Innovations Syst Softw Eng (2011) 7:141–150
DOI 10.1007/s11334-011-0146-3

SI: NFM 2010

Symbolic computation of strongly connected components and fair cycles using saturation

Yang Zhao · Gianfranco Ciardo

Received: 9 September 2010 / Accepted: 26 February 2011 / Published online: 11 March 2011

Abstract The computation of strongly connected components (SCCs) in discrete-state models is a critical step in formal verification of LTL and fair CTL properties, but the potentially huge number of reachable states and SCCs constitutes a formidable challenge. We consider the problem of computing the set of states in SCCs or terminal SCCs in an asynchronous system. We employ the idea of *saturation*, which has shown clear advantages in symbolic state-space exploration (Ciardo et al. in Softw Tools Technol Transf 8(1):4–25, 2006; Zhao and Ciardo in Proceedings of 7th international symposium on automated technology for verification and analysis, pp 368–381, 2009), to improve two previously proposed approaches. We use saturation to speed up state exploration when computing each SCC in the *Xie-Beerel algorithm*, and we compute the *transitive closure* of the transition relation using a novel algorithm based on saturation. Furthermore, we show that the techniques we developed are also applicable to the computation of *fair cycles*. Experimental results indicate that the improved algorithms using saturation achieve a substantial speedup over previous BFS algorithms. In particular, with the new transitive closure computation algorithm, up to 10^{150} SCCs can be explored within a few seconds.

Keywords Formal verification · Strongly connected component · Symbolic model checking

Y. Zhao (✉) · G. Ciardo
Department of Computer Science and Engineering,
University of California, Riverside, USA
e-mail: zhaoy@cs.ucr.edu

G. Ciardo
e-mail: ciardo@cs.ucr.edu

1 Introduction

Finding strongly connected components (SCCs) is a basic problem in graph theory. For discrete-state models, some interesting properties, such as those expressible in LTL [19] and fair CTL [12], are correlated with the existence of SCCs in the state-transition graph. The same problem is also central to the language emptiness check for ω-automata [17, 19]. For large discrete-state models, it is impractical to find SCCs using explicit depth-first state-space search [26] since its complexity is *at least* linear in the size of the graph, motivating the study of symbolic SCC computation. In this paper, the objective was to build the set of states in SCCs.

The structure of SCCs in a graph is captured by its *quotient graph*, obtained by collapsing each SCC into a single node. This graph is acyclic, thus defines a partial order on the SCCs. *Terminal SCCs* (or *bottom SCCs*) are nontrivial SCCs corresponding to leaf nodes in the quotient graph. For Markov chains [25], it is important to classify the reachable states as *recurrent* (belonging to terminal SCCs) or *transient* (all other states) since the probability that a Markov chain returns to a given state equals 1 iff that state is recurrent.

The complexity of these problems arises from two aspects: having to explore a huge state space (almost always the case in real-life problems) and having to deal with a large number of SCCs or terminal SCCs (sometimes the case in real-life problems). The former, known as *state explosion* [12], is the main obstacle to formal verification due to the obvious burden it imposes on computational resources. Traditional BDD approaches cope with this problem by employing *image* and *preimage* computations for state-space exploration but, while successful for fully synchronous systems [5], they do not work as well for asynchronous systems [8]. The latter constitutes the bottleneck for one class of previous work [28,29],

which enumerates SCCs one by one. Section 2.3 analyzes this problem in more detail.

We address the computation of states in SCCs or terminal SCCs by improving two previous approaches: one proposed by Xie-Beerel (XB) [28,29] and one based on computing the transitive closure (TC) of the transition relation [17]. We apply the saturation algorithm [8] to both to cope with the cost of state-space exploration. Our previous work demonstrated clear advantages for saturation in state-space generation [8] (summarized in Sect. 2.2) and CTL model checking [9,30] over traditional symbolic approaches. In this paper, we employ saturation for SCC analysis. Saturation greatly improve the XB algorithm over its original version using BFS. With regard to a potentially huge number of SCCs, the TC algorithm has the advantage of exploring all SCCs symbolically instead of enumerating them one by one. However, as previously proposed, computing the TC often requires large amount of runtime and memory [17], to the point that [22] claims that the TC algorithm is "infeasible for large examples". To disprove this myth, we propose a new saturation algorithm to compute the TC, making it a practical method of SCC computation for complex systems. Furthermore, we present an algorithm to compute the recurrent states (i.e., states in terminal SCCs) based on the TC.

Then we consider fair cycle detection, a problem related to SCC computation, but even more challenging. In Sect. 6, we discuss algorithms for detecting fair cycles satisfying Streett fairness (strong fairness) [23].

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on the data structures we use and saturation. Section 3 presents a constrained saturation algorithm that limits state-space exploration to a given set of states. Section 4 introduces an improved XB algorithm using saturation. Section 5 proposes our new algorithm to compute the TC using saturation and the corresponding algorithms for SCC and terminal SCC computation. Section 6 deals with the problem of finding fair cycles. Section 7 compares the performance of our algorithms and Lockstep. We discuss future work in the last section.

2 Preliminaries

Consider a discrete-state model $(\mathcal{S}, \mathcal{S}_{\text{init}}, \mathcal{E}, \mathcal{N})$ where:

- the potential state space $\mathcal{S}$ is given by the product $\mathcal{S}_L \times \cdots \times \mathcal{S}_1$ of the finite local state spaces of L submodels, so that each (global) state $\mathbf{i}$ is a tuple $(i_L, \ldots, i_1)$, where $i_k \in \mathcal{S}_k$, for $L \geq k \geq 1$;
- the set of initial states is $\mathcal{S}_{\text{init}} \subseteq \mathcal{S}$;
- the set of (asynchronous) events is $\mathcal{E}$;
- the next-state function $\mathcal{N} : \mathcal{S} \to 2^{\mathcal{S}}$ is described in disjunctively partitioned form as $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$, where $\mathcal{N}_\alpha(\mathbf{i})$ is the set of states that can be nondeterministically reached in one step when event α fires in state $\mathbf{i}$.

We say that α is *enabled* in state $\mathbf{i}$ if $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$. Correspondingly, $\mathcal{N}^{-1}$ and $\mathcal{N}_\alpha^{-1}$ denote the previous-state functions, e.g., $\mathcal{N}_\alpha^{-1}(\mathbf{i})$ is the set of states that can reach $\mathbf{i}$ in one step by firing event α.

