


Fast Symbolic Computation of Bottom SCCs

Anna Blume Jakobsen, Rasmus Skibdahl Melanchton Jørgensen, Jaco van de Pol, and Andreas Pavlogiannis(✉)

Aarhus University, Aarhus, Denmark
{jaco,pavlogiannis}@cs.au.dk

Abstract. The computation of bottom strongly connected components (BSCCs) is a fundamental task in model checking, as well as in characterizing the attractors of dynamical systems. As such, symbolic algorithms for BSCCs have received special attention, and are based on the idea that the computation of an SCC can be stopped early, as soon as it is deemed to be non-bottom.

In this paper we introduce Pendant, a new symbolic algorithm for computing BSCCs which runs in linear symbolic time. In contrast to the standard approach of *escaping* non-bottom SCCs, Pendant aims to *start* the computation from nodes that are likely to belong to BSCCs, and thus is more effective in sidestepping SCCs that are non-bottom. Moreover, we employ a simple yet powerful *deadlock-detection* technique, that quickly identifies singleton BSCCs before the main algorithm is run. Our experimental evaluation on three diverse datasets of 553 models demonstrates the efficacy of our two methods: Pendant is decisively faster than the standard existing algorithm for BSCC computation, while deadlock detection improves the performance of each algorithm significantly.

Keywords: BDDs · strongly connected components · symbolic algorithms

1 Introduction

The decomposition of a graph to its *strongly connected components* (*SCCs*) is one of the most standard tasks in automated system verification. For example, model checking against LTL and ω-regular properties reduces to computing cycles [30], while fairness conditions are typically checked given an SCC decomposition of the graph [21,34]. Of special interest are *bottom/terminal SCCs* (or *BSCCs*), i.e., SCCs that, once entered, cannot be escaped. BSCCs are used to speed up LTL model checking [28], and they capture the long-run properties of Markov Chains [4,11] and Markov Decision Processes [23,13], while they also correspond to the attractors of dynamical systems, as in signal transduction networks [29,33].

Large-scale model-checking settings comprise huge systems that suffer from the state-space explosion problem. These systems are usually represented compactly by a model, e.g., by means of a programming language, a logic or a reaction network, and have size that is exponentially large in its description. Nevertheless, the system typically exhibits numerous symmetries that can be preserved when

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 110–128, 2024.
https://doi.org/10.1007/978-3-031-57256-2_6

the state space is represented *symbolically* rather than *explicitly*. One predominant symbolic representation is via (reduced/ordered) Binary Decision Diagrams (BDDs) [10], which are found at the core of many classic and modern model checkers [14,24,20,26,5]. To benefit from the symbolic representation, analysis algorithms typically only have *coarse-grained* access to the graph, querying for the *successors* ($\text{Post}(X)$) and *predecessors* ($\text{Pre}(X)$) of a set of nodes X represented by a single BDD. Each such operation counts as a *symbolic step*. As symbolic steps are significantly slower than primitive operations, they serve as the complexity measure of symbolic algorithms [9,18,12,25].

Due to the prevalence of SCC decomposition, the problem has been studied extensively in the symbolic setting, starting with the Xie-Beerel algorithm [32] of symbolic complexity $O(n^2)$; LockStep [8] improves this bound to $O(n \log n)$, while Skeleton [17] achieves $O(n)$ time at the expense of $\Theta(n)$ symbolic space (i.e., number of BDDs). The most recent step in this progression is Chain [25] which achieves both $O(n)$ symbolic time and $O(\log n)$ symbolic space. In practice, heuristics aim to further improve the running time [31,16,34].

Naturally, the computation of BSCCs can be achieved by using one of the aforementioned algorithms to obtain an SCC decomposition, and check whether each SCC is indeed a BSCC. In practice, however, computing an SCC can be expensive, as it typically requires traversing it multiple times. For this reason, algorithms dedicated to BSCCs have received special attention. Although these do not offer theoretical improvements, they attempt to minimize the number of non-bottom SCCs computed and thus perform better in practice.

The predominant, general-purpose BSCC-decomposition algorithm is BwdFwd, which is a modification of Xie-Beerel [32], and has $O(n)$ complexity. Effectively, this algorithm aborts the computation of an SCC S as soon as it determines that S cannot be a BSCC, and removes it from the graph, as well as any node that can reach S. A recently-introduced preprocessing technique, called interleaved transition-guided reduction (ITGR) [6], aims to further detect and discard non-bottom SCCs before the main algorithm is run. ITGR is general-purpose, and was shown to be effective in handling asynchronous Boolean Network models [3,1,2]. However, as these algorithms are typically executed on huge inputs, issues of scalability often remain. We address this challenge here.

1.1 Our contributions

The Pendant algorithm. We develop a new, linear-time algorithm for symbolic BSCC computation, called Pendant, drawing inspiration from the recent Chain algorithm [25]. In contrast to the existing BSCC paradigm based on *stopping* the computation of SCCs that are deemed non-bottom, Pendant aims to *start* such computations from SCCs that are likely to be bottom. To achieve this, while Pendant computes an SCC, it also implicitly (at no extra cost) traverses the quotient graph Q downwards, making future SCC computations start from nodes that are close to the bottom of Q, and thus discover a BSCC quickly.

Deadlock detection. We employ a simple yet powerful preprocessing technique, called *deadlock-detection*. This is based on the insights that (i) each deadlock (singleton SCC) is a BSCC, and (ii) *all* deadlocks can be computed effectively in a *single* symbolic step.

Experimental evaluation. We implement Pendant and the deadlock-detection preprocessing, and evaluate their performance on computing the BSCCs of a large pool of models from three diverse datasets, namely, (i) Petri Nets from the Model Checking Contest [22], (ii) DiVinE models from the Benchmark of Explicit Models [27], and (iii) Asynchronous Boolean Network models [3,1,2]. Our experiments conclude that (i) Pendant is decisively more efficient than BwdFwd, (ii) deadlock-detection improves the performance of both algorithms, and (iii) after deadlock-detection, ITGR is scarcely effective.

2 Preliminaries

In this section we present standard definitions and the BwdFwd algorithm.

2.1 Graphs, Bottom SCCs and Symbolic Representations

Graphs. We consider directed graphs $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V$ is a set of edges. We often write $u \to v$ to denote an edge $(u, v) \in E$. For a node v, the image of v is $\mathrm{Post}(v) = \{u \mid v \to u\}$, while the pre-image of v is $\mathrm{Pre}(v) = \{u \mid u \to v\}$. These notions are extended to sets of nodes X in the natural way, i.e., $\mathrm{Post}(X) = \bigcup_{v \in X} \mathrm{Post}(v)$ and $\mathrm{Pre}(X) = \bigcup_{v \in X} \mathrm{Pre}(v)$.

