Fast Verified SCCs for Probabilistic Model Checking

Arnd Hartmanns, Bram Kohlen(✉), and Peter Lammich

University of Twente, Enschede, The Netherlands
b.kohlen@utwente.nl

Abstract. High-performance probabilistic model checkers like the Modest Toolset's mcsta follow the topological ordering of an MDP's strongly connected components (SCCs) to speed up the numerical analysis. They use hand-coded and -optimised implementations of SCC-finding algorithms. Verified SCC-finding implementations so far were orders of magnitudes slower than their unverified counterparts. In this paper, we show how to use a refinement approach with the Isabelle theorem prover to formally verify an imperative SCC-finding implementation that can be swapped in for mcsta's current unverified one. It uses the same state space representation as mcsta, avoiding costly conversions of the representation. We evaluate the verified implementation's performance using an extensive benchmark, and show that its use does not significantly influence mcsta's overall performance. Our work exemplifies a practical approach to incrementally increase the trustworthiness of existing model checking software by replacing unverified components with verified versions of comparable performance.

1 Introduction

Probabilistic model checking [2] is an automated verification technique for system models with randomness, such as Markov decision processes (MDPs). Today's probabilistic model checkers like Prism [26], Storm [22], or the Modest Toolset [17] check MDPs of tens to hundreds of millions of states on common desktop hardware in minutes. This performance is in part achieved by using the "topological approach" [8,18] where the analysis treats every sub-MDP corresponding to a strongly connected component (SCC) in the MDP's state-transition graph separately, solving them in their reverse topological order. The most well-known such method is topological value iteration [8], but the same idea applies to other approaches like using linear programming (LP) solvers, with one linear program generated for each SCC.

As verification tools, probabilistic model checkers are critical software: we use them in the design and evaluation of safety- and performance-critical systems, and rely on them delivering correct results. Yet, they are *not* thoroughly verified

Authors are listed in alphabetical order. This work was funded by NWO grant OCENW.KLEIN.311, the EU's Horizon 2020 research and innovation programme under MSCA grant no. 101008233 (mission), and NWO VENI grant 639.021.754.

É. André and J. Sun (Eds.): ATVA 2023, LNCS 14215, pp. 181–202, 2023.
https://doi.org/10.1007/978-3-031-45329-8_9

themselves: they constitute large trusted code bases (the Modest Toolset, for example, consists of approx. 150 k lines of C# code as of the writing of this paper) developed ad-hoc in academic contexts. In addition to the danger of implementation bugs, unsound algorithms have been used in probabilistic model checking in the past [14], and published pseudocode contains mistakes (e.g. that of the sound value iteration algorithm given in [41]). This calls for the application of verification technology to the probabilistic model checkers themselves.

Replacing a complete model checker's code base by a fully verified one, keeping the original tool's capabilities and performance, is a gigantic task. For this reason, we today only see inefficient fully verified (non-probabilistic) model checkers [6,44], and fully verified certifiers [43,45] that a posteriori establish the correctness of an unverified model checker's result. The latter, however, require cooperation from the unverified tool to produce an additional compact certificate, and the existence of an efficient certificate verification procedure.

In this paper, we exemplify a third, practical approach with an emphasis on performance: to incrementally replace an existing tool's unverified code by verified implementations of verified algorithms, component-by-component. In order to avoid performance regressions, the new components must use the original data structures and interfaces, and the verifier must work with or generate efficient imperative implementations. With every step, the trusted code base shrinks, and the trustworthiness of the larger tool increases. At every step, the new verified code can be thoroughly benchmarked and optimised.

We apply this approach to the mcsta probabilistic model checker of the Modest Toolset. It consists of components with well-defined interfaces, ranging from input language semantics over state space exploration, graph-based precomputations [12], finding SCCs, end component elimination, and essential states reduction [9] to the actual numeric solution methods like variants of value iteration or linear programming. Of these components, we chose to replace the step of finding SCCs that enables the topological solution methods. This is because (i) it is a critical step for both the performance of the solution method and the correctness of the final result, and (ii) we can reuse parts of an existing formalization [30] of Gabow's SCC-finding algorithm [13], allowing us to focus on the performance and tool integration challenges.

To produce a verified algorithm that works directly on the imperative data structures of mcsta, we use the Isabelle LLVM tool [32,34] that produces verified LLVM code using the Isabelle theorem prover [38]. To keep the abstract algorithmic ideas separate from the actual implementations and data structures, we use a stepwise refinement approach supported by the Isabelle Refinement Framework [35], consisting of four conceptual steps: A correctness proof for general path-based SCC algorithms (Sect. 3), the use of Gabow's particular data structures (Sect. 4), the imperative implementation (Sect. 5), and the generation of LLVM code (Sect. 6). The first two steps are an adaptation of the ideas of the existing verified but slow functional implementation of Gabow's algorithm [30] to prepare for the imperative refinement. The last two steps are entirely new, using new imperative data structures both internally and on the interface to

mcsta. In particular, the algorithm needs to work with mcsta's representation of the graph (in terms of an MDP) and return the information about the found SCCs to mcsta.

To assess the impact of replacing an unverified component of a probabilistic model checker by a correct-by-construction version, we performed an extensive experimental comparison using benchmarks from the Quantitative Verification Benchmark Set [20] (Sect. 7). We found that, by using an imperative implementation and avoiding costly glue code and transformations or copies of the data (e.g. of mcsta's MDP data structures into a more generic graph representation), our verified implementation outperforms the existing implementation of Tarjan's algorithm in mcsta (being around twice as fast) and achieves performance comparable to a manually-optimised unverified C implementation of Gabow's algorithm that we newly built as a comparison baseline (which on average is only a bit faster). This means that we have replaced a unverified algorithm with a faster, provably correct one.

Related Work. Only a few verified model checker implementations exist that can be applied to significant problem sizes: CAVA [6,10] is a fully verified LTL model checker, featuring a fragment of Promela [37] as input language. While able to check medium-size examples in reasonable time, it is much slower than highly optimized unverified tools such as SPIN [25]. Similarly, the fully verified MUNTA model checker [44] for timed automata is still significantly slower than the highly optimized unverified counterpart UPPAAL [5], and the verified IsaSAT solver [11] placed last in the SAT2022 competition [4].

On the other hand, the results of model checking can be certified by a formally verified certifier. This requires the existence of a practical certification mechanism, and the support of the unverified model checker. Formally verified certification tools that work on significant problem sizes exist for e.g. timed automata model checking [43,45] and SAT solving [23,33].

There are some formalizations of Markov decision processes and value iteration in Isabelle/HOL [24] and Coq [42]. However, there is no documentation on extracting executable code from these proofs. Additionally, there is a formalization of value iteration for discounted expected rewards [36] which extracts Standard ML code from the proof. Strongly connected component finding algorithms have been formally verified with various tools, including Isabelle/HOL [30], Coq [39], and Why3 [7]. However, [39] and [7] do not report on extracting executable code from their verification at all, and the code extracted from [30] is roughly one order of magnitude slower than a textbook reference implementation of the same algorithm in Java.

Our replacement of mcsta's unverified SCC-finding implementation by a verified one is part of a larger effort to improve the trustworthiness of the tool, in which we already developed an efficient sound variant of value iteration [19] and proposed a way to avoid floating-point rounding errors with limited performance impact [16], but did not yet apply verification to the tool itself.