In this paper, $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{k}$ denote single states, $\mathcal{X}$ and $\mathcal{Y}$ denote sets of states, and $\mathcal{N}$, possibly with a subscript or superscript, denotes a next-state function.

Locality is a fundamental property enjoyed by asynchronous systems, expresses the fact that most events affect only few systems components:

- An event α is *independent* of the kth submodel if its enabling does not depend on i_k and its firing does not change the value of i_k.
- If α is not independent of kth submodel, k is in the *support* of event α, written as $k \in supp(\alpha)$.
- Let $Top(\alpha)$ be the top index in $supp(\alpha)$, and $\mathcal{E}_k$ be the set $\{\alpha \in \mathcal{E} : Top(\alpha) = k\}$.
- Let $\mathcal{N}_k$ be the next-state function corresponding to all events in $\mathcal{E}_k$, i.e., $\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$.

The problem of state-space generation refers to computing the set $\mathcal{S}_{\text{rch}}$ of states reachable from $\mathcal{S}_{\text{init}}$. Section 2.2 recalls our state-space generation algorithm, *saturation*, executed prior to SCC computation as a preprocessing step. Thus, $\mathcal{S}_{\text{rch}}$, the sets $\mathcal{S}_k$, and their sizes n_k are assumed known in the following discussion, and we let $\mathcal{S}_k = \{0, \ldots, n_k - 1\}$ without loss of generality.

2.1 Symbolic encoding of discrete-state systems

We employ *multi-way decision diagrams* (MDDs) [18] to encode discrete-state systems. MDDs extend binary decision diagrams (BDDs) by allowing integer-valued variables, thus are suitable for discrete-state models with unknown (but hoped to be bounded) state spaces, such as Petri nets [21]. An MDD has two possible terminal nodes, **0** and **1**, and a single root node.

A set of states is encoded with an L-level *quasi-reduced* MDD. Given a node a, its level is denoted with $a.lvl$ where $L \geq a.lvl \geq 0$, $a.lvl = 0$ if a is **0** or **1**, and $a.lvl = L$ if a is the root node. If $a.lvl = k > 0$, then a has n_k outgoing edges, each labeled with a different $i_k \in \mathcal{S}_k$. The node pointed by the edge labeled with i_k is denoted $a[i_k]$ and, if not **0**, it must be at level $k - 1$.

Turning to the encoding of the next-state functions, locality can be exploited to obtain a compact symbolic expression. Since $\alpha \in \mathcal{E}_k$ does not affect $i_L, \ldots, i_{k+1}$, $\mathcal{N}_\alpha((i_L, \ldots, i_1))$ can be obtained as $\{(i_L, \ldots, i_{k+1})\} \times \mathcal{N}_\alpha((i_k, \ldots, i_1))$. In other words, $\mathcal{N}_\alpha$ only needs to be applied to the lower k state

components, thus, when a set of states is encoded with an MDD, firing $\mathcal{N}_\alpha$ on them only modifies the sub-MDDs rooted at level k of the MDD. The next-state function $\mathcal{N}$ is encoded using a $2L$-level MDD, with levels ordered $L, L', \ldots, 1, 1'$, where unprimed and primed levels describe "from" and "to" states, respectively, and we let $U(k) = U(k') = k$. We use *quasi-identity-fully (QIF)* reduced [27] MDDs to encode next-state functions. This means that, for $\alpha \in \mathcal{E}_k$, $\mathcal{N}_\alpha$ is encoded with a $2k$-level MDD, so that the application of $\mathcal{N}_\alpha$ only needs to start at level $Top(\alpha)$.

We use lowercase letters to denote MDD nodes and refer to an MDD by its root node, thus, "MDD a" means "the MDD rooted at node a". MDD a encodes a set $\mathcal{B}(a) \subseteq \mathcal{S}_{a.lvl} \times \cdots \times \mathcal{S}_1$ of (sub)-states, corresponding to the paths from a to $\mathbf{1}$. However, to simplify notation, we identify some "global" quantities with their MDDs, e.g., $\mathcal{N}$ means either the next-state function or the MDD encoding it, depending on the context.

Given a set of states $\mathcal{X} = \mathcal{B}(x) = \subseteq \mathcal{S}$, the computation of $\mathcal{N}_\alpha(\mathcal{X})$ can be implemented with the symbolic operator *RelProd*, which takes an L-level MDD and a $2L$-level MDD in input and returns an L-level MDD:

$$\mathcal{N}_\alpha(\mathcal{X}) = \mathcal{B}(RelProd(x, \mathcal{N}_\alpha)).$$

In our pseudocode, we use the following functions:

$$\text{(forward image)} \quad ImageF(x) \triangleq RelProd(x, \mathcal{N}),$$
$$\text{(backward image)} \quad ImageB(x) \triangleq RelProd(x, \mathcal{N}^{-1}).$$

Using MDDs does not lead to higher efficiency than BDDs, or even than explicit approaches (decision diagrams are, after all, heuristics). Furthermore, as with BDDs, the chosen *variable order* greatly affects the size and efficiency of MDDs, but choosing an optimal variable order is hard [3]. We use MDDs instead of BDDs in our work because it is easier to cope with models whose local state spaces are not known a priori, such as Petri nets, by letting the size of MDD nodes grow on the fly during state-space generation [8,27].

2.2 State-space generation using saturation

All symbolic x state-space generation algorithms use some variant of symbolic image computation. The simplest approach is a breadth-first search (BFS) directly implementing the definition of $\mathcal{S}_{\text{rch}}$ as the fixed point $\mathcal{S}_{\text{init}} \cup \mathcal{N}(\mathcal{S}_{\text{init}}) \cup \mathcal{N}^2(\mathcal{S}_{\text{init}}) \cup \cdots$ The forward reachable states from a set of states $\mathcal{X}$ can be computed by $ReachF(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}(\mathcal{X}) \cup \mathcal{N}^2(\mathcal{X}) \cup \cdots$ Analogously, we let $ReachB(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}^{-1}(\mathcal{X}) \cup \mathcal{N}^{-2}(\mathcal{X}) \cup \cdots$