A path is a sequence $P = v_1 \to v_2 \to \cdots \to v_k$, in which case we also write $v_1 \rightsquigarrow v_k$, and say that v_k is reachable from v_1. The length of P is $|P| = k - 1$. For a set of nodes X we write $\mathrm{Fwd}(X) = \{u \mid \exists v \in X, v \rightsquigarrow u\}$ for the forward set of X and $\mathrm{Bwd}(X) = \{u \mid \exists v \in X, u \rightsquigarrow v\}$ for the backward set of X. We call a set $X \subseteq V$ *forward-closed* if $\mathrm{Fwd}(X) \subseteq X$. The restriction of G on a set $X \subseteq V$ is the graph $G[X] = (X, (X \times X) \cap E)$. A node $v \in V$ is called a *deadlock* if it has no outgoing edges, i.e., $\mathrm{Post}(v) = \emptyset$.

Bottom Strongly Connected Components (BSCCs). A *strongly connected component* (*SCC*) of G is a maximal set of nodes S such that for all $u, v \in S$ we have $u \rightsquigarrow v$. Each node v belongs to one SCC, written $\mathrm{SCC}(v)$. A set $X \subseteq V$ is called *SCC-closed* if for each $v \in X$, we have $\mathrm{SCC}(v) \subseteq X$. The *diameter* of an SCC S is the maximum distance between two nodes in S, i.e.,

$$\delta(S) = \max_{u,v \in S} \min_{P\colon u \rightsquigarrow v} |P|$$

The quotient graph of G represents each SCC of G by a single node, and has a directed edge $S \to S'$ iff $\mathrm{Post}(S) \cap S' \neq \emptyset$, i.e., there exists nodes $u \in S$ and $v \in S'$ with $u \to v$. The quotient graph is a directed acyclic graph. The leaf nodes of a quotient graph represent the SCCs that have no outgoing edges to

any other SCCs, called *bottom SCCS* (or *BSCCs*). We denote by SCCs(G) and BSCCs(G) the set of SCCs and BSCCs of G, respectively.

The problem targeted in this paper is the computation of BSSCs. The following two simple properties of BSCCs are used throughout the paper.

Proposition 1. *An SCC S is a BSCC if and only if* $\mathrm{Fwd}(S) = S$.

Proposition 2. *If S is a BSCC then there is no BSCC in* $\mathrm{Bwd}(S) \setminus S$.

Symbolic operations and complexity. In large-scale model-checking settings, graphs are typically represented symbolically. One popular symbolic representation is Binary Decision Diagrams (BDDs) [19]. In particular, the node set V and edge relation E are represented compactly as BDDs, while algorithms use BDDs as data structures for representing subsets of V and E. The basic BDD operations give only coarse-grained access to the graph: given a BDD representing a set of nodes X, an algorithm can access $\mathrm{Pre}(X)$ and $\mathrm{Post}(X)$, each of which counts as one *symbolic step*. The complexity of symbolic algorithms is measured in the number of symbolic steps they execute [12,25], since these are much slower than elementary operations (e.g., incrementing a counter). Basic set operations on BDDs (union, intersection, etc.) also do not count towards the time complexity*. Finally, given a set X represented as a BDD, we use a $\textsc{Pick}(X)$ operation which returns an arbitrary node $v \in X$. This operation is natural and efficient for BDDs, and has been common in symbolic SCC algorithms [17,8,25].

2.2 The BwdFwd Algorithm for BSCCs

The symbolic computation of BSCCs(G) can be performed by computing each $S \in \mathrm{SCCs}(G)$ using some existing symbolic algorithm [32,17,8,25], and then reporting that S is a BSCC iff $\mathrm{Post}(S) \subseteq S$ (following Proposition 1). Although this approach runs in $O(n)$ symbolic steps when using Chain [25] or Skeleton [17], it can be unnecessarily slow in practice, as it typically spends considerable time computing SCCs that are not BSCCs. For this reason, the computation of BSCCs is targeted by algorithms dedicated to this task. The standard symbolic BSCC algorithm is BwdFwd, which we briefly present here.

The Backward-Forward BSCC algorithm. BwdFwd is an adaptation of the standard Xie-Beerel algorithm [32]. Algorithm 2 follows its recent presentation in [6], adapted to our setting. The algorithm uses the standard mechanism for computing SCCs symbolically: given a pivot node v, we have $\mathrm{SCC}(v) = \mathrm{Fwd}(v) \cap \mathrm{Bwd}(v)$. Given such a node v, BwdFwd first calls Algorithm 1 (Line 3) to retrieve the backward set $\mathrm{Bwd}(v)$ (called the *basin* of v) using a standard fixpoint computation. Then, it uses a similar fixpoint computation to retrieve $\mathrm{Fwd}(v)$ (Line 5) in F. This computation is terminated early

*For many algorithms, including ours, counting set operations does not affect the asymptotic complexity.

Algorithm 1: Bwd

```
   Input: A graph G = (V, E) and a node v ∈ V
1  B = {v}
2  while Pre(B) ⊈ B do                          // Fixpoint not reached
3     B = B ∪ Pre(B)                            // Update with new predecessors
4  return B
```

Algorithm 2: BwdFwd

```
   Input: A graph G = (V, E)
1  if V = ∅ then return
2  v = Pick(V)                                  // Pick a pivot
3  B = Bwd(G, v)                                // Compute safe-to-remove nodes
4  F = ∅; Layer = {v}
5  while Layer ≠ ∅ and F ⊆ B do                 // Compute and detect BSCC
6     F = F ∪ Layer
7     Layer = Post(Layer) \ F
8  if F ⊆ B then                                // Output if BSCC
9     output Fwd
10 BwdFwd(G[V \ B])                             // Recursive call w/o safe nodes
```

if the algorithm discovers that $\mathrm{Fwd}(v) \not\subseteq \mathrm{Bwd}(v)$, as then $\mathrm{Fwd}(v) \not\subseteq \mathrm{SCC}(v)$, and due to Proposition 1, we have that $\mathrm{SCC}(v)$ is not a BSCC. On the other hand, if the computation is carried to a fixpoint, we have that $\mathrm{Fwd}(v) \subseteq \mathrm{Bwd}(v)$ and thus $\mathrm{Fwd}(v) = \mathrm{SCC}(v)$; then, Proposition 1 guarantees that $\mathrm{SCC}(v)$ is a BSCC. Since the check in Line 9 succeeds, BwdFwd correctly outputs $\mathrm{SCC}(v)$ as a BSCC. Finally, Proposition 2 guarantees that the basin $\mathrm{Bwd}(v)$ contains no BSCC, except possibly $\mathrm{SCC}(v)$ which was just outputted. The algorithm hence safely removes $\mathrm{Bwd}(v)$ from G, and proceeds recursively (Line 10).