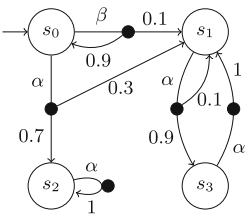


Fig. 1. MDP $\mathcal{M}_{ex}$

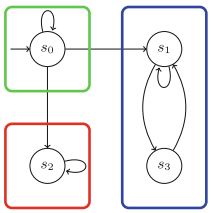


Fig. 2. Graph $G_{ex} \coloneqq (S, E_{\mathcal{M}_{ex}})$ with its SCCs

2 Preliminaries

$\mathbb{R}$ is the set of real numbers; $[0, 1] \subseteq \mathbb{R}$ denotes the real numbers from 0 to 1. For a set X, 2^X is its power set. A *(discrete) probability distribution* over a set X is a function $\mu \colon X \to [0, 1]$ where $\mathit{support}(\mu) \stackrel{\text{def}}{=} \{\, x \in X \mid \mu(x) > 0 \,\}$ is finite and $\sum_{x \in \mathit{support}(\mu)} \mu(x) = 1$. $\mathit{Dist}(X)$ is the set of probability distributions over X.

2.1 Markov Decision Processes

Our work is implemented in the context of the probabilistic model checker mcsta. One of the core problems in probabilistic model checking are *reachability properties*, where an optimal solution regarding some metric towards reaching a set of target states is computed in a *Markov decision process (MDP)* [40].

Definition 1. *An MDP is a triple* $\mathcal{M} = (S, s_0, \mathit{prob})$ *of a finite set of* states S *with* initial state $s_0 \in S$ *and a* transition function $\mathit{prob} \colon S \to 2^{\mathit{Dist}(S)}$.

An MDP moves in discrete time steps. In each step, from current state s, one distribution $\mu \in \mathit{prob}(s)$ is chosen non-deterministically and sampled to obtain the next state. A *policy* resolves the non-determinism in an MDP by choosing one probability distribution for each state. The goal is to find a policy that maximizes/minimizes the probability or expected reward to reach a target state.

Example 1. Figure 1 shows an MDP with states s_0 to s_3 where s_0 is the initial state. Using distribution α, we go to state s_1 with probability 0.3 and to s_2 with probability 0.7. Using distribution β, we go to state s_0 and s_1 with probability 0.9 and 0.1 respectively. The maximal probability to reach state s_1 is 1, achieved by the policy that chooses β until we reach s_1, where it chooses α indefinitely.

A *graph* is a pair $G = (V, E)$ of a set of vertices V connected by edges $E \subseteq V \times V$. E^* is the reflexive transitive closure of E.

Definition 2. *A* strongly connected component *(SCC) of* G *is a set* $U \subseteq V$ *such that* $U \times U \subseteq E^*$ *(it is strongly connected) and* $\forall\, U' \supsetneq U \colon \neg(U' \times U' \subseteq E^*)$ *(it is maximal).*

Given $\mathcal{M} = (S, s_0, \mathit{prob})$, let $E_{\mathcal{M}} \stackrel{\text{def}}{=} \{\, (s, s') \mid \exists\, \mu \in \mathit{prob}(s) \colon s' \in \mathit{support}(\mu) \,\}$. Then $(S, E_{\mathcal{M}})$ is the graph of $\mathcal{M}$. U is an SCC of $\mathcal{M}$ iff U is an SCC of $(S, E_{\mathcal{M}})$.

Example 2. Figure 2 shows the graph G_{ex} of MDP $\mathcal{M}_{ex}$ from Fig. 1. It also shows the three SCCs of G_{ex} outlined in green, blue, and red.

The optimal value (probability or expected reward) of a state (and consequently the decision of the optimal policy) only depends on the optimal values of its successors. Decomposing the MDP into its SCCs and solving the SCCs in reverse topological order guarantees that each state's successors have either been solved to (ϵ-)optimality before, or are being considered in the current SCC. This approach breaks down the MDP into smaller subproblems; if the MDP consists of many similarly sized SCCs, the computation uses much less time and memory than naive methods [8,18]. mcsta currently implements topological LP solving.

2.2 Program Verification Based on Refinement in Isabelle/HOL

To comprehend (and verify) the optimized implementation of an algorithm, we use a stepwise refinement approach. We start with the abstract algorithmic idea describing the essence of the algorithm on the level of manipulating mathematical objects like maps and sets. We then use a series of refinement steps to gradually replace the abstract mathematical objects by actual data structures until we arrive at the executable implementation. In the process, we prove that each refinement step preserves correctness. The steps are typically independent, which helps to keep the overall proof structured and manageable. Different components can be refined independently (e.g. separating data and program refinement), to be assembled at a later stage or used in other algorithms, without re-playing the intermediate steps. A good refinement design is key to a manageable proof, and, as all design choices, requires experience and involves trade-offs.

The Isabelle Refinement Framework (IRF) [35] implements stepwise refinement on top of Isabelle/HOL. It provides a formal notion of programs and refinement, tools like a verification condition generator that facilitate proving, and a library of reusable verified data structures. Its recent LLVM backend [32] supports the generation of LLVM bytecode. In the following, we give a brief overview of the IRF. For an in-depth description, we refer the reader to [32,35].

Programs are modelled by shallow embedding into a nondeterminism error monad $'a\ nres \equiv \mathbf{fail} \mid \mathbf{spec}\ ('a \Rightarrow bool)$. Intuitively, a program fails (**fail**) or nondeterministically returns a result that satisfies P (**spec** P). The **return** x combinator returns the only result x, and the bind combinator **do** $\{x \leftarrow m;\ f\ x\}$ selects a result x of m, and then executes $f\ x$. A program fails if there is at least one nondeterministic possibility to fail. Thus, **do** $\{x \leftarrow m;\ f\ x\}$ fails if m fails, or if f fails for at least one possible result of m. The **assert** P combinator does nothing (i.e. returns a unit value) if P holds, and fails otherwise. The IRF provides further combinators and syntax for control flow.

A (concrete) program m' *refines* an (abstract) program m (written $m' \leq m$) if every possible result of m' is a result of m. Also, $m' \leq \mathbf{fail}$ and $\mathbf{fail} \not\leq \mathbf{spec}\ P$: the intuition is to assume that the abstract program does not fail, i.e. the concrete program can do anything in case the abstract program fails. We lift a refinement relation R between concrete and abstract data to program m using $\Downarrow R\ m$, which returns all concrete results that are related to some abstract result of m.

Example 3. We use the IRF to refine a pop operation of a stack:

$(*1*)$
$ssel :: 'a\ set \Rightarrow ('a \times 'a\ set)\ nres;\ lpop :: 'a\ list \Rightarrow ('a \times 'a\ list)\ nres$
$ssel\ s \equiv \textbf{do}\ \{\ \textbf{assert}\ s{\neq}\{\};\ \textbf{spec}\ \lambda\ (x,s').\ x{\in}s \wedge s'{=}s{-}x\ \}$
$lpop\ l \equiv \textbf{do}\ \{\ \textbf{assert}\ l{\neq}[];\ \textbf{return}\ (last\ l,\ butlast\ l)\}$
$'a\ da = 64\ word \times 64\ word \times 'a\ ptr\ \ (*\ length,\ capacity,\ pointer\ to\ array\ *)$
$apop :: 'c\ da \Rightarrow ('c \times 'c\ da)\ llM$
$apop\ (l,c,a) \equiv \textbf{do}\ \{\ l{\leftarrow}ll\_sub\ l\ 1;\ p{\leftarrow}ll\_ofs\_ptr\ a\ l;\ r{\leftarrow}ll\_load\ p;\ \textbf{return}\ (r,(l,c,a))\ \}$