Saturation employs a different approach, recursively computing *local fixpoints* and exploiting locality in the next-state functions. The key idea is to fire events in an order consistent with their *Top*: we attempt firing events in $\mathcal{E}_k$ on node a at level k only after having exhaustively fired events in $\mathcal{E}_h$, for all $h < k$, on nodes below a, until no new states are found. Then, a is *saturated* if it is a fixed point w.r.t. events independent of the levels above k: $\forall h, k \geq h \geq 1, \forall \alpha \in \mathcal{E}_h, \mathcal{B}(a) \supseteq \mathcal{N}_\alpha(\mathcal{B}(a))$.

```
mdd SaturateF(mdd s)
 1 if InCache_SaturateF(s,t) then return t;      • don't repeat work
 2 level k ← s.lvl;
 3 mdd t ← 0;
 4 foreach i ∈ S_k s.t. s[i] ≠ 0 do      • first, saturate nodes below
 5   t[i] ← SaturateF(s[i]);
 6 endfor;
 7 repeat                  • keep firing N_k until reaching convergence
 8   foreach i, i' ∈ S_k s.t. N_k[i][i'] ≠ 0 do
 9     t[i'] ← Union(t[i'], RelProdSat(t[i], N_k[i][i']));
10   endfor;
11 until t does not change;
12 t ← InsertUT(t);      • Unique Table to avoid duplicate nodes
13 CacheAdd_SaturateF(s,t);      • to avoid repeating work later
14 return t;
```

```
mdd RelProdSat(mdd s, r)
 1 level k ← s.lvl;
 2 mdd t ← 0;
 3 if s = 1 and r = 1 then return 1; endif;      • terminal case
 4 if InCache_RelProdSat(s,r,t) then return t; endif;
 5 foreach i, i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 6   t[i'] ← Union(t[i'], RelProdSat(s[i], r[i][i']));
 7 endfor;
 8 t ← InsertUT(t);      • analogous to RelProd until here...
 9 t ← SaturateF(t);  • ...but now we saturate t before returning
10 CacheAdd_RelProdSat(s,r,t);
11 return t;
```

Fig. 1 The (forward) saturation algorithm

Figure 1 shows the pseudocode of the saturation algorithm. Given a discrete-state model, $\mathcal{N}_L, \ldots, \mathcal{N}_1$ are globally available. The pseudocode shows a forward exploration, as it uses the next-state functions, but it can naturally be used for backward exploration by replacing $\mathcal{N}_k$ with $\mathcal{N}_k^{-1}$. The input parameter s is the root node to be saturated; thus it is initially set to the root of the MDD encoding $\mathcal{S}_{\text{init}}$. *SaturateF* saturates the nodes of MDD s in order, from the bottom level to the top level. Unlike the traditional relational product operation, *RelProdSat* always returns a saturated MDD.

2.3 Previous work

Symbolic SCC analysis has been widely explored [13–16,22,24], and much effort has been spent on computing the *SCC hull*. A family of SCC hull algorithms [24] employing BFS iteration is available. Although closely related, there is crucial difference between SCC hull computation and our work, because an SCC hull contains not only states in nontrivial SCCs, but also states on the paths between them, constituting a superset of our desired result. In non-probabilistic model checking, computing the SCC hull aims at detecting the *existence* of reachable fair cycles, which is critical to verify fair

CTL, LTL and ω-automata properties. However, for Markov chains, the aim of SCC analysis is to collect the states belonging to (trivial or nontrivial) terminal SCCs [1,10]. To this end, an SCC hull must be further decomposed into states in an SCC and those not. To the best of our knowledge, no existing efficient symbolic algorithm is available for this task. Hence, SCC hull computation algorithms are *not* applicable to our problem, and we will not discuss them further. Instead, we review two categories of related previous work, which can compute the precise set of states in SCCs: the TC and the XB algorithms.

Hojati et al. [17] presented a symbolic algorithm to test ω-regular language containment by computing the TC $\mathcal{N} \cup \mathcal{N}^2 \cup \mathcal{N}^3 \cup \cdots$ Section 5 discusses the TC in more detail. Matsunaga et al. [20] proposed a symbolic procedure to compute the TC, but the runtime is so poor that building the TC has long been considered impractical for complex systems.

Xie et al. [29] proposed an algorithm, referred to as the XB algorithm in this paper, combining explicit state enumeration and symbolic state-space exploration. They explicitly pick a state as a "seed", compute the forward and backward reachable states from the seed, and find the SCC containing this seed as the intersection of these two sets of states. Bloem et al. [2] presented an improved algorithm called *Lockstep* (Fig. 2) which, given a seed, instead of computing the forward and backward reachable states separately, it alternates BFS iterations between the two so that it can stop as soon as one of the two converges. This optimization achieves a $O(n \log n)$ complexity, compared to the $O(n^2)$ of the XB algorithm, computed in terms of number of image and preimage computations, where n is the number of reachable states. Our experiments show that Lockstep works very well when the number of SCCs is manageable. However, as the number of SCCs grows, the exhaustive enumeration of SCCs becomes a formidable problem for both the XB and the Lockstep algorithms.

Xie et al. [28] proposed a similar idea to compute the recurrent states in large Markov chains (*XB_TSCC* in Fig. 2). From a randomly picked seed, if the set of forward reachable states $\mathcal{B}(f)$ is a subset of the set of backward reachable states $\mathcal{B}(b)$, then $\mathcal{B}(f)$ is a terminal SCC; otherwise, $\mathcal{B}(b)$ contains no terminal SCC and can be eliminated from further consideration. Functions *ReachF* and *ReachB* were implemented using BFS.