It is not hard to see that BwdFwd runs in $O(n)$ symbolic steps, but offers two practical improvements over general SCC-decomposition algorithms. In each recursive call, the algorithm avoids computing SCCs in $\mathrm{Bwd}(v) \setminus \mathrm{SCC}(v)$ as they are guaranteed to be non-bottom; nodes in this set are only accessed during the basin computation in Algorithm 1, which is cheaper. Moreover, it stops computing $\mathrm{SCC}(v)$ as soon as it discovers that $\mathrm{Fwd}(v) \not\subseteq \mathrm{Bwd}(v)$ (as $\mathrm{SCC}(v)$ is not a BSCC). However, the algorithm can spend significant time in computing $\mathrm{Fwd}(v)$ before it discovers that $\mathrm{Fwd}(v) \not\subseteq \mathrm{Bwd}(v)$, which results in wasteful symbolic operations. The following example illustrates this issue on a small graph.

Example. Fig. 1 shows a graph $G = (V, E)$ (a) and two recursion trees. The left-most tree (b) illustrates the execution of BwdFwd on G. Each node in the tree has its variables subscripted by the pivot node v chosen in the corresponding recursive call, with the variables showing their values in that recursive call. E.g.,

F_v is the value of F after the loop of Line 5 has completed, given that v was chosen as pivot in that recursive call. The number of a node is underlined in F_v if it is a node is outside the backward set B_v and cuts the computation of F_v short (Line 5). Observe that the algorithm makes four recursive calls, where the second ($v = 2$) and third ($v = 3$) call spend considerable time in the forward computation (of the sets F_2 and F_3, respectively), and essentially compute SCC(2) and SCC(3) before determining that these are not BSCCs.

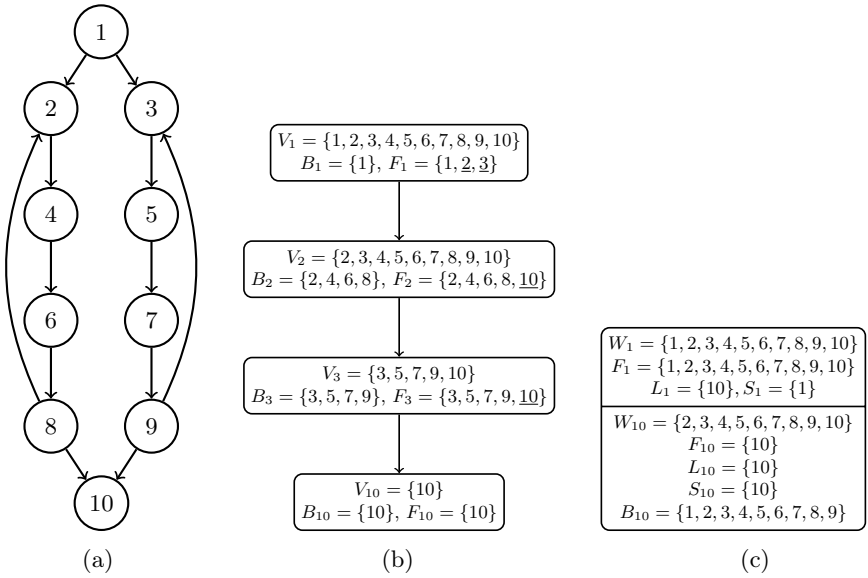

Fig. 1: An example input graph (a) and the recursion trees of the BwdFwd (b) and Pendant (c) algorithms on it.

3 The Pendant Algorithm for BSCCs

In this section we present our new algorithm, Pendant, for computing BSCCs symbolically. Like BwdFwd, Pendant spends linear time in the number of nodes of the input graph. In particular, we have the following theorem.

Theorem 1. *Given a graph $G = (V, E)$ of n nodes,* Pendant *computes BSCCs(G) in $O(\sum_{S \in \mathrm{SCCs}(G)} \delta(S)) = O(n)$ symbolic time.*

However, as we will see in Section 5, in practice Pendant typically requires fewer symbolic steps than BwdFwd. Intuitively, this is achieved by making, over time, smarter choices of pivot nodes v to start the SCC computation, meaning nodes v that are more likely to have $\mathrm{SCC}(v)$ close to the leaves of the quotient graph.

In turn, this reduces the number of non-bottom SCCs computed throughout the execution of the algorithm, which reduces the number of symbolic steps.

3.1 Pendant

Pendant is shown in Algorithm 4, and uses FwdLastLayer, shown in Algorithm 3, as a sub-procedure.

Algorithm 3: FwdLastLayer

```
   Input: A graph G = (V, E) and a node v ∈ V
1  F = ∅; Layer = {v}; L = ∅
2  while Layer ≠ ∅ do                                  // Fixpoint not reached
3     F = F ∪ Layer                                    // Update with new successors
4     L = Layer                  // L stores the last layer of nodes reached
5     Layer = Post(Layer) \ F                          // Compute the new layer
6  return F, L
```

FwdLastLayer. FwdLastLayer computes the forward set $\mathrm{Fwd}(v)$ of a node v using a standard fixpoint computation. The algorithm also keeps track of the last layer L of nodes discovered during the fixpoint computation, and returns both $\mathrm{Fwd}(v)$ (represented in F) and L. Intuitively, $\mathrm{Fwd}(v)$ is used by Pendant for computing $\mathrm{SCC}(v)$ and testing whether it is a BSCC, while L is used to guide the selection of future pivots downwards in the quotient graph.

Pendant. On input $G = (V, E)$, Pendant begins by Pick'ing an arbitrary pivot node v (Line 2), with the aim to compute $\mathrm{SCC}(v)$ and test whether it is a BSCC. For this purpose, it calls FwdLastLayer to retrieve $F = \mathrm{Fwd}(v)$, and L being the last layer of $\mathrm{Fwd}(v)$ (Line 5). It then computes $S = \mathrm{SCC}(v)$, by calling Bwd (Algorithm 1, Line 6) to compute the backward set of v restricted to $\mathrm{Fwd}(v)$. At this point, there are two cases.