$(*2*)$
$R_{ls} :: 'a\ list \times 'a\ set;\ R_{ls} \equiv \{\ (xs,set\ xs)\ |\ xs.\ distinct\ xs\ \}$
$(l,s) \in R_{ls} \implies lpop\ l \leq \Downarrow(Id \times R_{ls})\ (ssel\ s)\ \ (short:\ \ lpop,ssel : R_{ls} \to Id \times R_{ls}\ )$
$A_{da} :: ('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a\ list \Rightarrow 'c\ da \Rightarrow assn;\ (*\ definition\ elided\ *)$
$apop,\ lpop : (A_{da}\ e)^d \to e \times A_{da}\ e$

$(*3*)$
$A_{set} :: ('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a\ set \Rightarrow 'c\ da \Rightarrow assn;\ A_{set}\ e \equiv A_{da}\ e\ O\ R_{ls}$
$apop,\ ssel : (A_{set}\ e)^d \to e \times A_{set}\ e$

First (1) we define functions to remove an arbitrary element from a non-empty set (*ssel*) and to pop the last entry of a non-empty list and dynamic array (*lpop*/*apop*). Then (2) we define the refinement relation R_{ls} between distinct lists and sets. We show for related arguments $(l,s){\in}R_{ls}$ that all possible outputs of *lpop l* and *ssel s* are related through $Id \times R_{ls}$, i.e. the first elements of the pair are equal, and the second elements are related by R_{ls}. We also introduce a shortcut notation that elides the parameter names. We then define a refinement between dynamic arrays and lists: $A_{da}\ e\ l\ d$ is a separation logic assertion that states that the dynamic array d contains the elements from list l, where the elements themselves are refined by e[1]. The annotation ^{d} on a parameter refinement indicates that this refinement is no longer valid after execution of the concrete program (typically the data has been destructively updated). Finally (3) we compose (O) our assertion with a relation to show that *apop* refines *ssel*.

2.3 Existing Formalisation of Gabow's Algorithm

Our work builds on an existing formalisation [30] of Gabow's algorithm [13]. That formalisation uses an early version of the IRF [29], targeting purely functional SML code; it is an order of magnitude slower than a reference implementation in Java. While incompatible with our goal of creating a fast drop-in replacement to be used directly on the mcsta data structures, we can reuse parts of the existing abstract formalisation. In the following sections, we indicate the parts we reused, referring to the existing work as the *original formalization*.

[1] Note that the order of the refinement relations ($('a \times 'c)\ set$) is different from the assertions ($'c \Rightarrow 'a \Rightarrow assn$).

3 Abstract Path-Based Algorithm

Gabow's SCC-finding algorithm is a *path-based* algorithm: It maintains a path from the start node that is extended in each iteration via an edge from the path's tail. When the edge leads back onto the path, the resulting cycle is collapsed into a single node. When there are no more outgoing edges left, the last node corresponds to an SCC and is removed from the path. We follow a similar design approach as the original formalization: We first define a *skeleton* algorithm that performs the DFS, but discards the found SCCs. We then reuse parts of the skeleton to define an actual SCC-finding algorithm. This technique makes the proof more modular, factoring out general properties of Gabow-style algorithms [30].

3.1 The Skeleton Algorithm

```
1  skeleton ≡ do {
2    let D0 = {};
3    r ← foreach outer_invar V0 (λv0 D0. do {
4      if v0∉D0 then do {
5        s ← initial v0 D0;
6        (p,D,pE,vE) ← while (invar v0 D0) (λ(p,_). p ≠ []) (λ(p,D,pE,vE). do {
7            (vo,(p,D,pE,vE)) ← select_edge (p,D,pE,vE);
8            case vo of
9              None ⇒ do { return (pop (p,D,pE,vE)) }
10           | Some v ⇒ do {
11               if v ∈ ⋃(set p) then do { return (collapse v (p,D,pE,vE)) }
12               else if v∉D then do { push v (p,D,pE,vE) }
13               else do { return (p,D,pE,vE) }
14             }
15         }) s;
16         return D
17       } else return D0
18     }) D0;
19     return r}
```

The outer loop of the skeleton (l. 3) iterates over all nodes *V0*. The inner loop performs a DFS, maintaining a program state consisting of a segmented path *p :: 'v set list*, the "done" nodes *D :: 'v set*, pending edges *pE :: 'v multiset*, and visited edges *vE :: 'v set*. The operations perform changes to that state, e.g.

definition *collapse v (p,D,pE,vE)* ≡
let *i=idx_of p v; p = take i p @ [⋃(set (drop i p))]* **in** *(p,D,pE,vE)*

where *@* appends two lists, *idx_of p v* returns the index of *v* in *p* (which we prove to exist) and *take i p*/*drop i p* yields/discards the first *i* elements of *p*. In essence this operation combines all segments from index *i* onwards. For the other operations we refer to the supplementary material.

In each step, the skeleton selects a pending edge from the last segment of the path (l. 7). If no such edge exists (l. 9), the last segment is an SCC. In the

Fig. 3. The state of the path based SCC algorithm before and after a collapse step. The dotted nodes and edges are not yet visited by the algorithm.

skeleton, we pop the last segment from the path. Later, we will perform some extra work to mark the found SCC. Otherwise (l. 10), if the selected edge goes back into the path (l. 11), we have found a cycle, and collapse all its nodes into a single segment. If the edge leads to a new node (l. 12), we add this node to the current path. Otherwise (l. 13), the edge leads to a done node and we ignore it.

Example 4. Figure 3 visualizes a step in the program where $p = [\{0\}, \{1,3\}, \{2\}]$, $D = \{\}$, $pE = \{(1,4),(2,3)\}$ and $vE = \{(0,3),(3,1),(1,3),(1,2)\}$. Then, exploring e.g. back edge $(2,3)$ collapses all segments in that cycle. Now $p = [\{0\}, \{1,3,2\}]$, $D = \{\}$, $pE = \{(1,4)\}$ and $vE = \{(0,3),(3,1),(1,3),(1,2),(2,3)\}$.

The original formalization only supports successor functions that return a set of nodes. However, the successor function on the graph data structure of mcsta is more efficient if we allow duplicates in the list of successors. This can cause the same edge to be explored multiple times, which, however, does not matter as the target node is marked as done on successive explorations. To later allow this implementation, we had to change pE to be a multiset of pending edges in the abstract algorithm. This revealed a problem in the original formalization: the set of visited edges was defined implicitly, but a multiset of pending edges does not allow for such implicit representation. We solved this by explicitly introducing vE into the abstract state, which even simplified the existing proofs. Note that vE is a *ghost variable*, i.e. no other parts of the state depend on it. Thus, we can easily eliminate it in the next refinement step (Sect. 4.1).