Our work is the first to employ saturation in SCC analysis. The two approaches we propose build upon the above two previous approaches. For the XB algorithm, we replace BFS state-space exploration with saturation. For the TC approach, we propose a new algorithm to compute TC using saturation.

```
mdd  Lockstep(mdd p)                          • initially, B(p) = S_rch
 1  if (p = 0) then return 0;
 2  mdd s ← PickState(p);                     • pick a seed s from B(p)
 3  mdd a ← 0;                                • a accumulates the answer
 4  mdd f ← 0;                                • f accumulates the forward set from s
 5  mdd b ← 0;                                • b accumulates the backward set from s
 6  mdd c ← 0;                                • c will be the SCC containing s, if any
 7  mdd f' ← Intersect(ImageF(s), p);
 8  mdd b' ← Intersect(ImageB(s), p);
 9  while f' ≠ 0 and b' ≠ 0 do
10    f ← Union(f, f');
11    b ← Union(b, b');
12    f' ← Diff(Intersect(ImageF(f'), p), f);
13    b' ← Diff(Intersect(ImageB(b'), p), b);
14  endwhile;
15  if (f' = 0) then
16    mdd v ← f;                              • v remembers the set that converged first, f
17    while Intersect(b', f) ≠ 0 do           • continue exploring backward
18      b' ← Diff(Intersect(ImageB(b'), p), b);
19      b ← Union(b, b');
20    endwhile;
21  else
22    mdd v ← b;                              • v remembers the set that converged first, b
23    while Intersect(f', b) ≠ 0 do           • continue exploring forward
24      f' ← Diff(Intersect(ImageF(f'), p), f);
25      f ← Union(f, f');
26    endwhile;
27  endif;
28  if Intersect(f, b) ≠ 0 then
29    c ← Intersect(f, b);                    • s belongs to the nontrivial SCC c
30  endif;
31  a ← Union(c, Lockstep(Diff(v,c)), Lockstep(Diff(p, Union(v,s))));
      • divide and conquer in the remaining state space
32  return a;
```

```
mdd  XB_TSCC(mdd p)                           • initially, B(p) = S_rch
 1  mdd a ← 0;                                • a collects the states in nontrivial terminal SCCs
 2  mdd s, f, b;
 3  while (p ≠ 0) do
 4    s ← PickState(p);                       • pick a seed s from B(p)
 5    f ← Intersect(ReachF(s), p);
 6    b ← Intersect(ReachB(s), p);
 7    if Diff(f, b) = 0 then                  • B(f) is a subset of B(b)
 8      a ← Union(a, f);                      • add the new terminal SCC to a
 9    endif;
10    p ← Diff(p, b);                         • don't consider the states in B(b) again
11  endwhile;
12  return a;
```

Fig. 2 Lockstep for SCC computation and XB algorithm for terminal SCC computation

3 Constrained saturation

The motivation behind the *constrained saturation* algorithm [30] is the computation of the CTL operator $\mathsf{E}[a\mathsf{U}b]$, which returns the set of states backward reachable from states satisfying property b through paths satisfying property a. To constrain the state-space exploration to a set of states a, a BFS approach can simply intersect the newly reached states with a after each step. If we used a similar idea in ordinary saturation, however, intersection operations would have to be executed after each event firing, destroying the efficiency of saturation. Constrained saturation uses instead a "check-and-fire" strategy that constrains the exploration to a in each recursive call of the relational product. The idea of constrained saturation is based on the following observation:

```
mdd ConsSatB(mdd a, mdd s)
 1  if InCache_ConsSatB(a, s, t) then return t;
 2  level l ← s.lvl;
 3  mdd t ← 0;
 4  foreach i ∈ S_l s.t. s[i] ≠ 0 do      • first, saturate nodes below
 5⋆  if a[i] ≠ 0 then
 6⋆    t[i] ← ConsSatB(a[i], s[i]);
 7⋆  else                  • no new state in this sub-space since a[i] ≠ 0
 8⋆    t[i] ← s[i];
 9⋆  endif;
10  endfor;
11  repeat                                • compute the local fixed point
12    foreach i, i' ∈ S_l s.t. N_l^{-1}[i][i'] ≠ 0 do
13⋆    if a[i'] ≠ 0 then • execute only if the resulting states are in a
14       t[i'] ← Union(t[i'], ConsRelProdB(a[i'], t[i], N_l^{-1}[i][i']));
15⋆    endif;
16  until t does not change;
17  t ← InsertUT(t);
18  CacheAdd_ConsSatB(a, s, t);
19  return t;
```

```
mdd ConsRelProdB(mdd a, mdd s, mdd r)
 1  if s = 1 and r = 1 then return 1;
 2  if InCache_ConsRelProdB(a, s, r, t) then return t;
 3  level l ← s.lvl;
 4  mdd t ← 0;
 5  foreach i, i' ∈ S_l s.t. r[i][i'] ≠ 0 do
 6⋆  if a[i'] ≠ 0 then  • execute only if the resulting states are in a
 7⋆    t[i'] ← Union(t[i'], ConsRelProdB(a[i'], s[i], r[i][i']));
 8⋆  endif;
 9  endfor;
10  t ← ConsSatB(a, InsertUT(t));  • saturate t before returning
11  CacheAdd_ConsRelProdB(a, s, r, t);
12  return t;
```

Fig. 3 Constrained (backward) saturation to compute EU

$$\mathcal{B}(t) = RelProd(s, r) \cap \mathcal{B}(a) \Rightarrow$$
$$\mathcal{B}(t[i']) = \bigcup_{\forall i \in \mathcal{S}_l} \left(RelProd(s[i], r[i][i']) \cap \mathcal{B}(a[i'])\right),$$

where t and s are l-level MDDs encoding sets of states and r is a $2l$-level MDD encoding a next-state function. This can be considered an extension of the well-known *ITE* operator [4], from boolean to integer variables. The overall process of EU computation based on constrained saturation is shown in Fig. 3, where a is the constraint and s is the set being saturated. The key differences from the saturation algorithm of [8] are marked with a "⋆". The new states found by applying a next-state function are guaranteed to satisfy a.

Neither saturation nor constrained saturation improves the theoretical computational complexity compared to the corresponding BFS algorithms. First, saturation could degrade to BFS if all events fall in $\mathcal{E}_L$. Second, both saturation and BFS may perform as bad as enumerating states one by one in the worst case. However, [8,Theorem 1] demonstrates that saturation can avoid building many MDD node that only appear in intermediate results, thus tends to result in much more compact MDDs. Saturation is experimentally superior to BFS for the local-synchronous-global-asynchronous systems we studied. This is also reflected in the comparison with Lockstep in the next section.

4 Using saturation in the XB algorithm

A natural improvement to the XB algorithm is to employ saturation to explore the forward and backward reachable states. Figure 4 shows the pseudocode of our algorithms: the two versions of *XBSat* shown in the figure compute the states in SCCs, or just in the terminal SCCs, respectively. Unlike Lockstep, which uses the set that converges first to bound the other, our algorithm cannot interleave the two computations or even predict which one will converge first, since saturation does not run in a step-by-step manner. One obvious approach is then to run a forward and a backward saturation and intersect the results. However, we experimentally found that we can do better by bounding one of the two saturations with the other. In Fig. 4, for example, we always "bet" on forward exploration and use it to bound backward exploration (Line 7). This bet is of course most beneficial when (unconstrained) backward exploration is much slower than forward exploration. Thus, there is a trade-off between using the slower BFS, which however allows us to interleave the two explorations, and the faster saturation, which does not. Performance is also affected by which seed is picked in each iteration, and for a fair comparison, we pick the same seed in both algorithms at each iteration.