- If $F \setminus S \neq \emptyset$, then S is not a BSCC. At this point, the set $W = F \setminus S$ is guaranteed to contain a BSCC, and the algorithm resumes its search for a BSCC in this set, running a new iteration of the main loop. Moreover, the algorithm attempts to pick a new pivot in the last layer of $\mathrm{Fwd}(v)$ (Line 10), as opposed to an arbitrary node in W. Intuitively, this effectively allows Pendant to traverse the quotient graph downwards towards its leaves, and thus quickly pick a pivot v such that $\mathrm{SCC}(v)$ is a BSCC.
- If $F \setminus S = \emptyset$, then $D = \mathrm{SCC}(v)$ is guaranteed to be a BSCC; this is reported (Line 15), and the loop breaks (Line 4). Then the backwards set of B is computed and removed from the graph, as it is guaranteed to not contain any other BSCC, and the algorithm proceeds recursively in the remaining graph (Line 17). Note that the number of recursive calls of Pendant thus equals the number of BSCCs in the input graph.

Algorithm 4: PENDANT

```
   Input: A graph G = (V, E)
1  if V = ∅ then return
2  v = Pick(V)                                          // Pick a pivot
3  W = V; D = ∅                                         // D stores a BSCC, once found
4  while D = ∅ do                                       // Find a BSCC
5     F, L = FwdLastLayer(G[W], v)                      // Get Fwd(v) and its last layer
6     S = Bwd(G[F], v)                                  // Compute SCC(v)
7     if F \ S ≠ ∅ then                                 // Not a BSCC
8        W = F \ S                                      // W contains a BSCC, continue here
9        if L ∩ W ≠ ∅ then                              // If there are candidates in last layer,
10          v = Pick(L ∩ W)                             // pick new pivot from the last layer
11       else
12          v = Pick(W)                                 // otherwise, pick any v from W
13    else
14       D = S
15       output D
16 B = Bwd(D, G)                                        // Compute safe-to-remove nodes
17 Pendant(G[V \ B])                                    // Recursive call w/o safe nodes
```

Observe the qualitative differences between PENDANT and BWDFWD. First, BWDFWD begins with a backward search from the pivot v, while PENDANT begins with a forward search from v. Second, BWDFWD removes the basin $\mathrm{Bwd}(v)$ from G as soon as $\mathrm{SCC}(v)$ is deemed to be non-bottom, while PENDANT delays this step, and only computes (and removes) the basin of BSCCs. Third, BWDFWD picks pivots completely arbitrarily, whereas PENDANT, any time it computes an SCC S that is not bottom, it picks the next pivot from a distant successor of S in the quotient graph, which allows it to discover BSCCs quickly.

Example. Let us revisit our example in Fig. 1. The right-most recursion tree (c) illustrates the computation of PENDANT. Since there is only one BSCC, there is only one recursive call, but the node is subdivided to show each iteration of the loop in Line 4. As before, variables are subscripted with the pivot node v of that iteration. Initially, PENDANT chooses arbitrarily $v = 1$, like BWDFWD, and computes $\mathrm{Fwd}(1)$. Then, it deems $\mathrm{SCC}(1)$ as a non-bottom SCC, and the next pivot is chosen from the last layer of $\mathrm{Fwd}(1)$, i.e., $v = 10$. Effectively, PENDANT has reached a leaf of the quotient graph (the only leaf, in this case), and thus identifies a BSCC quickly. Importantly, it skips the expensive computation of two SCCs with large diameters ($\mathrm{SCC}(2)$ and $\mathrm{SCC}(3)$), in contrast to BWDFWD.

3.2 Correctness

We now turn our attention to the correctness of PENDANT. We start with two simple lemmas regarding forward-closed sets.

Lemma 1. *Assume that $X \subseteq V$ is forward-closed, and $D \subseteq X$ is a BSCC. Then $X \setminus \mathrm{Bwd}(D)$ is forward-closed.*

Proof. For any node $v \in X$, if $\mathrm{Fwd}(v) \cap \mathrm{Bwd}(D) \neq \emptyset$ then clearly $v \in \mathrm{Bwd}(D)$ and hence $v \notin X \setminus \mathrm{Bwd}(D)$. Thus, for every node $v \in X \setminus \mathrm{Bwd}(D)$, we have $\mathrm{Fwd}(v) \cap \mathrm{Bwd}(D) = \emptyset$, and since X is forward-closed, we have $\mathrm{Fwd}(v) \subseteq X$. □

Lemma 2. *For any node v, the set $\mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$ is forward-closed.*

Proof. For any node $u \in \mathrm{Fwd}(v)$, if $\mathrm{Fwd}(u) \cap \mathrm{SCC}(v) \neq \emptyset$, then $u \in \mathrm{SCC}(v)$. Hence for every node $u \in \mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$, we have $\mathrm{Fwd}(u) \cap \mathrm{SCC}(v) = \emptyset$, and thus $\mathrm{Fwd}(u) \subseteq \mathrm{Fwd}(v) \setminus \mathrm{SCC}(v)$. The desired result follows. □

We now prove the soundness of Pendant, i.e., every SCC outputted in Line 15 is a BSCC. For this, we prove the following stronger lemma, which states three invariants maintained by the algorithm.

Lemma 3. *At each iteration of the main loop of* Pendant*, the following invariants hold: (a) V and W are forward-closed, (b) S is an SCC, and (c) D is a BSCC.*

Proof. Before entering the first iteration of the loop, we have that each of W and V is the whole node set of the input graph, hence both are trivially forward-closed. Now, assuming that W is forward-closed, we have that $F = \mathrm{Fwd}(v)$ in Line 5. In turn, this implies that $S = \mathrm{SCC}(v)$ in Line 6. Moreover, due to Proposition 1, if $F \subseteq S$ in Line 7, then S is a BSCC, thus D outputted in Line 15 is indeed a BSCC.

To complete the invariant proof, it remains to argue that V' and W' remain forward-closed after they have been updated. There are two cases.

1. If the algorithm proceeds with another iteration of the main loop, we have $V' = V$ and $W' = F \setminus S$. Since $F = \mathrm{Fwd}(v)$ and $S = \mathrm{SCC}(v)$, Lemma 2 implies that W' is forward-closed.
2. Otherwise, the algorithm proceeds with a new recursive call in Line 17. We have that $W' = V' = V \setminus B$, where $B = \mathrm{Bwd}(D)$, and D is a BSCC. By Lemma 1, we have that $V \setminus B$ is forward-closed, as desired. □

Observe that case (c) of Lemma 3 establishes the soundness of Pendant. Next we establish its completeness, thereby concluding the correctness of Pendant.