Invariants. To define the invariants, we use Isabelle's locale mechanism [3] that allows us to define named hierarchical contexts with fixed variables and assumptions. First, we define a set of initial nodes $V0$. Then, we define finite graphs as an adjacency function *E_succ* that maps each node to a list of adjacent nodes and an according abstraction E_α that returns the set of edges induced by *E_succ*:

locale *fr_graph* = **fixes** *V0* :: *'v set* **and** *E_succ* :: (*'v* ⇒ *'v list*)
assumes *1: finite* (*E_α** `` *V0*)

The invariant of the outer loop extends *fr_graph*, adding the loop's state (*it*,*D*):

locale *outer_invar_loc* = *fr_graph V0 E_succ*
for *V0* **and** *E_succ* :: (*'v* ⇒ *'v list*) + **fixes** *it* :: *'v set* **and** *D* :: *'v set*
assumes *1: it* ⊆ *V0* **and** *2: V0* − *it* ⊆ *D*
and *3: D* ⊆ *E_α** `` *V0* **and** *4: E_α* `` *D* ⊆ *D*

The invariant guarantees that (1) the nodes we still have to iterate over (it) are in $V0$, (2) the nodes we already have iterated over ($V0 - it$) are done, (3) done nodes are reachable, and (4) done nodes can only reach other done nodes. The invariant *invar_loc* of the inner loop is defined using the same locale construct, but with more extensions. The invariant states that: all nodes within a segment of path p are mutually reachable and segments are topologically ordered; done nodes remain done, are reachable, and only reach other done nodes; edges in pE start in p; and visited edges lead to segments that are topological successors of the source segment. Furthermore, we added restrictions to the new set of visited edges. These state that: edges from done nodes are visited; visited edges only exist between done and path nodes; and unvisited edges from p are pending.

The termination proof of the original formalization was more involved. But using vE we were able to simplify it significantly: Each iteration of the inner loop decreases the lexicographic ordering of the number of unvisited edges, pending edges, and length of the path. Using the IRF's verification condition generator (VCG), we prove that every operation preserves the invariant and decreases the termination ordering. Equipped with these lemmas, the VCG can automatically show that the skeleton terminates and preserves the invariant.

3.2 Abstract SCC-Finding Algorithm

We can then refine the algorithm to also compute a list *l::'v set list* of the SCCs in topological order. Formally:

$scc\_set \equiv \{scc.\ is\_scc\ E\_\alpha\ scc \wedge scc \subseteq E\_\alpha^*\ ``\ V0\}$
$ordered\ l \equiv (\forall i\ j.\ i < j \longrightarrow j < length\ l \longrightarrow l!i \times l!j \cap E\_\alpha^* = \{\})$
$compute\_SCC\_spec \equiv \textbf{spec}\ (\lambda l.\ set\ l = scc\_set \wedge ordered\ l)$

For this, we add a list l of discovered SCCs to the algorithm's state and amend the pop function to add the identified SCC to that list. We call that new algorithm *compute_SCC*. To prove it correct, we extend the invariant of both the outer and inner loop by the statement that l contains exactly the SCCs of the done part of the graph, in topological order (definition elided). With this extension, and reusing the lemmas we have already proved for the skeleton, it is straightforward to show:

theorem *compute_SCC_correct: compute_SCC* $\leq$ *compute_SCC_spec*

4 Formalizing Gabow's Algorithm

The main challenge of path-based approaches is finding efficient data structures to capture the segments. Gabow's data structure exploits the behaviour of the DFS: it stores the path as a stack of nodes, in the order they are visited. Adjacent nodes in the path are in the same SCC or in a topological successor/predecessor, such that a list of boundary indices can be used to encode the segmentation.

4.1 The Skeleton of Gabow's Algorithm

To refine our algorithm to use Gabow's data structure, we again work in two steps: We first refine the skeleton, then reuse this refinement for an actual SCC-finding algorithm. Gabow's data structure uses three stacks and a map to represent the state: The *sequence* stack $S :: 'v\ list$ contains the states on the path in the order they were first visited. The *boundaries* stack $B :: nat\ list$ contains natural numbers representing indices (or bounds) on stack S: all nodes on S between subsequent entries in B form a segment, the last entry of B being the start index of the last segment. The *working list* $P :: ('v \times 'v\ list)\ list$ contains a tuple of nodes on the path and a nonempty list of pending successors. Finally, the node state map $I :: 'v \Rightarrow node\_state\ option$ maps nodes to *node states*:

datatype *node_state* = *STACK nat* | *DONE nat*

$I\ v = None$ indicates that node v has not yet been discovered, $Some\ (STACK\ i)$ indicates that v is on the sequence stack S at index i, and $Some\ (DONE\ j)$ indicates that v is done and belongs to SCC number j. Note that we do not use j in the skeleton, but already add it to *node_state* for convenience.

Similarly to the abstract algorithm, the operations perform changes to the concrete program state, e.g.

definition *collapse_impl_fr* (*S,B,I,P*) *v* ≡ **do** {
  i←*idx_of_impl* (*S,B,I,P*) *v*; **assert** (*i+1* ≤ *length B*);
  let *B* = *take* (*i+1*) *B*; **return** (*S,B,I,P*) }

where *idx_of_impl* implements *idx_of* through a lookup using I and B. For the other implementations we refer to the supplementary material.

Data Structure Invariants. The invariant *oGS_invar* makes sure that the stack is empty on the outer loop. *GS_invar* for Gabow's data structure remained largely unchanged w.r.t. the original formalization: it ensures that B is sorted, distinct, and points to a node on S; as long as there are nodes in S, there are bounds in B starting at 0; I specifies that node v lies at index j in S; parent nodes in P are also in S and have unprocessed successors; parent nodes in P are distinct and sorted by their index in S. We added that S consists only of reachable nodes ($set\ S \subseteq (E\_\alpha^*\ ``V0")$). While this is not required to show the correctness of the data structure at this abstraction level, it comes in handy to show that the length of the stack is bounded when we refine the indexes to 64-bit machine words in the next step. This is a recurring design pattern: some assertions that are only required for a concrete refinement step are most easily proved already on the abstract level.

Example 5. A possible encoding of Fig. 3 in Gabow's data structure is $S = [0,3,1,2]$, $B = [0,1,3]$, I: $[0 \mapsto Some\ (STACK\ 0), 1 \mapsto Some\ (STACK\ 2), 2 \mapsto Some\ (STACK\ 3), 3 \mapsto Some\ (STACK\ 1), 4 \mapsto None]$, $P = [(1,[4]),(2,[3])]$. Then, back edge (2,3) is explored. We pop that entry in P, i.e. $P = [(1,[4])]$. $I(3) = Some\ (STACK\ 1)$ so we pop B until we reach $v \leq 1$; B becomes $[0,1]$.

Iterators. We implement P as a stack containing pairs of a node of the graph and an iterator over its successors. We also considered implementing P as a stack of stacks, where the inner stacks contain the unvisited successors. This was slower in practice as it requires more memory allocations and deallocations. While the iterator is an implementation detail of the data structure that we consider in Sect. 5, we reason about iterators here because we expect that reasoning about the link between the iterators and the graph in separation logic in the next refinement layer would be more complex. We define an iterator as a tuple of a node u and an index ci representing the ci-th index of $E\_succ\ u$. We let $succ\_count\ u \equiv length\ (E\_succ\ u)$ and define five operations for the iterator:

$index\_begin\ u = (u,0) \quad get\_state = \lambda\ (u,ci).\ u$
$successor\_at = \lambda\ (u,ci).\ (E\_succ\ u)\ !\ ci$
$has\_next \equiv \lambda\ (u,ci).\ Suc\ ci < succ\_count\ u \quad next\_index = \lambda\ (u,ci).\ (u,ci + 1)$

index_begin creates an iterator pointing to the start of the iteration sequence; *get_state* returns the source node of the iterator (in other words the state whose successors we iterate over); *successor_at* returns the element that the iterator points to; *has_next* checks if there exists a next element in the sequence (in our case it checks there are unprocessed successors left); and *next_index* updates the iterator to point to the next element in the sequence.