Both our new algorithm and Lockstep improve the original XB algorithm in different ways. Lockstep aims at reducing

```
mdd XBSat(mdd p)                          • initially, B(p) = S_rch
 1  if p = 0 then return 0; endif;
 2  mdd a ← 0;
 3  mdd s ← PickState(p);                 • pick a seed s from B(p)
 4  mdd f' ← Intersect(ImageF(s), p);
 5  mdd b' ← Intersect(ImageB(s), p);
 6  mdd f ← Intersect(SaturateF(f'), p);
    • forward reachable states from s in p
 7  mdd b ← Intersect(ConsSatB(f, b'), p);
    • the intersection of backward and forward reachable states
 8  if b ≠ 0 then a ← b; endif;  • b≠∅ is a set of states in an SCC
 9  a ← Union(a, XBSat(Diff(f, b)), XBSat(Diff(p, f)));
    • divide and conquer in the remaining state space
10  return a;
 7' mdd b ← Intersect(SaturateB(b'), p);
    • backward reachable states from s in p
 8' if Diff(f, b) = 0 then a ← Union(a, f); endif;
    • B(f) is a terminal SCC if B(f) ⊆ B(b)
 9' a ← Union(a, XBSat(Diff(p, b)));
    • recursion in the remaining state space
10' return a;
```

Fig. 4 Improved XB algorithm using saturation (*lines 7–10* compute SCCs; *lines 7'–10'* compute terminal SCCs)

the number of steps by carefully scheduling the image and preimage computations, while our algorithm leverages event locality. Measuring complexity in number of BFS steps, $O(n \log n)$ for Lockstep [2], is not meaningful for saturation, which uses light-weight event firings instead of global image computations. Moreover, we argue that the number of steps is too rough a measure of complexity because the cost of each symbolic step varies greatly, exponentially with the number of levels in the worst case. Instead, saturation aims at reducing the real cost in symbolic manipulation: as mentioned in Sect. 3, it avoids building many unnecessary intermediate MDD nodes. Thus, while Lockstep ensures a tight bound on the number of steps, saturation often executes fewer node operations, thus lower runtime and memory requirements. Our experimental results show that, for most models we studied, the improved XB algorithm using saturation outperforms Lockstep, sometimes by orders of magnitude.

5 Computing the TC with saturation

We now define the forward and the backward TC, $\mathcal{T}$ and $\mathcal{T}^{-1}$, of a discrete-state model.

Definition 1 The transitive closure $\mathcal{T} \subseteq \mathcal{S}_{\text{rch}} \times \mathcal{S}_{\text{rch}}$ contains all $(\mathbf{i}, \mathbf{j})$ such that there is a nontrivial (i.e., positive length) path from $\mathbf{i}$ to $\mathbf{j}$, denoted by $\mathbf{i} \to \mathbf{j}$. Analogously, we let $(\mathbf{i}, \mathbf{j}) \in \mathcal{T}^{-1}$ iff $\mathbf{j} \to \mathbf{i}$.

$\mathcal{T}^{-1}$ represents relations between pairs of states, and thus, it can be encoded with a $2L$-level MDD with the same variable order with next-state functions. i_k is placed on level k (unprimed level) and j_k on level k' (primed level), where $U(k) = U(k') = k$. As $\mathcal{T}$ and $\mathcal{T}^{-1}$ are symmetric, we focus on the algorithm to compute $\mathcal{T}^{-1}$. After $\mathcal{T}^{-1}$ has been built, $\mathcal{T}$ is obtained by simply swapping the unprimed and primed levels in the MDD encoding $\mathcal{T}^{-1}$, written as $\mathcal{T} = \mathit{Inverse}(\mathcal{T}^{-1})$.

We base our algorithm on the following observation:

$$(\mathbf{i}, \mathbf{j}) \in \mathcal{T}^{-1} \Leftrightarrow \exists \mathbf{k} \in \mathcal{N}^{-1}(\mathbf{i}) \wedge \mathbf{j} \in \mathcal{B}(\mathit{ConsSatB}(\mathcal{S}_{\text{rch}}, s)),$$

where $\mathcal{B}(s) = \{\mathbf{k}\}$. Instead of running saturation on $\mathbf{j}$ for each pair $(\mathbf{i}, \mathbf{j})$, we propose an algorithm that executes on the $2L$-level MDD encoding $\mathcal{N}^{-1}$. Line 1 in function *SCC_TC* of Fig. 5 computes $\mathcal{T}^{-1}$ by calling *TransClosSat*, which runs bottom-up recursively. As for the constrained saturation in Fig. 3, this function runs node-wise on primed level and fires lower level events exhaustively until the local fixed point is obtained. This procedure ensures the following properties.

Property 1 Given a k-level MDD a and $2k$-level MDD n, $\mathit{TransClosSat}(a, n)$ returns a $2k$-level MDD t s.t. $\forall (\mathbf{i}, \mathbf{j}) \in \mathcal{B}(n), \mathbf{k} \in \mathcal{B}(\mathit{ConsSatB}(a, \mathbf{j})) \Rightarrow (\mathbf{i}, \mathbf{k}) \in \mathcal{B}(t)$.