Lemma 4. Pendant *outputs every BSCC of the input graph.*

Proof. First, observe every time Pendant calls itself recursively in Line 17, it has outputted a BSCC D, and the recursion proceeds on the subgraph $V \setminus \mathrm{Bwd}(D)$. Due to Proposition 2, the algorithm has outputted all BSCCs in $V \setminus$

$\mathrm{Bwd}(D)$. Hence, in each recursive call on a graph $G = (V, E)$, the node set V contains all the BSCCs not already outputted by the algorithm. It thus suffices to argue that, in each recursive call, the main loop eventually terminates, as in doing so it outputs a BSCC.

In each iteration of the main loop, the set W is updated to $W' = F \setminus S$ (Line 8), where $F = \mathrm{Fwd}(v)$ and $S = \mathrm{SCC}(v)$, where v is the current pivot. Since $F \subseteq W$ and $S \neq \emptyset$, it follows that $W' \subsetneq W$, and thus the loop must eventually terminate. □

3.3 Complexity

Although the linear upper-bound of BwdFwd is trivial, the case of Pendant is more involved. This is because a call to FwdLastLayer may compute forward sets that consist of many layers (and thus cost many symbolic steps), while these sets are not immediately removed from the graph (as opposed to the backward set computed by BwdFwd), and are again accessed in future iterations of the algorithm. Nevertheless, a careful analysis shows that the complexity is indeed linear. We start with a simple lemma.

Lemma 5. *Assume that $X \subseteq V$ is forward-closed and $D \subseteq X$ is a BSCC. Then $\mathrm{Bwd}(D) \cap X$ is SCC-closed.*

Proof. Consider any node $v \in \mathrm{Bwd}(D) \cap X$. Since X is forward-closed, we have $\mathrm{Fwd}(v) \subseteq X$ and thus $\mathrm{SCC}(v) \subseteq X$. Moreover, $\mathrm{Bwd}(v) \subseteq \mathrm{Bwd}(D)$ and thus $\mathrm{SCC}(v) \subseteq X$. Hence $\mathrm{SCC}(v) \subseteq \mathrm{Bwd}(D) \cap X$.

We now prove the complexity of Pendant.

Lemma 6. Pendant *runs in $O(\sum_{S \in \mathrm{SCCs}(G)} \delta(S)) = O(n)$ symbolic steps.*

Proof. In each recursive call, Pendant makes symbolic steps to (i) compute the SCCs of the picked pivots (Lines 5 and 6), and (ii) compute the backwards set of the outputted BSCC (Line 16). We will argue that, in total, case (i) takes $\sum_{S \in \mathrm{SCCs}(G)} 3\delta(S)$ time, while case (ii) takes $\sum_{S \in \mathrm{SCCs}(G)} \delta(S)$ time.

We start with case (i). For a given pivot v, computing $\mathrm{SCC}(v)$ is done in two steps: (a) Line 5 computes the forward set F of v restricted to the node set W, while (b) Line 6 computes $\mathrm{SCC}(v)$ as the backward set of v restricted to F. Clearly, (b) takes $\delta(\mathrm{SCC}(v))$ symbolic steps, thus summing over all pivots v, we have that Line 6 takes at most $\sum_{S \in \mathrm{SCCs}(G)} \delta(S)$ time. To bound the time spent in (a), denote by $\mathrm{Levels}(v)$ the number of iterations executed in FwdLastLayer, i.e., Pendant spends $\mathrm{Levels}(v)$ symbolic steps in Line 5. If $F \setminus \mathrm{SCC}(v) = \emptyset$ or $L \setminus \mathrm{SCC}(v) = \emptyset$, we have $\mathrm{Levels}(v) = \delta(\mathrm{SCC}(v))$. Otherwise, the next pivot v' is Pick'ed from L (Line 10). Consider a shortest-path $P\colon v \rightsquigarrow v'$, and let $\{S_1, \ldots S_k\}$ be the SCCs of nodes along P (except v), and note that $\mathrm{Levels}(v) \leq$

$\sum_{i=1}^{k} \delta(S_i)$. Moreover, we have $S_i \subseteq \text{Bwd}(v')$ for each $i \in \{1, \ldots, k\}$, and thus each S_i is not touched again by FwdLastLayer, except if $S_i = \text{SCC}(v')$, but this case is accounted for already. Summing over all such S_i across all pivots v, we have that $\sum_v \text{Levels}(v) \leq \sum_{S \in \text{SCCs}(G)} \delta(S)$. Hence the total symbolic time spent for case (i) is bounded by $\sum_{S \in \text{SCCs}(G)} 3\delta(S)$.

We now turn our attention to case (ii). Due to Lemma 3, W is forward closed and D is a BSCC. By Lemma 5, the set B computed in Line 16 is SCC-closed. The number of symbolic steps is hence bounded by $\sum_{S \in \text{SCCs}(B)} \delta(S)$. Finally, B is removed from the graph in the recursive call, hence it will not be processed again. Thus the total time for case (ii) is $\sum_{S \in \text{SCCs}(G)} \delta(S)$. □

4 Deadlock Detection

We now outline a simple but effective preprocessing technique for BSCCs.

Recall that a deadlock is a node v without outgoing edges, i.e., $\text{Post}(v) = \emptyset$. Observe that all deadlocks are BSCCs: formally we have $\text{Fwd}(\{v\}) = \{v\} = \text{SCC}(v)$, and thus the statement follows from Proposition 1 (the opposite is, of course, not true in general). Thus, deadlock-detection can be seen as a natural preprocessing step to any BSCC algorithm.

The key observation in this approach is that *the set of all* deadlocks can be computed efficiently, in only one symbolic step; this is achieved by Algorithm 5. In particular, the deadlock set is computed as $D = V \setminus H$ where H is the set of nodes u that have a successor. In turn, H can be computed by a single Pre operation on the entire node set. Finally, due to Proposition 2, the set $\text{Bwd}(D)$ is guaranteed to contain no BSCCs other than those in D, and thus it can be removed. The resulting graph is then passed to the main BSCC algorithm.

Algorithm 5: Deadlock detection (preprocessing)

Input: A graph $G = (V, E)$
1 $H = \text{Pre}(V, G)$ `// Compute all nodes that have a successor`
2 $D = V \setminus H$ `// Compute all deadlocks`
3 $B = \text{Bwd}(D, G)$ `// Compute safe-to-remove nodes`
4 **output** each node in D `// Output BSCCs`
5 **return** $G[V \setminus B]$ `// Return remaining graph for further computation`

5 Experiments

Here we report on an implementation of Pendant, including the deadlock-detection technique, and an experimental evaluation of its performance on a large dataset of standard model-checking benchmarks across various domains.

