∗) **and** *(I v = Some (STACK j))* ⟷ *(j<length S* ∧ *v = S!j)*
(∗4∗) **and** *sorted (map fst P)* **and** *distinct (map fst P)*
(∗5∗) **and** *set P* ⊆ {*0..<length S*}×{*x. x≠{}*}

Intuitively, Line 1 states that the boundary stack is sorted, distinct, and contains valid indexes into S. Line 2 states that a non-empty stack implies a non-empty boundary stack with the first boundary being 0, and that S is distinct. Line 3 states that the index map is consistent with the stack. Finally, Lines 4 and 5 state that the first elements of the pending edge stack are sorted and distinct, and that the pending edge stack contains valid indexes into the stack and no empty successor sets.

To map concrete to abstract states, we define (in the locale GS):

seg_start i ≡ B!i
seg_end i ≡ **if** *i+1 = length B* **then** *length S* **else** *B!(i+1)*
seg i ≡ {*S!j | j. seg_start i ≤ j* ∧ *j < seg_end i*}

p_α ≡ map seg [0..<length B]
D_α ≡ {*v. I v = Some DONE*}
pE_α ≡ { *(u,v) . ∃j I. (j,I)∈set P* ∧ *u = S!j* ∧ *v∈I* }
GS_α ≡ (p_α,D_α,pE_α)

Here, GS_α is the abstraction function, mapping the concrete state (S, B, P, I) (fixed by the locale GS) to its corresponding abstract state. Finally, we define *GS_rel* as the refinement relation induced by GS_α and *GS_invar*. Similarly, we define *oGS_rel* for the state of the outer loop.

Next, we provide concrete versions of the operations and show that they refine their abstract counterparts. For example, for the pop operation, we define (in GS):

pop_impl ≡ **do** {
  $I \leftarrow$ *mark_as_done* (*seg_start* (|*B*| − *Suc 0*)) (*seg_end* (|*B*| − *Suc 0*)) *I;*
  return (*take* (*last B*) *S, butlast B, I, P*) }

Here, *mark_as_done l u* marks the nodes in $\{S!i \mid l \leq i \wedge i < u\}$ as finished. We show the following refinement lemma:

pop_refine: ⟦((*S,B,I,P*), *p, D, pE*) ∈ *GS_rel; p* ≠ []; *pE* ∩ *last p* × *UNIV* = {}⟧
  ⟹ *pop_impl* (*S,B,I,P*) ≤ ⇓$_{GS\_rel}$ (**return** (*pop* (*p, D, pE*)))

After having defined the other operations, and shown similar lemmas for them, we finally define *skeleton_impl* and show that it refines *skeleton*. Exploiting the automation provided by the Refinement Framework, this is straightforward:

theorem *skeleton_impl* ≤ ⇓$_{oGS\_rel}$ *skeleton*
  unfolding *skeleton_impl_def skeleton_def*
  by (*refine_rcg skeleton_refines*)
    (*vc_solve* (*nopre*) *solve: asm_rl I_to_outer simp: skeleton_refine_simps*)

4.1 Refinement of SCC Computation and GBA Emptiness Check

In order to implement the actual algorithm for computing a list of SCCs, the only thing we have to add is a function that builds a set out of the last segment of S. This set is added to the output list upon finishing a c-node. This extension is straightforward, and, all together, the formalization requires less than 100 lines. It completely reuses what is already proved for the skeleton algorithm.

The refinement for the GBA emptiness check is more complicated. For better manageability, it is split into two steps: In the first step, the sets of acceptance classes for each c-node on the path are explicitly maintained in a list A, and the check after the collapse operation is refined to use A. Proving refinement is quite simple as only redundant information is added. In the second step, we refine the algorithm to use Gabow's data structure.

For a clearer structure of the formalization, we decided to define new constants for the operations of the emptiness check algorithm, which use the operations from the skeleton, and add the new functionality for keeping track of A. For the collapse operation, we have to compute *idx_of v* twice: Once in the original collapse operation, and a second time for updating A. Thus, we add another refinement step to refine this operation to an optimized version that computes *idx_of v* only once. Note that this refinement is limited to the collapse operation, and does not affect the overall proof.