Property 2 $\mathit{TransClosSat}(\mathcal{S}_{\text{rch}}, \mathcal{N}^{-1})) = \mathcal{T}^{-1}$.

```
mdd SCC TC(N^{-1})
1  mdd T^{-1} ← TransClosSat(S_rch, N^{-1});
2  mdd SCC ← TCtoSCC(T^{-1});
3  return SCC;
```

```
mdd TransClosSat(mdd a, mdd n)
1  if n = 1 then return 1; endif;
2  if InCache_TransClosSat(a, n, t) then return t; endif;
3  level k ← n.lvl;                                   • L ≥ U(k) ≥ 1
4  mdd t ← 0;
5  mdd r ← N^{-1}_{U(k)};
6  foreach i, j ∈ S_k s.t. n[i][j] ≠ 0 do
7    if a[j] ≠ 0 then                                 • first, saturate nodes below
8      t[i][j] ← TransClosSat(a[j], n[i][j]);
9    else                                             • no new path in this sub-space since a[i] ≠ 0
10     t[i][j] ← n[i][j];
11   endif;
12 endfor;
13 foreach i ∈ S_{U(k)} s.t. n[i] ≠ 0 do
14   repeat                                           • compute the local fixed point
15     foreach j,j' ∈ S_{U(k)} s.t. n[i][j] ≠ 0 ∧ r[j][j'] ≠ 0 ∧ a[j'] ≠ 0 do
       • r is applied to the primed levels of n
16       t[i][j'] ← Union(t[i][j'], TCRelProdSat(a[j'], t[i][j], r[j][j']));
17     endfor;
18   until t does not change;
19 endfor;
20 t ← InsertUT(t);
21 CacheAdd_TransClosSat(a, n, t);
22 return t;    • t is a 2L-level MDD encoding TransClosSat(a, n)
```

```
mdd TCRelProdSat(mdd a, mdd n, mdd r)
1  if n = 1 and r = 1 then return 1; endif;
2  if InCache_TCRelProdSat(a, n, r, t) then return t; endif;
3  level k ← n.lvl;                                   • L ≥ U(k) ≥ 1
4  mdd t ← 0;
5  foreach i ∈ S_{U(k)} s.t. n[i] ≠ 0 do
6    foreach j,j' ∈ S_{U(k)} s.t. n[i][j] ≠ 0 ∧ r[j][j'] ≠ 0 ∧ a[j'] ≠ 0 do
     • r is applied on primed levels in n
7      t[i][j'] ← Union(t[i][j'], TCRelProdSat(a[j'], n[i][j], r[j][j']));
8    endfor;
9  endfor;
10 t ← TransClosSat(a, InsertUT(t));   • return a saturated result
11 CacheAdd_TCRelProdSat(a, n, r, t);
12 return t;
```

```
mdd TCtoSCC(mdd n)
1  if n = 1 return 1; endif;
2  if InCache_TCtoSCC(n, t) then return t; endif;
3  level k ← n.lvl;                                   • L ≥ U(k) ≥ 1
4  mdd t ← 0;
5  foreach i ∈ S_{U(k)} s.t. n[i][i] ≠ 0 do
6    t[i] ← TCtoSCC(n[i][i]);
7  endfor;
8  t ← InsertUT(t);
9  CacheAdd_TCtoSCC(n, t);
10 return t;    • t is an L-level MDD enconding {i : (i, i) ∈ B(n)}
```

Fig. 5 Building the TC using saturation

The top-level pseudocode of the SCC computation using TC is shown as *SCC_TC* in Fig. 5. Function *TCtoSCC* extracts all states $\mathbf{i}$ such that $(\mathbf{i}, \mathbf{i}) \in \mathcal{T}^{-1}$. Unlike SCC enumeration algorithms such as XB or Lockstep, a TC approach does not necessarily suffer from a large number of SCCs.

```
mdd TSCC TC(N^-1)
1 mdd T^-1 ← TransClosSat(N^-1);
2 mdd T ← Inverse(T^-1);        • swap adjacent levels k and k'
3 mdd scc ← TCtoSCC(T^-1);
4 mdd r* ← Diff(T^-1, T);
5 mdd nontscc ← QuantUnpr(r*);
   • quantify out unprimed levels in r*, nontscc is an L-level MDD
6 mdd recurrent ← Diff(scc, nontscc);
7 return recurrent;
```

Fig. 6 Computing recurrent states using TC

Nevertheless, due to the complexity of building $\mathcal{T}^{-1}$, this approach had been considered infeasible for complex systems. By employing saturation, instead, our algorithm to compute $\mathcal{T}^{-1}$ completes on some large models, such as the dining philosopher problem with 1,000 philosophers, while, for models containing many SCCs, it may succeed where other algorithms have no hope. Thus, while the TC approach is not as robust as Lockstep, it can be used as an alternative to it when Lockstep cannot enumerate all SCCs.

$\mathcal{T}^{-1}$ can also be used to find recurrent states, i.e., states in terminal SCCs. State $\mathbf{j}$ is in a terminal SCC if, for any state $\mathbf{i}$, $\mathbf{j} \to \mathbf{i} \Rightarrow \mathbf{i} \to \mathbf{j}$. Given two states $\mathbf{i}, \mathbf{j}$, let $\mathbf{j} \mapsto \mathbf{i}$ mean $\mathbf{j} \to \mathbf{i}$ and $\mathbf{i} \not\to \mathbf{j}$. We can encode this relation with a $2L$-level MDD, obtained as $\mathcal{T}^{-1} \setminus \mathcal{T}$. The pseudocode for this algorithm is shown as *TSCC_TC* in Fig. 6. The set $\{(\mathbf{i}, \mathbf{j}) \mid \mathbf{j} \mapsto \mathbf{i}\}$ is encoded with a $2L$-level MDD r^*. Then the set of states $\{\mathbf{j} \mid \exists \mathbf{i}, \mathbf{j} \mapsto \mathbf{i}\}$, which do *not* belong to terminal SCCs, is computed by existentially quantifying out the unprimed levels (Line 5) and stored in MDD *nontscc*. All other states are the recurrent states belonging to terminal SCCs.

To the best of our knowledge, this is the first TC-based algorithm for terminal SCC computation. This algorithm is more costly in runtime and memory than SCC computation because of the need to obtain the $\mapsto$ relation. However, by employing *TransClosSat*, it works for most of the models we considered. Moreover, for models with a huge number of terminal SCCs, this algorithm is, again, the only feasible approach.

6 Fair cycles

One application of the SCC computation is to check the language emptiness of an ω-automaton. The language of an ω-automaton is nonempty if there is a nontrivial cycle which satisfies a certain fairness condition, i.e., a *fair cycle*. Thus, it is necessary to extend the SCC computation to finding fair cycles. Two widely used fairness conditions are Büchi fairness (weak fairness) and Streett fairness (strong fairness) [12].