Baselines. To assess the performance of Pendant and deadlock detection, we compare it with BwdFwd (Algorithm 2), as well as the recently introduced *interleaved transition guided reduction* (*ITGR*) [6], which we have implemented in our setting. ITGR is applicable when the transition relation is partitioned into a number of smaller relations $E = (R_1, \ldots, R_k)$ (as is the case in our setup), and works as a preprocessing step, much like our deadlock detection. At a high level, ITGR employs some local reasoning for each relation R_i to identify sets of nodes that do not contain BSCCs. Such sets can be removed, reducing the size of the graph that is further processed by a BSCC-computation algorithm.

Research Questions. Our setup is centered around the following questions.

RQ1 How does the performance of Pendant compare to that of BwdFwd?
RQ2 How does deadlock detection impact the performance of Pendant and BwdFwd?
RQ3 How does ITGR impact the performance of Pendant and BwdFwd?
RQ4 How does the performance of Pendant compare to the performance of BwdFwd when both use deadlock detection?
RQ5 How does ITGR impact the performance of Pendant after deadlock detection?

Datasets. We use benchmarks from the following categories.

- Petri Net models from MCC, the Model Checking Contest [22].
- DiVinE models from BEEM, the Benchmark of Explicit Models [27].
- Asynchronous Boolean Network models [3,1,2].

We do not apply any selection criteria, except discarding models that are too slow to handle by all algorithms in our timed experiments. This results in 553 models in total.[†] In each model, the edge relation is naturally partitioned into subrelations $R_1, \ldots, R_k$, following the structure of the high-level specifications (transitions in Petri Nets and DiVinE state machines, and reactions in the Boolean Networks). We use the language-independent model checker LTSmin [20] to generate symbolic graphs for the DVE and PNML models. Since LTSmin does not handle Boolean Networks, these graphs are generated by a custom parser. The time taken for the graph generation is not measured in the running time of each algorithm. We use the BDD package Sylvan as our symbolic representation [15].

Experimental setup. Our experiments are run on a Linux machine with 2.4GHz CPU speed and 60GB of memory. We measure both symbolic steps and run time, but only present the results on symbolic steps here, as they reflect the true symbolic time-complexity of the algorithms, and are independent of the choice of the underlying BDD package. The results on time are qualitatively the same. Each run is timed out after 400 seconds, indicated as the graph taking 10^9 symbolic steps on the figures. Since our input relation is partitioned into

[†]Tool and data set available at `https://doi.org/10.5281/zenodo.10427894`.

several sub-relations $E = (R_1, \ldots, R_k)$, each Pre/Post operation incurs k symbolic steps (for all algorithms). Our setup is completely deterministic, however certain operations, like Pick'ing a node, are executed arbitrarily.

Experimental results. We now present our experimental results for addressing the above research questions. Note that all figures are plotted in log-scale.

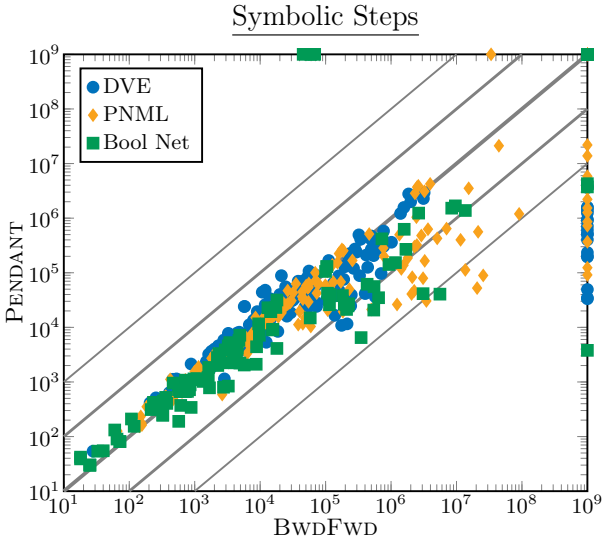

Fig. 2: The number of symbolic steps executed by Pendant and BwdFwd.

RQ1*:* Pendant *vs* BwdFwd. The performance of Pendant and BwdFwd is shown in Fig. 2, across all three datasets. Both algorithms manage to handle many models within the time limit, though there are a few time outs. We see that Pendant is generally no slower than BwdFwd, with the clear exception of three timeout outliers. For the rest, the models that are slower for Pendant sit only slightly above the $x = y$ line, meaning that the slowdown is comparatively small. On the other hand, there are several models on which Pendant is generally faster than BwdFwd, and the speedup increases as we go towards more demanding benchmarks (more than two orders of magnitude). Finally, Pendant times out on much fewer models than BwdFwd. Overall, Pendant is measurably faster than BwdFwd, and this trend persists across all three datasets (DVE, PNML and Boolean Networks).

RQ2*: The impact of deadlock detection.* The impact of deadlock detection to both Pendant and BwdFwd is shown in Fig. 3. We see that deadlock detection improves the performance of both algorithms significantly. Indeed, detecting

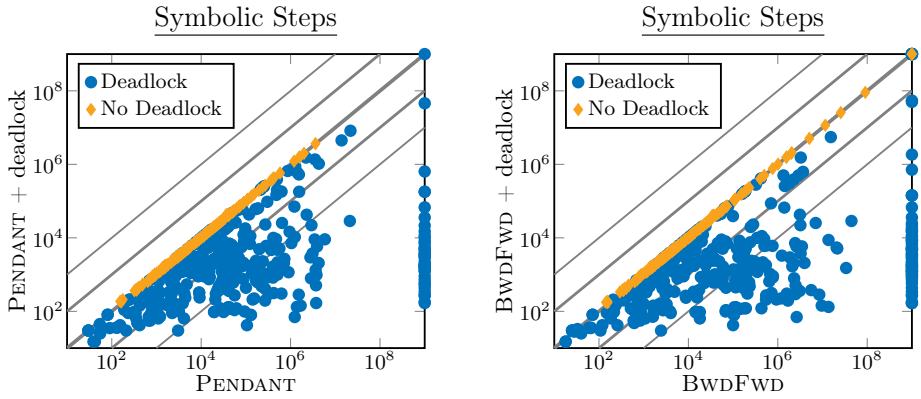

Fig. 3: The impact of deadlock detection in the number of symbolic steps executed by Pendant (left) and BwdFwd (right). Data points are classified as those having at least one deadlock, and those having no deadlock.

deadlocks requires only one symbolic step (per relation R_i), hence it is natural to expect that it does not slow down any algorithm, and has no effect on models that have no deadlocks. On the other hand, it leads to a measurable speedup on the models that have deadlocks, and the impact varies depending on the fraction of the graph that is removed during deadlock removal. Interestingly, deadlock detection also reduces significantly the number of timeouts for both Pendant and BwdFwd. In conclusion, deadlock detection helps both algorithms.