Refinement Relation. We connect Gabow's data structure to the abstract program state via an abstraction function:

$seg\_start\ i \equiv B!i \quad seg\_end\ i \equiv \textbf{if}\ i+1 = length\ B\ \textbf{then}\ length\ S\ \textbf{else}\ B!(i+1)$
$seg\ i \equiv \{S!j \mid j.\ seg\_start\ i \le j \wedge j < seg\_end\ i\} \quad remaining\_successors$
$= (\lambda\ (u,ci).\ map\ (\lambda\ ci'.\ successor\_at\ (u,ci'))\ [ci..<succ\_count\ u])$
$edges\_of\_succs = (\lambda\ (u,ci).\ map\ (\lambda v.\ (u,v))\ (remaining\_successors\ (u,ci)))$
$p\_\alpha \equiv map\ seg\ [0..<length\ B] \quad D\_\alpha \equiv \{v.\ \exists\ i.\ I\ v = Some\ (DONE\ i)\}$
$pE\_\alpha = mset\ (concat\ (map\ edges\_of\_succs\ P))$
$GS\_\alpha \equiv (p\_\alpha,D\_\alpha,pE\_\alpha) \quad oGS\_\alpha\ I \equiv \{v.\ \exists\ i.\ I\ v = Some\ (DONE\ i)\}$

Here, $map\ f\ xs$ returns a list in which each element in xs is mapped using f, *mset* turns a list into a multiset and *concat* concatenates a list of lists into a single list. This reconstructs the p, D, and pE parts of the abstract state from the concrete state. The vE part is a ghost variable and remains unconstrained in the refinement relation. We define:

$GS\_rel \equiv \{\ (c,(p,D,pE,vE))\ .\ (c,(p,D,pE)) \in br\ GS\_\alpha\ (GS\_invar\ V0\ E\_succ)\ \}$
$oGS\_rel \equiv br\ oGS\_\alpha\ (oGS\_invar\ V0\ E\_succ)$

Here, $br\ \alpha\ Inv \equiv \{(c,\ \alpha\ c) \mid c.\ Inv\ c\}$ builds a relation from an abstraction function and invariant.

Refinement Proof. We first show that the concrete operations refine the corresponding abstract ones, e.g.

lemma *collapse_refine:* $(s,(p,D,pE,vE)) \in GS\_rel \wedge (v,v') \in Id \wedge v' \in \bigcup(set\ p)$
$\implies$ *collapse_impl_fr* $v\ s \leq \Downarrow GS\_rel\ (RETURN\ (collapse\ v'\ (p,D,pE,vE)))$

We proceed analogously for the other operations. This allows us to show that the concrete inner loop refines the abstract inner loop. This works similarly for the outer loop. Finally, we get the following theorem:

theorem *skeleton_impl_refine:* $skeleton\_impl \leq \Downarrow_o GS\_rel\ skeleton$

4.2 Gabow's SCC-Finding Algorithm

We implement the list l of SCCs in the abstract algorithm by the length i of the list and the node state map: the nodes of the SCC $l!j$ have state $Some\ (DONE\ j)$:

$SCC\_at\ I\ j \equiv \{v.\ I\ v = Some\ (DONE\ j)\}$
$SCC\_\alpha\ (i,I) \equiv map\ (SCC\_at\ I)\ [0..<i]$

locale *GSS_invar_ext* $= \ldots +$
assumes *1:* $j < i \implies SCC\_at\ I\ j \neq \{\}$ **and** *2:* $j \geq i \implies SCC\_at\ I\ j = \{\}$
assumes *3: finite* $(SCC\_at\ I\ j)$ **and** *4:* $I\ v \neq None \implies v \in E\_\alpha^*\ ``V0$

locale *SCC_invar* $=$ *GSS_invar_ext* $+$ **assumes** *5:* $I\ v \neq Some\ (STACK\ i)$
$SCC\_rel \equiv br\ SCC\_\alpha\ SCC\_invar$

The invariant ensures that (1) every index $j < i$ is assigned to a non-empty SCC, and (2) no indices $j \geq i$ have been assigned. Moreover, (3) SCCs are finite and (4) only assigned to reachable nodes. During the outer loop, and for the representation of the returned result (*SCC_rel*), we additionally know (5) that the stack is empty.

The algorithm *compute_SCC_impl* adds the counter i to the skeleton algorithm. Reusing the lemmas from refining the skeleton algorithm, it is straightforward to show

theorem *compute_SCC_impl_refine:*
$compute\_SCC\_impl \leq \Downarrow SCC\_rel\ compute\_SCC$

That is, our new algorithm returns a pair (i, I) that represents the topologically ordered list of SCCs returned by the abstract algorithm *compute_SCC*.

5 Refinement to LLVM

We now make the step from our model of Gabow's algorithm with abstract data types (*compute_SCC_impl*) to a model of an LLVM implementation with concrete LLVM data types along mcsta's interface (*Modest_compute_SCC_impl*). At this point, we depart from the path taken by the original formalisation.

Some of the data structures we refine to are standard and well-supported by the IRF library: we represent nodes and indices as 64-bit words, use dynamic arrays for the stacks S, B, and P in the algorithm state, and represent I by an *array map*: an array of node states indexed by the nodes. In this section, we highlight the two most interesting refinements.

5.1 Node State

Recall (cf. Sect. 4.1) that the node state is either *None*, *STACK* j, or *DONE* k, where j is an index into the stack S, and k is the number of the SCC that the node belongs to. For a graph with N nodes, we have $j, k < N$. This gives us a straightforward encoding of node states into 64-bit words: *None* becomes -1, *STACK* j becomes j, and *DONE* k becomes $k + N$. While certainly not optimal, this encoding is easy to realise with the IRF standard library. We consider the incurred graph size bound of $N < 2^{62}$ to be sufficient.

5.2 MDP Graph Data Structure

Performance-wise, it is crucial that our algorithm works on the state space (MDP) data structure provided by mcsta, rather than copying to its own data structure. mcsta encodes a state (node) as a 64-bit word $< N$, where N is the number of states in the MDP. It represents the graph structure of the MDP by three arrays St, Tr, and Br. Each element of St indicates an interval in the Tr array, describing the outgoing transitions (i.e. distributions) of a state. Similarly, Tr represents intervals in the Br array, describing the outgoing branches (i.e. elements of the distribution's support) of the transition. Finally, the Br array contains the target nodes of the branches. The intervals in St and Tr are encoded as a 20-bit length and 44-bit start index, packed into a single 64 bit word.

The iterator on the graph is independent of mcsta. We use a structure for the iterator consisting of five 64-bit words (v, tc, te, bc, be). v represents the state, tc and te represent the current and last index of the iteration sequence to Tr and bc and be represent the current and last index to Br.

Example 6. By representing the bit-packing as a tuple of natural numbers we have that $St = [(2,0),(1,2),(1,3),(1,4)]$, $Tr = [(2,0),(2,2),(2,4),(1,6),(1,7)]$ and $Br = [1,2,0,1,1,3,2,1]$ encodes G_{ex} from Fig. 2. State s_1 (at index 1) has 1 transition at index 2 (derived from $St[1] = (1,2)$). This transition has 2 branches starting at index 4 (derived from $Tr[2] = (2,4)$). The successors of s_1 are thus s_1 (as $Br[4] = 1$) and s_3 (as $Br[5] = 3$). The iterator $(v, tc, te, bc, be) = (1,0,1,1,2)$ points to s_3. We observe $v = 1$ which means we consider a successor of state s_1. We also observe that $tc = 0$ which means that the iterator points to transition index 0. We also remember that the transition sequence for this state starts at index 2 ($St[1] = (1,2)$) which we add to tc. So the index points to transition 2. Lastly, we observe that $bc = 1$, which is also relative. The branch sequence for transition 2 starts at 4 ($Tr[2] = (2,4)$) which we add to bc. So the iterator points to branch 5. Since $Br[5] = 3$ the iterator points to state s_3.