Strong fairness is specified with a set of pairs of sets $\{(\mathcal{R}_1, \mathcal{C}_1), \ldots, (\mathcal{R}_n, \mathcal{C}_n)\}$, and a cycle satisfies it iff, for each $(\mathcal{R}_i, \mathcal{C}_i)$, either no state of $\mathcal{R}_i$ is in the cycle or some state of $\mathcal{C}_i$ is in the cycle. *Weak fairness* is specified with a set of sets of states $\{\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n\}$. Its semantics is expressed by the special case of strong fairness where, in each pair, $\mathcal{R}_i = \mathcal{S}_{\text{rch}}$ and $\mathcal{C}_i = \mathcal{F}_i$; thus, we focus on strong fairness.

Lockstep is also able to find strongly fair cycles [2], but might suffer from SCC *refinement* [2,Figure 4]. This occurs when an SCC intersects a $\mathcal{R}_i$ but not $\mathcal{C}_i$. Then we need to filter out all states in $\mathcal{R}_i$ and run Lockstep on the remaining states in this SCC. This process may enumerate all states in the worst case, as with SCC enumeration. Here, we present a TC approach that avoids enumeration. To the best of our knowledge, this is the first TC approach for fair cycle detection. The following defines the *constrained TC*.

Definition 2 Given a set of states $\mathcal{X}$, the constrained backward TC is $\mathcal{T}_{\mathcal{X}}^{-1} = \mathit{TransClosSat}(\mathcal{X}, \mathcal{N}^{-1})$.

$\mathcal{T}_{\mathcal{X}}^{-1}$ specifies the relation between two states $\mathbf{i}$ and $\mathbf{j}$, such that $(\mathbf{i}, \mathbf{j}) \in \mathcal{T}_{\mathcal{X}}^{-1} \Leftrightarrow \mathbf{j} \to \mathbf{t}_1 \to \mathbf{t}_2 \to \cdots \to \mathbf{i}$ and $\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{i} \in \mathcal{X}$. Thus, the set of states belonging to some (strongly) fair cycle is given by:

$$\bigcap_{m=1,\ldots,n} \left(\mathit{TCtoSCC}\left(\mathcal{T}^{-1}_{\mathcal{S}_{\text{rch}} \setminus \mathcal{R}_m}\right) \cup \left\{ \mathbf{i} \mid \exists \mathbf{c}_m \in \mathcal{C}_m, \left(\mathcal{T}(\mathbf{c}_m, \mathbf{i}) \cap \mathcal{T}^{-1}(\mathbf{c}_m, \mathbf{i}) \right) \right\} \right),$$

If $(\mathbf{i},\mathbf{i}) \in \mathcal{T}^{-1}_{\mathcal{S}_{\text{rch}} \setminus \mathcal{R}_m}$, there is a nontrivial path from $\mathbf{i}$ to $\mathbf{i}$ that avoids $\mathcal{R}_m$. Thus, $\mathit{TCtoSCC}(\mathcal{T}^{-1}_{\mathcal{S}_{\text{rch}} \setminus \mathcal{R}_m})$ returns the states in cycles that contain no state in $\mathcal{R}_m$ and satisfy strong fairness. If $(\mathbf{i},\mathbf{c}_m) \in (\mathcal{T}(\mathbf{c}_m,\mathbf{i}) \cap \mathcal{T}^{-1}(\mathbf{c}_m,\mathbf{i}))$, $\mathbf{i}$ belongs to a cycle which contains states in $\mathbf{c}_m$. The subformula in the second line corresponds to the states in a cycle containing at least one state in $\mathcal{C}_m$, satisfying strong fairness.

The pseudocode in Fig. 7 computes the above formula. For each $(\mathcal{R}_m, \mathcal{C}_m)$, c encodes the states in cycles that do not intersect $\mathcal{R}_m$, d encodes the states in cycles that intersect $\mathcal{C}_m$, and $\mathit{Cross}(\mathcal{C}_m, \mathcal{S}_{\text{rch}})$ returns a $2L$-level MDD encoding the cross-product of $\mathcal{C}_m$ and $\mathcal{S}_{\text{rch}}$. The cost mainly lies in computing the intersection of $\mathcal{T}^{-1}$ and $\mathcal{T}$, which is similar to

```
mdd FairSCC TC(S_rch, N^-1, {(R_1, C_1), ..., (R_n, C_n)})
1 mdd T^-1 ← TransClosSat(S_rch, N^-1);
2 mdd T ← Inverse(T^-1);
3 mdd s ← S_rch;
4 foreach m = 1, ··· , n do
5   mdd c ← TCtoSCC(TransClosSat(Diff(S_rch, R_m), N^-1));
6   mdd d ← Intersect(T^-1, T);
7   d ← QuantUnpr(Intersect(d, Cross(C_m, S_rch)));
    • Cross(C_m, S_rch) encodes {(c_m, i) : c_m ∈ R_m ∧ i ∈ S_rch}
8   s ← Intersect(s, Union(c, d));
9 endfor;
10 return s;
```

Fig. 7 Computing fair cycles using TC

computing the relation $\mapsto$. Moreover, the complexity grows linearly with the size n of the fairness condition.

7 Experimental results

We implemented our algorithms in SMART [7] and report experimental results running on an Intel Xeon 3.0 Ghz workstation with 3 GB RAM under SuSE Linux 9.1. The models, described as Petri nets and translated into the input language of SMART, include the closed queue network (*cqn*) of [29], two implementations of arbiters ($arbiter_1$, $arbiter_2$) [11], one which guarantees fairness and the other which does not, the queen placement problem (*queens*), the dining philosopher problem (*phil*), and the leader selection protocol (*leader*) [6]. The size for each model is controlled by a parameter N. The number of SCCs (terminal SCCs) and states in SCCs (terminal SCCs) for each model is listed in column "SCC count" ("TSCC count") and column "SCC states" ("TSCC states"), respectively. We set an upper bound of 2 h for runtime and 1 GB for the unique table (used to canonically store the MDD nodes). The main metrics of our comparison are runtime and peak memory consumption (for unique table plus operation cache, required for efficient dynamic programming implementation of MDD operation), which are measured in seconds and megabytes, respectively.