RQ3*: The impact of ITGR.* The impact of ITGR to both Pendant and BwdFwd is shown in Fig. 4. Perhaps surprisingly, we find that ITGR does not have a consistent effect: it can both speed up and slow down each of the algorithms. At closer inspection, we observe that ITGR has a positive effect on most Boolean Network models, which is indeed the context in which it was introduced [6]. On the other hand, it has both positive and negative effects on DVE and PNML models, and even makes both algorithms time out on instances that they could easily handle without ITGR.

RQ4*:* Pendant *vs* BwdFwd*, with deadlock detection.* Since deadlock detection has a clear positive effect on both algorithms, it is natural to revisit **RQ1** and ask about the performance of the two algorithms when also using deadlock detection. The result is shown in Fig. 5. Deadlock detection makes the performance of the two algorithms more similar in many benchmarks (i.e., more data points lie closer on the $x = y$ line). However, Pendant remains decisively faster on many models, and thus its benefit is not overshadowed by the positive impact of deadlock detection. At closer inspection, we see that Pendant is faster on DVE and PNML models, but not on Boolean Networks. This is due to the fact that

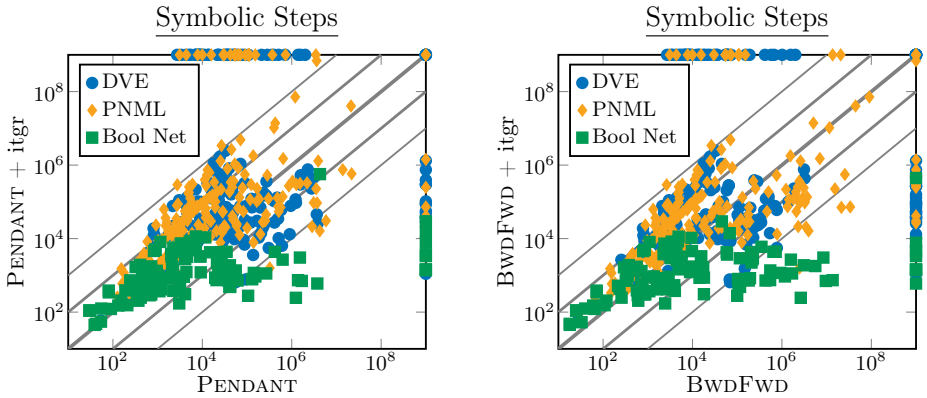

Fig. 4: The impact of ITGR in the number of symbolic steps executed by Pendant (left) and BwdFwd (right).

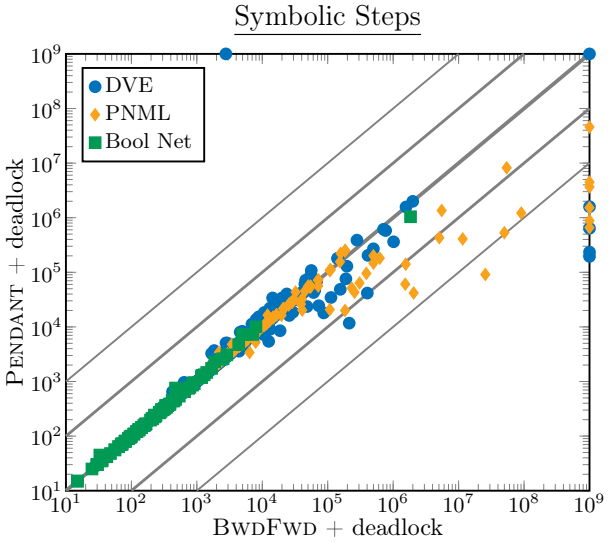

Fig. 5: The number of symbolic steps executed by Pendant and BwdFwd, when also using deadlock detection.

most Boolean Networks have many deadlocks, and thus the common deadlock-detection component simplifies such models considerably, making the remaining performance of the two algorithms similar.

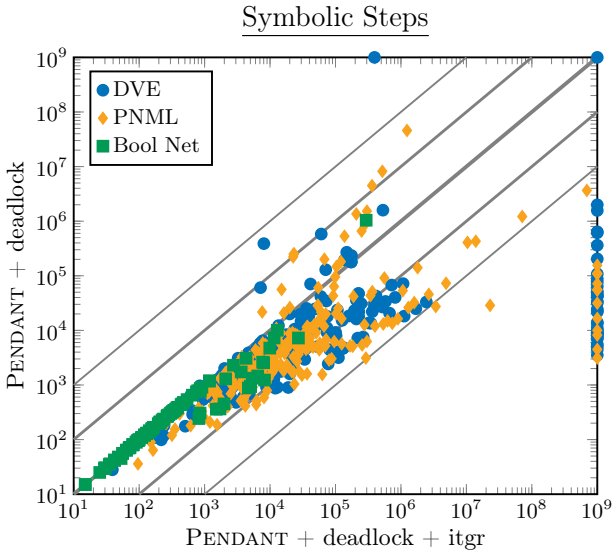

Fig. 6: The impact of ITGR after using deadlock detection.

RQ5*: The impact of ITGR after deadlock detection.* Finally, in Fig. 6 we examine whether ITGR improves the performance of PENDANT after deadlock detection has run. Although ITGR improves the performance on a few models, it generally leads to a slowdown, as well as to more timeouts. Interestingly, ITGR has the fewest positive effects (on top of deadlock detection) for Boolean Network models, for which it was originally introduced. Since these models have several deadlocks, the fast deadlock-detection preprocessing simplifies them considerably, at which point the cost of ITGR is not worth its little (or no) impact.

6 Conclusion

We have introduced PENDANT, a new symbolic algorithm for computing BSCCs, as well as a deadlock-detection technique for this task. Though both PENDANT and the standard BWDFWD have $O(n)$ symbolic-time complexity, our experimental results show that PENDANT is typically faster, and thus to be preferred for this task. Moreover, deadlock-detection is an efficient and effective preprocessing technique for reporting singleton BSCCs (and removing their basin),

before handing the computation to the general algorithm. Finally, the recently introduced ITGR, although effective on Boolean Network models, has mixed effects on DVE and PNML models, while its effect is often negative after deadlock detection (but not always). Some opportunities for future research include introducing saturation techniques [34] to Pendant, extending the algorithm to symbolically handle colored graphs [7,25], and understanding better the settings in which ITGR is effective.