We have to implement the graph and the iterator with its operations from Sect. 4. We choose a two-step approach, following a similar structure as in Example 3. We model the graph as $mg_1 :: (nat \times nat)\ list \times (nat \times nat)\ list \times nat\ list$ and the iterator as $it_1 :: (nat \times nat \times nat \times nat \times nat)$:

$R_{mg1}\ N :: (mg_1 \times ('v \Rightarrow 'v\ list))\ set;\ R_{mg1}\ N \equiv br\ mg\_\alpha\ (mg\_invar\ N)$

$R_{it1} :: (it_1 \times ('v \Rightarrow 'v\ list))\ set;\ R_{it1} \equiv br\ it\_\alpha\ (it\_invar)$
$succ\_at_1 :: mg_1 \Rightarrow it_1 \Rightarrow nat\ nres;\ (definition\ elided)$
$succ\_at_1,\ successor\_at : R_{mg1}\ N \rightarrow R_{it1} \rightarrow Id$

Refinement relation $R_{mg1}\ N$ uses abstraction functions mg_α and mg_invar (definitions elided) that encode that we have N states, indices are in bounds, and intervals do not overlap, which ensures that the numbers of successors are bounded by N. Refinement relation R_{it1} uses abstraction functions it_α and it_invar (definitions elided) which encode that the iterator is valid for the given graph structure and is within bounds. Function $succ\_at_1$ refines $successor\_at$ w.r.t.R_{mg1} and R_{it1}, which we prove using the IRF's VCG. The representation of the result does not change (as indicated by the Id relation).

In the next step, we do the bit-packing (A_{bp}), represent nodes by 64-bit words (A_{snat}), and use arrays for the lists (A_{arr}). The output list is refined by

$A_{bp} :: nat \times nat \Rightarrow 64\ word \Rightarrow assn \qquad A_{snat} :: nat \Rightarrow 64\ word$
$A_{arr} :: ('a \Rightarrow 'c \Rightarrow assn) \Rightarrow 'a\ list \Rightarrow 'c\ ptr \Rightarrow assn$

$mg_2 \equiv 64\ word\ ptr \times 64\ word\ ptr \times 64\ word\ ptr$
$A_{mg2} :: mg_1 \Rightarrow mg_2 \Rightarrow assn;\ A_{mg2} \equiv A_{arr}\ A_{bp} \times A_{arr}\ A_{bp} \times A_{arr}\ A_{snat}$

$it_2 \equiv 64\ word \times 64\ word \times 64\ word \times 64\ word \times 64\ word$
$A_{it2} :: it_1 \Rightarrow it_2 \Rightarrow assn;\ A_{it2} \equiv A_{snat} \times A_{snat} \times A_{snat} \times A_{snat} \times A_{snat}$

$succ\_at_2 :: mg_2 \Rightarrow it_2 \Rightarrow 64\ word\ da\ llM\ (def.\ elided,\ generated\ by\ Sepref)$
$succ\_at_2,\ succ\_at_1 : A_{mg2} \rightarrow A_{it2} \rightarrow A_{snat}$

The definition of $succ\_at_2$ and the refinement lemma are synthesised by Sepref, which implements a heuristics to apply data refinements automatically [31].

Finally, we combine the two steps to get the desired refinement from abstract graphs to mcsta's concrete MDP data structure, which we can then use to refine our main algorithm:

$A_{mg}\ N :: ('v \Rightarrow 'v\ list) \Rightarrow mg_2 \Rightarrow assn$
$A_{mg} \equiv A_{mg2}\ O\ R_{mg1}\ N \qquad A_{it} \equiv A_{it2}\ O\ R_{it1}$
$succ\_at_2,\ successors\_at :: A_{mg}\ N \rightarrow A_{it} \rightarrow A_{snat}$

We have omitted similar steps for the other iterator operations. For those, we refer to the supplementary material.

5.3 Main Algorithm

We again use Sepref to synthesise an implementation of *compute_SCC_impl* which we call *Modest_compute_SCC_impl*. We use the aforementioned refinements to achieve this. We combine all refinements to relate it with the specification *compute_SCC_spec* (cf. Sect. 3.2). The resulting theorem states that our implementation is correct. As this is part of the trusted code base, we invested some effort into making the theorem readable and eliminate unnecessary dependencies on internal IRF concepts. At the end, we obtain a Hoare triple using separation logic and refinement assertions for the input and output data types:

theorem *Modest_graph_SCC_impl_correct_htriple: llvm_htriple*
(∗*1*∗) (A_{snat} *N ni* ∗ A_{mg} *N E_succ Ei* ∗ *N* < 2^{62})
(∗*2*∗) (*Modest_compute_SCC_impl ni Ei*)
(∗*3*∗) (λ*ri. EXS r.* A_{snat} *N ni* ∗ A_{mg} *N E_succ Ei* ∗ *N* < 2^{62} ∗ A_{out} *r ri* ∗
(∗*4*∗) *set r* = *scc_set* ∧ *ordered r*)

The precondition (1) requires a signed 64-bit integer *ni* with value $N :: nat$, and an MDP graph data structure *Ei* representing an abstract successor function *E_succ* with N nodes. We also assume that N is within the bounds incurred by our encoding of node states (cf. Sect. 5.1). Then (2) running our LLVM program *Modest_compute_SCC_impl ni Ei* yields that (3) *ni* and the graph remain unaltered, and (4) the result *ri* encodes a list r of SCCs in topological order.

Here, $ri = (ii, Ii)$ contains the number of SCCs and an array that contains the SCC number for each node. The assertion A_{out} first maps *ri* to a natural number and an actual map (A_{am}), and then uses *SCC_rel* (cf. Sect. 4.2) to map that to a list of SCCs:

$A_{out} \equiv (A_{snat} \times A_{am})\ O\ SCC\_rel$

6 Implementation in the Modest Toolset

We have now refined our specification into a model of LLVM in Isabelle/HOL. As the last step in our approach, we extract executable LLVM code. We generate a C header with type definitions to encapsulate our data so that it can be used by LLVM as well as from C. This allows us to easily align the header with the format that mcsta supports. The **export_llvm** command generates the LLVM code of our SCC finding algorithm as well as the corresponding C header file:

```
export_llvm Modest_compute_SCC_impl is
void compute_SCC(my_size_t, modest_graph_t *, scc_result_t *) defines ‹
  typedef uint64_t my_size_t; typedef my_size_t node_t;
  typedef uint64_t shared_nat_t; typedef uint64_t *bitset_t;
  typedef struct { shared_nat_t *states; struct
      { shared_nat_t *transitions; node_t *branches; }; } modest_graph_t;
  typedef struct { my_size_t num_sccs; node_t *scc_map;
  } scc_result_t;
› file modest_gabow.ll
```

The nested anonymous **struct** in *modest_graph_t* reflects Isabelle/HOL's modelling of tuples as right-nested pairs. We compile the LLVM code into a shared library and invoke the functions in this library from mcsta via C#'s "P/Invoke" mechanism to use libraries following the C ABI. In mcsta, we added a command-line option to choose the SCC algorithm to use for its topological LP implementation: the previous unverified C# implementation of Tarjan's algorithm; a new, manually implemented and optimized version of Gabow's algorithm in C that we added for a fairer performance comparison; and the new verified Isabelle LLVM

implementation. This allows us to easily run tests and performance benchmarks on the three algorithms.