All together, the implementation of the emptiness check requires 900 lines of proof text. Using the refinement framework, we have broken down the formalization into small, manageable steps: First, we introduced the list A, then we introduced Gabow's data structure, and for the collapse operation, we added an additional optimization step. If we would have done all these in one big step, we would have ended up with a complicated formalization, which is hard to maintain or change. In contrast, our approach has already proven its flexibility: In a first version, we had only formalized the inner loop of the algorithms. Thus, we could only handle graphs where all nodes are reachable from a single node v_0. Later, we added the outer loop without any major problems.

5 Refinement to Efficient Standard ML Code

In order to generate efficient SML code, we first have to decide for the data structures used to implement the stacks S, B, and P, and the map I. We resort to the large selection of verified data structures provided by the Isabelle Collection Framework [14,16]: For the stacks, we use arrays with dynamic resizing, which have an amortized constant time per operation. For the index map I, we also use an array, assuming the nodes to be natural numbers.[3] Once we have fixed the refinement relations, the Autoref tool [18] synthesizes and proves correct the refined versions of the algorithms automatically. Finally, the code generator [11,12] is used to extract the SML code.

For computing the SCCs, we encode the output as a list of distinct lists. This is adequate as we only prepend items to the output, which is a constant time operation. Moreover, when extracting an SCC from the path, we know that each node occurs at most once on the path. Thus, building a distinct list of these nodes can be done in linear time. When adding a corresponding assertion to the program, the Autoref tool performs this optimization automatically.

For the GBA emptiness check, we represent the acceptance classes C by natural numbers in the range $\{0..<|C|\}$, and use bitvectors to implement the sets of acceptance classes that are stored on the stack.

6 Benchmarks

We have benchmarked the extracted code against a reference implementation of Gabow's algorithm in Java, taken from Sedgewick and Wayne's book on algorithms [27] and slightly adapted to work with an explicit set of root nodes. To produce the input graphs, we used a random graph generator, also taken from [27], and let it produce graphs with the number of edges $|E|$ in the range from 10^5 to 10^6. Each graph has $|V| = \lfloor 6\sqrt{|E|} \rfloor$ nodes, and $\lfloor \frac{|V|}{10} \rfloor$ SCCs. The results of the benchmark are displayed in Figure 1, using a log-log scale where

[3] This choice is adequate for comparison with the Java reference implementation, which also uses an array for I. For model checking, a hash table is more adequate, which can be used by changing only a few lines of the formalization.

the y-axis is the required time to determine the lists of SCCs in milliseconds, and the x-axis is the number of edges. We compiled the extracted code with PolyML 5.5.1 and MLton 20130715, and used Java 7 for the reference implementation. All tests where performed on an x86/64 linux platform.

All implementations scale linearly with the graph size. On first glance, our implementation looks slightly faster than the Java reference implementation. However, performance in Java is very unpredictable, in particular due to just in time compilation taking place in parallel to program execution. Thus, we have modified the Java implementation to allow it to "warm up" with a few graphs, before we started the measurement. This ensures that the JIT compiler has gathered enough statistics about the program and actually finished compilation. The "Java*" - line displays the results for the modified program, which are roughly one order of magnitude better, but required some non-obvious modification to the Java program, exploiting intimate knowledge of the JIT compiler.

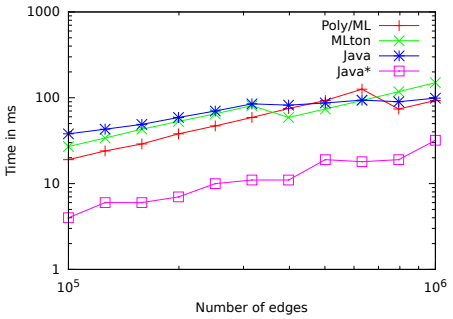

Fig. 1: Benchmarking of extracted code against Java reference Implementation

7 Conclusion

We have presented a verification of two variants of Gabow's algorithm: Computation of the strongly connected components of a graph, and emptiness check of a generalized Büchi automaton. We have extracted efficient code with a performance comparable to a reference implementation in Java.