Acknowledgements. This work was supported in part by Villum Fonden (Project VIL42117).

References

1. EMBL-EBI's BioModels model repository (2023), https://www.ebi.ac.uk/biomodels/, Last accessed on 2023-10-10
2. PyBoolNet model repository (2023), https://github.com/hklarner/pyboolnet/tree/master/pyboolnet/repository, Last accessed on 2023-10-10
3. SBML models repository (2023), https://github.com/sybila/biodivine-lib-param-bn/tree/master/sbml_models, Last accessed 2023-10-10
4. Abraham, E., Jansen, N., Wimmer, R., Katoen, J.P., Becker, B.: DTMC model checking by SCC reduction. In: Proceedings of the 2010 Seventh International Conference on the Quantitative Evaluation of Systems. p. 37–46. QEST '10, IEEE Computer Society, USA (2010). https://doi.org/10.1109/QEST.2010.13
5. Amparore, E.G., Donatelli, S., Gallà, F.: starMC: an automata based CTL* model checker. PeerJ Comput. Sci. **8**, e823 (2022)
6. Benes, N., Brim, L., Pastva, S., Safránek, D.: Computing bottom SCCs symbolically using transition guided reduction. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, CAV 2021, Part I. LNCS, vol. 12759, pp. 505–528. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_24
7. Benes, N., Brim, L., Pastva, S., Safránek, D.: BDD-based algorithm for SCC decomposition of edge-coloured graphs. Logical Methods in Computer Science **18**(1) (2022). https://doi.org/10.46298/lmcs-18(1:38)2022
8. Bloem, R., Gabow, H.N., Somenzi, F.: An algorithm for strongly connected component analysis in n log n symbolic steps. Formal Methods in System Design **28**(1), 37–56 (2006)
9. Bloem, R., Ravi, K., Somenzi, F.: Efficient decision procedures for model checking of linear time logic properties. In: Proceedings of the 11th International Conference on Computer Aided Verification. p. 222–235. CAV '99, Springer (1999)
10. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
11. Buchholz, P., Katoen, J.P., Kemper, P., Tepper, C.: Model-checking large structured Markov chains. The Journal of Logic and Algebraic Programming **56**(1), 69–97 (2003). https://doi.org/https://doi.org/10.1016/S1567-8326(02)00067-X, probabilistic Techniques for the Design and Analysis of Systems
12. Chatterjee, K., Dvořák, W., Henzinger, M., Loitzenbauer, V.: Lower bounds for symbolic computation on graphs: Strongly connected components, liveness, safety, and diameter. In: Proc. 29th ACM-SIAM Symp. on Discrete Algorithms. p. 2341–2356. SODA '18, Soc. for Industrial and Applied Mathematics, USA (2018)

13. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: Proc. 22nd ACM-SIAM Symp. on Discrete Algorithms. p. 1318–1336. SODA '11, Society for Industrial and Applied Mathematics, USA (2011)
14. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002)
15. van Dijk, T., van de Pol, J.: Sylvan: multi-core framework for decision diagrams. Int. Journal on Software Tools for Technology Transfer **19**(6), 675–696 (2017)
16. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: Proc. 7th IC on Tools and Algorithms for the Construction and Analysis of Systems. p. 420–434. TACAS 2001, Springer (2001)
17. Gentilini, R., Piazza, C., Policriti, A.: Computing strongly connected components in a linear number of symbolic steps. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. p. 573–582. SODA '03, Society for Industrial and Applied Mathematics, USA (2003)
18. Hardin, R.H., Kurshan, R.P., Shukla, S.K., Vardi, M.Y.: A new heuristic for bad cycle detection using BDDs. Form. Methods Syst. Des. **18**(2), 131–140 (mar 2001). https://doi.org/10.1023/A:1008727508722
19. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press (2004)
20. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-performance language-independent model checking. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 692–707. Springer (2015)
21. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model checking with strong fairness. Formal Methods Syst. Des. **28**(1), 57–84 (2006)
22. Kordon, F., Garavel, H., Hillah, L., Paviot-Adet, E., Jezequel, L., Hulin-Hubard, F., Amparore, E.G., Beccuti, M., Berthomieu, B., Evrard, H., Jensen, P.G., Botlan, D.L., Liebke, T., Meijer, J., Srba, J., Thierry-Mieg, Y., van de Pol, J., Wolf, K.: MCC'2017 - the seventh model checking contest. Trans. Petri Nets Other Model. Concurr. **13**, 181–209 (2018)
23. Kučera, A., Stražovský, O.: On the controller synthesis for finite-state Markov decision processes. In: Sarukkai, S., Sen, S. (eds.) FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science. pp. 541–552. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
24. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV. Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer (2011)
25. Larsen, C.A., Schmidt, S.M., Steensgaard, J., Jakobsen, A.B., van de Pol, J., Pavlogiannis, A.: A truly symbolic linear-time algorithm for SCC decomposition (2023)
26. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: an open-source model checker for the verification of multi-agent systems. Int. J. Softw. Tools Technol. Transf. **19**(1), 9–30 (2017)
27. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: SPIN. Lecture Notes in Computer Science, vol. 4595, pp. 263–267. Springer (2007)
28. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Strength-Based Decomposition of the Property Büchi Automaton for Faster Model Checking. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 580–593. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_42

29. Saadatpour, A., Albert, I., Albert, R.: Attractor analysis of asynchronous boolean models of signal transduction networks. Journal of Theoretical Biology **266**(4), 641–656 (2010). https://doi.org/https://doi.org/10.1016/j.jtbi.2010.07.022
30. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: TACAS. Lecture Notes in Computer Science, vol. 3440, pp. 174–190. Springer (2005)
31. Wang, C., Bloem, R., Hachtel, G.D., Ravi, K., Somenzi, F.: Divide and compose: SCC refinement for language emptiness. In: Proceedings of the 12th International Conference on Concurrency Theory. p. 456–471. CONCUR '01, Springer-Verlag, Berlin, Heidelberg (2001)
32. Xie, A., Beerel, P.A.: Implicit enumeration of strongly connected components and an application to formal verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19**(10), 1225–1230 (2000)
33. Yuan, Q., Mizera, A., Pang, J., Qu, H.: A new decomposition-based method for detecting attractors in synchronous boolean networks. Science of Computer Programming **180**, 18–35 (2019). https://doi.org/https://doi.org/10.1016/j.scico.2019.05.001
34. Zhao, Y., Ciardo, G.: Symbolic computation of strongly connected components and fair cycles using saturation. Innov. Syst. Softw. Eng. **7**(2), 141–150 (2011)