The topological LP implementation in mcsta requires not only that the SCCs are topologically sorted—which is a postcondition of our LLVM program—but also that the states are sorted by SCC. The latter is done on-the-fly by mcsta's Tarjan implementation, and currently by unverified "glue code" integrating the new algorithms. We aim to either remove this requirement from mcsta, or adapt the verified implementation to include the on-the-fly sorting of states as well.

7 Benchmarks

We benchmark our new SCC implementation to show that unverified and verified code have similar performance. We do so by comparing the runtime of all three algorithms now available in mcsta on a set of benchmark instances (combinations of a parametrised system model, parameter values, and a property to check) from the Quantitative Verification Benchmark Set (QVBS) [20].

7.1 Benchmark Selection

We consider three types of models from the QVBS for our benchmark set: DTMC, MDP, and probabilistic timed automata (PTA) [28]. mcsta syntactically converts the latter to MDP via the digital clocks approach [27]. As SCC algorithms have linear complexity, we need large state spaces to stress-test the implementations. This means that memory is our main bottleneck. We thus selected all DTMC, MDP, and PTA benchmark instances from the QVBS that have between 1 and 100 million states. We found that models with fewer states finish too quickly for reliable runtime measurements, while larger models lead to out-of-memory situations on the machine we use.

A benchmark instance includes a property (e.g. a query for a maximum reachability probability) to check. Since our focus is not on the actual numeric algorithms computing the value of the property, but the SCC-finding preprocessing step, we limit ourselves to one property per applicable model-parameters combination, and instruct mcsta via its `--exhaustive` option to explore the full state space. This leaves us with 39 different instances to benchmark.

7.2 Benchmarking Setup

All our benchmarks were performed on an Intel Core i7-12700H system with 32 GB of RAM running 64-bit Ubuntu Linux 22.04. We use the mobench utility of the Modest Toolset to run the benchmarks in an automated fashion based on a JSON file specifying the benchmark instances to use and tools to execute. For the latter, we specify three command line invocations for mcsta: one for our new verified implementation of Gabow's algorithm ("Isabelle Gabow"), one for the manual C implementation of the same algorithm ("C Gabow"), and one for the pre-existing C# implementation of Tarjan's algorithm ("Tarjan"). Running

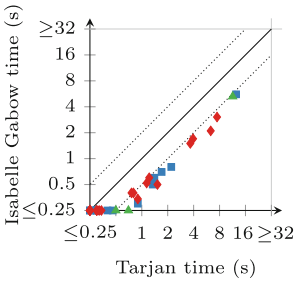


Fig. 4. SCCs only

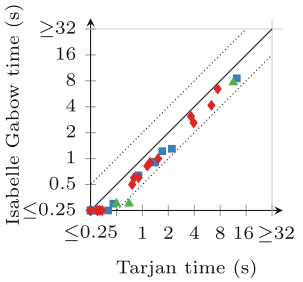
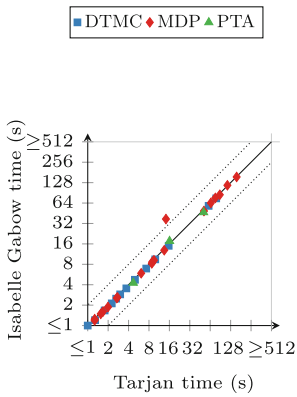
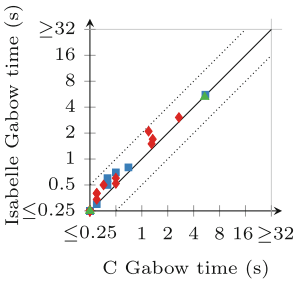


Fig. 5. SCCs + glue code





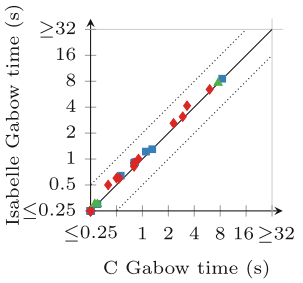


Fig. 6. Topological LP time

mobench yields a CSV file with all measurements, and log files for each individual benchmark run. We then use the MODEST TOOLSET's moplot utility to generate the scatter plots shown in the remainder of this paper. Each benchmark instance results in one point (x, y) indicating that the x-axis task took x seconds to complete while the y-axis task took y seconds for this instance. We note that the "consensus" instance with parameter values $\mathtt{K} = 2$, $\mathtt{N} = 8$ ran out of memory, and the "zeroconf" model caused the LP solver to time out on both its instances. We thus omit these 3 failed instances in our plots.

7.3 Benchmarking Results

Figure 4 compares the runtime of all three implementations of the core SCC-finding algorithms, excluding any time used by the glue code described in Sect. 6. Replacing the C# Tarjan implementation by the verified one of Gabow's algorithm appears to boost the performance, with benchmarks that are more than twice as fast. We suspect the two most important reasons for the difference in performance to be that the Tarjan implementation additionally sorts the MDP's states on-the-fly, and that we compare two different algorithms. However, we have no reliable data to determine the influence on the performance for the latter, if it exists. On the other hand, the manual C implementation of Gabow's algorithm appears to perform similarly to the verified implementation, with the manual implementation having a slight edge in general. This is expected as we have more control over micro-optimizations in the manual implementation.

Figure 5 shows the same comparison but including the time spent in the "glue code" that sorts the MDP's states by SCC for the topological LP solver. Recall that, for Tarjan's algorithm in mcsta, this is done on-the-fly, so we cannot measure it separately. The "fair" difference in performance for these two implementations thus lies between what is shown in Figs. 4 and 5. As expected, the runtime shifts in favour of the Tarjan implementation here, yet the performance boost remains considerable, with speedups of up to two times. This is because the glue code generally takes much less time than the actual SCC algorithm. When comparing the manual C implementation to the verified Isabelle LLVM implementation in this setting, the manual implementation still wins. Both implementations use the same glue code, so effectively we see a fixed offset added to both runtimes.

Finally, Fig. 6 compares the entire model checking procedure, including state space exploration and LP solving (using mcsta's default LP solver, which is currently GLOP, part of the Google OR Tools). We see that, in the grand scheme of things, we maintain the performance of mcsta by replacing the existing unverified Tarjan implementation by a verified implementation of Gabow's algorithm. We improved the performance of the SCC calculation, but since this is only a small fragment of the model checking procedure it does not show in the figure. It does however mean that we have replaced an important part of our model checker without affecting the performance.

We see one outlier in Fig. 6, which is the instance of the "ij" model with parameters `num_tokens_var` = 20. This is caused by a combination of two effects: First, mcsta converts the probabilities in the model to floating-point numbers at some point, which incurs a rounding error and may lead to the accumulation of imprecisions on further processing. This instance works with very small numbers, causing the imprecisions to accumulate along the topology, eventually resulting in a linear program that the LP solver considers infeasible. This causes mcsta to abort with a corresponding error message to the user[2]. Second, topological orderings are not unique, and different implementations of different SCC-finding algorithms can produce different orderings. For this benchmark instance, our implementations of Tarjan's and Gabow's algorithms in fact produce different orderings; and the one obtained by Tarjan's finds an infeasible SCC much later than the one of Gabow's. As a result, the topological LP procedure—solving the SCCs in reverse topological order—aborts much earlier on the Tarjan ordering.

Another notable benchmark instance was of the "zeroconf" model, where the LP solver did not terminate for unknown reasons no matter which SCC-finding algorithm we used (and which was therefore excluded from Figs. 4, 5 and 6). This highlights the need to verify code—especially for the core components such as SCC finding or LP solving of safety-critical software like a model checker.