We have modularized the formalization in two directions: First, we share most of the proofs between the two variants of the algorithm. Second, we use a stepwise refinement approach to separate the algorithmic ideas and the correctness proof from implementation details. Sharing of the proofs reduced the overall effort of developing both algorithms. Using a stepwise refinement approach allowed us to formalize an efficient implementation, without making the correctness proof complex and unmanageable by cluttering it with implementation details.

Our development approach is independent of Gabow's algorithm, and can be reused for the verification of other algorithms.

Current and Future Work Currently, we are integrating our algorithm into the Cava [8] verified LTL model checker. We expect a considerable improvement in checking speed. Moreover, fine-tuning of the used data structures, e. g. using bit vectors and machine words instead of the currently used arbitrary precision integers may give some performance improvements. The foundations for using those low-level data structures in Isabelle/HOL have recently been laid [20].

Acknowledgement We thank the anonymous reviewers for their helpful comments.

References

1. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
2. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNAI, vol. 4108, pp. 31–43. Springer (2006)
3. Büchi, J.R.: On a Decision Method in Restricted Second-Order Arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science. pp. 1–11. Stanford University Press (1962)
4. Cheriyan, J., Mehlhorn, K.: Algorithms for dense graphs and networks on the random access computer. Algorithmica 15(6), 521–549 (1996)
5. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design 1(2/3), 275–288 (1992)
6. Couvreur, J.M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. In: Proc. of SPIN. pp. 169–184. Springer (2005)
7. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976), ch. 25
8. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.G.: A fully verified executable LTL model checker. In: CAV, LNCS, vol. 8044, pp. 463–478. Springer (2013)
9. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Information Processing Letters 74(3–4), 107 – 114 (2000)
10. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. 345(1), 60–82 (Nov 2005)
11. Haftmann, F.: Code Generation from Specifications in Higher Order Logic. Ph.D. thesis, Technische Universität München (2009)
12. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Functional and Logic Programming (FLOPS 2010). LNCS, Springer (2010)
13. Krauss, A.: Recursive definitions of monadic functions. In: Proc. of PAR. vol. 43, pp. 1–13 (2010)
14. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Proc. of ITP. LNCS, vol. 6172, pp. 339–354. Springer (2010)
15. Lammich, P.: Formalization of Gabow's algorithm, `http://www21.in.tum.de/~lammich/isabelle/gabow`, Isabelle Theories
16. Lammich, P.: Collections Framework. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Collections.shtml` (Dec 2009), formal proof development
17. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs. `http://afp.sf.net/entries/Refine_Monadic.shtml` (2012), formal proof development
18. Lammich, P.: Automatic data refinement. In: Interactive Theorem Proving, LNCS, vol. 7998, pp. 84–99. Springer Berlin Heidelberg (2013)
19. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Proc. of ITP. LNCS, vol. 7406, pp. 166–182. Springer (2012)
20. Lochbihler, A.: Native word. Archive of Formal Proofs (Sep 2013), `http://afp.sf.net/entries/Native_Word.shtml`, Formal proof development
21. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press (1997)

22. Munro, I.: Efficient determination of the transitive closure of a directed graph. Information Processing Letters 1(2), 56 – 58 (1971)
23. Neumann, R.: A framework for verified depth-first algorithms. In: McIver, A., Höfner, P. (eds.) Proc. of the Workshop on Automated Theory Exploration (ATX 2012). pp. 36–45. EasyChair (2012)
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
25. Purdom, Paul, J.: A transitive closure algorithm. BIT Numerical Mathematics 10(1), 76–94 (1970)
26. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Three SCC-based emptiness checks for generalized Büchi automata. In: Logic for Programming, Artificial Intelligence, and Reasoning, LNCS, vol. 8312, pp. 668–682. Springer (2013)
27. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley Professional (2011), 4th edition
28. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications 7(1), 67–72 (Jan 1981)
29. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
30. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115, 1–37 (1994)
31. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: In Proceedings of the 1st Symposium on Logic in Computer Science. pp. 322–331 (1986)
32. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science. pp. 61–78 (1992)