[2] Despite the error, we did not exclude this instance from the figures because the error is after the SCC computation, so mobench did not flag it as a problem—and ultimately, this provides an interesting insight.

8 Conclusion

We have replaced the SCC-finding algorithm of the state-of-the-art probabilistic model checker mcsta, part of the Modest Toolset, by a verified version, without negatively affecting mcsta's overall performance. We see this work as a first step in gradually replacing the unverified components of a probabilistic model checker by verified ones. While this approach does not immediately produce a fully verified model checker, we can benchmark and optimize each verified component to avoid performance regressions, while decreasing the trusted code base of the model checker with each replacement step. To avoid expensive copying of data representations at the interface between verified and unverified components, the verified algorithms have to work on the same data structures as the original model checker. To this end, we used a stepwise refinement approach to obtain verified LLVM code which can readily be linked with mcsta.

Our verification is based on an existing verification of a rather inefficient purely functional SCC algorithm, which we significantly extended: we generalized the abstract algorithm to support duplicate successor nodes, and clarified the proof by introducing a ghost variable. Moreover, we added additional data structure invariants that are required for in-bounds checks when refining to 64-bit integers. Finally, we replaced the whole implementation by an efficient imperative one. In particular, we accurately modelled mcsta's graph data structure.

We embedded the resulting verified LLVM code into mcsta, and extensively benchmarked it. Our verified algorithm in isolation is faster than the original unverified one used by mcsta, so its use has no negative effect on mcsta's overall performance. To explore the optimisation potential for future work, we also benchmarked a hand-optimized C implementation.

Future Work. The biggest bottleneck in the current implementation is the glue code. We aim to remove this by means of a different encoding, by removing from mcsta the need for states to be sorted by SCC, or by extending the verified implementation by an on-the-fly sorting. Beyond that, our experiments suggest that further optimizing our verified SCC implementation will only have a minute effect on the overall performance. Thus, it is also worth looking at other components of the model checker: Maximal end component finding algorithms [1] are required for sound (i.e. guaranteed ϵ-correct) MDP solution algorithms like interval iteration [15]. As they require an SCC algorithm as a subroutine, they are an obvious next candidate for verification. Interval iteration itself is a further promising verification target, in particular its floating-point-correct variants [16]. To this end, we are already working on extending the IRF to reason about floating-point numbers.

Data Availibility Statement. Our supplementary material, proofs, and the tools used to obtain the results presented in this paper are archived and available at DOI 10.4121/aff9f553-0e9e-4ec2-90e0-20c5b6152862 [21].

References

1. de Alfaro, L.: Formal verification of probabilistic systems. Ph.D. thesis, Stanford University, USA (1997). https://searchworks.stanford.edu/view/3910936
2. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 963–999. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_28
3. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 34–50. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24849-1_3
4. Balyo, T., Heule, M.J.H., Iser, M., Järvisalo, M., Suda, M. (eds.): Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2022-1. Department of Computer Science, University of Helsinki (2022). http://hdl.handle.net/10138/318450
5. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3
6. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. J. Autom. Reason. **60**(1), 3–21 (2017). https://doi.org/10.1007/s10817-017-9418-4
7. Chen, R., Lévy, J.-J.: A semi-automatic proof of strong connectivity. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 49–65. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_4
8. Dai, P., Goldsmith, J.: Topological value iteration algorithm for Markov decision processes. In: Veloso, M.M. (ed.) 20th International Joint Conference on Artificial Intelligence (IJCAI), pp. 1860–1865 (2007). http://ijcai.org/Proceedings/07/Papers/300.pdf
9. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reduction and refinement strategies for probabilistic analysis. In: Hermanns, H., Segala, R. (eds.) PAPM-PROBMIV 2002. LNCS, vol. 2399, pp. 57–76. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45605-8_5
10. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
11. Fleury, M.: Optimizing a verified SAT solver. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2019. LNCS, vol. 11460, pp. 148–165. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20652-9_10
12. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_3
13. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3), 107–114 (2000). https://doi.org/10.1016/S0020-0190(00)00051-X
14. Haddad, S., Monmege, B.: Reachability in MDPs: refining convergence of value iteration. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP 2014. LNCS, vol. 8762, pp. 125–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11439-2_10

15. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018). https://doi.org/10.1016/j.tcs.2016.12.003
16. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13244, pp. 41–59. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99527-0_3
17. Hartmanns, A., Hermanns, H.: The Modest Toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
18. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13993, pp. 469–488. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30823-9_24
19. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 488–511. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_26
20. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 344–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_20
21. Hartmanns, A., Kohlen, B., Lammich, P.: Artifact for the paper "Fast verified SCCs for probabilistic model checking". 4TU.Centre for Research Data (2023). https://doi.org/10.4121/aff9f553-0e9e-4ec2-90e0-20c5b6152862
22. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). https://doi.org/10.1007/s10009-021-00633-z
23. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 269–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_18
24. Hölzl, J.: Markov chains and Markov decision processes in Isabelle/HOL. J. Autom. Reason. **59**(3), 345–387 (2016). https://doi.org/10.1007/s10817-016-9401-5
25. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
26. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
27. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006). https://doi.org/10.1007/s10703-006-0005-2
28. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theor. Comput. Sci. **282**(1), 101–150 (2002). https://doi.org/10.1016/S0304-3975(01)00046-9
29. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_9
30. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21

31. Lammich, P.: Refinement to Imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_17
32. Lammich, P.: Generating verified LLVM from Isabelle/HOL. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving (ITP). LIPIcs, vol. 141, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
33. Lammich, P.: Efficient verified (UN)SAT certificate checking. J. Autom. Reason. **64**(3), 513–532 (2019). https://doi.org/10.1007/s10817-019-09525-z
34. Lammich, P.: Refinement of parallel algorithms down to LLVM. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving (ITP). LIPIcs, vol. 237, pp. 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ITP.2022.24
35. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12
36. Mansour, M.A., Schäffeler, M.: Formally verified solution methods for Markov decision processes. In: 37th AAAI Conference on Artificial Intelligence, pp. 15073–15081 (2022). https://doi.org/10.1609/aaai.v37i12.26759
37. Neumann, R.: Using Promela in a fully verified executable LTL model checker. In: Giannakopoulou, D., Kroening, D. (eds.) VSTTE 2014. LNCS, vol. 8471, pp. 105–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12154-3_7
38. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL – A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
39. Pottier, F.: Depth-first search and strong connectivity in Coq. In: Vingt-sixièmes journées francophones des langages applicatifs (JFLA) (2015)
40. Puterman, M.L.: Markov decision processes. Handb. Oper. Res. Manag. Sci. **2**, 331–434 (1990)
41. Quatmann, T., Katoen, J.-P.: Sound value iteration. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 643–661. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_37
42. Vajjha, K., Shinnar, A., Trager, B.M., Pestun, V., Fulton, N.: CertRL: formalizing convergence proofs for value and policy iteration in Coq. In: Hritcu, C., Popescu, A. (eds.) 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pp. 18–31. ACM (2021). https://doi.org/10.1145/3437992.3439927
43. Wimmer, S., Herbreteau, F., van de Pol, J.: Certifying emptiness of timed Büchi automata. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 58–75. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_4
44. Wimmer, S., Lammich, P.: Verified model checking of timed automata. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 61–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_4
45. Wimmer, S., Mutius, J.: Verified certification of reachability checking for timed automata. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12078, pp. 425–443. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_24