The top part of Table 1 compares three algorithms for SCC computation: the TC algorithm (column "TC") of Sect. 5, the improved XB algorithm (column "XBSAT") of Sect. 4, and Lockstep (column "Lockstep"). We can see that the improved XB algorithm using saturation is better than Lockstep for most models, in both runtime and memory. Compared with the SCC enumeration algorithms, the TC algorithm is often more expensive but, for *queens* and $arbiter_2$, it completes within the time limit while the other two algorithms fail. For $arbiter_2$, our TC algorithm explores over 10^{150} SCCs in a few seconds, while it is obviously not feasible to exhaustively enumerate all SCCs in reasonable time. To the best of our knowledge, this is the best result of SCC computation reported, confirming the main advantage of the TC algorithm: its insensitivity to the number of SCCs. With the help of our new algorithm, the TC can be built for some large systems, such as the dining philosopher problem with 1,000 philosophers.

The bottom part of Table 1 compares three algorithms for terminal SCC computation: *XBSat* (column "XBSat") presented in Sect. 4, *TSCC_TC* (column "TC") presented in Sect. 5, and the BFS XB algorithm *XB_TSCC* (column "XBBFS") shown in Fig. 2. The basic trends are similar to those for SCC computation: algorithm *XBSat* works consistently better than the traditional method, while *TSCC_TC* is less efficient for most models. In the framework of the XB algorithm, computing terminal SCCs is faster than computing SCC because a larger set of states is pruned at each recursion. On the contrary, *TSCC_TC* is more expensive than *SCC_TC* due to the computation of the $\mapsto$ relation. As a consequence,

Table 1 Results for SCC and terminal SCC computation

Model		SCC count	SCC states	TC		XBSat		Lockstep	
Name	N			Mem	Time	Mem	Time	Mem	Time
SCC computation									
	10	11	2.09e+10	34.2	13.6	3.4	<0.1	4.0	3.9
cqn	15	16	2.20e+15	64.4	73.8	5.0	0.2	89.1	44.5
	20	21	2.32e+20	72.7	687.8	25.8	0.5	118.7	275.0
	100	1	4.96e+62	5.0	0.5	3.2	<0.1	52.0	4.5
phil	500	1	3.03e+316	33.0	4.0	24.5	0.1	–	to
	1,000	1	9.18e+626	40.5	7.8	29.1	0.3	–	to
	10	3.22e+4	3.23e+4	8.2	1.6	64.4	14.5	63.9	12.4
queens	11	1.53e+5	1.53e+5	45.8	9.0	94.2	108.6	96.3	93.6
	12	7.95e+5	7.95e+5	184.8	60.6	170.2	1220.4	281.9	1663.9
	13	4.37e+6	4.37e+6	916.5	840.6	–	to	–	to
	3	4	6.78e+2	6.0	1.4	20.8	<0.1	20.8	<0.1
leader	4	11	9.50e+3	70.3	73.1	25.4	1.1	23.8	0.3
	5	26	1.25e+5	116.6	3830.4	35.6	40.8	49.4	6.4
	6	57	1.54e+6	–	to	41.6	1494.9	417.2	387.9
	10	1	2.05e+4	24.1	1.2	21.4	<0.1	21.8	0.1
$arbiter_1$	15	1	9.83e+5	128.3	63.0	45.1	<0.1	62.1	6.8
	20	1	4.19e+7	mo	–	709.7	<0.1	mo	–
	10	1024	1.02e+4	20.3	<0.1	26.2	0.7	31.1	1.1
$arbiter_2$	15	32768	4.91e+5	20.4	<0.1	31.1	51.8	211.3	990.3
	20	1.05e+6	2.10e+7	20.4	<0.1	31.2	2393.3	–	to
	500	**3.27e+150**	(**1.64e+151**)*	**41.0**	**4.0**	–	to	–	to

Table 1 continued

Model		TSCC count	TSCC states	TC		XBSat		XBBFS	
Name	N			Mem	Time	Mem	Time	Mem	Time
Terminal SCC computation									
	10	10	2.09e+10	37.9	15.5	21.4	<0.1	33.5	3.4
cqn	15	15	2.18e+15	64.8	79.6	23.0	0.3	59.4	33.7
	20	20	2.31e+20	72.7	691.3	26.2	0.8	90.0	280.5
	100	2	2	26.5	0.5	20.9	<0.1	39.2	8.7
phil	500	2	2	34.3	4.1	23.2	<0.1	–	to
	1,000	2	2	44.4	11.3	26.5	0.2	–	to
	10	1.28e+4	1.28e+4	36.2	3.0	46.7	2.8	62.3	35.1
queens	11	6.11e+4	6.11e+4	76.5	19.3	70.6	24.5	145.2	364.2
	12	3.14e+5	3.14e+5	244.1	205.4	98.8	179.4	mo	–
	13	1.72e+6	1.72e+6	mo	–	269.0	1940.81	mo	–
	3	3	3	26.6	1.5	20.7	<0.1	21.4	0.1
leader	4	4	4	70.6	75.1	24.4	0.9	38.0	4.5
	5	5	5	119.3	3845.3	30.6	26.9	41.1	87.6
	6	6	6	–	to	39.0	492.9	44.8	1341.5
	10	1	2.05e+4	24.1	1.2	20.4	<0.1	22.4	0.4
$arbiter_1$	15	1	9.83e+5	128.3	63.1	20.4	<0.1	65.3	23.3
	20	1	4.19e+7	mo	–	20.5	<0.1	–	to
	10	1	1	20.4	<0.1	20.9	<0.1	39.6	6.4
$arbiter_2$	15	1	1	20.5	<0.1	40.6	4.6	–	to
	20	1	1	20.5	<0.1	450.0	2897.8	–	to

*A value computed from the model structure, not using XB

TSCC_TC suffers even more from large memory and runtime requirement. Nevertheless, for models with large numbers of terminal SCCs, such as *queens*, *TSCC_TC* outperforms the BFS XB algorithm.

Some conclusions can be drawn from the above results and discussion. First, saturation is effective in speeding up SCC and terminal SCC computation within the framework of the XB algorithm. Second, our new saturation algorithm makes TC computation feasible for some complex models containing up to 10^{150} states. Third, SCC computation based on TC is superior to SCC enumeration algorithms, which find SCCs one by one, for models with huge numbers of SCCs.

Although the TC approach is not as robust as Lockstep, especially when the number of SCCs in the model is manageable, we argue that it should be considered as a reasonable alternative worth of further research. Given a new model with an unknown number of existing SCCs, employing both of these approaches at the same time will be ideal. Current trend of multi-core computers provide a possible means of parallelizing these two algorithms. Some of the common data structures, like MDDs encoding the state space and next-state functions, could then even be shared.

8 Conclusion and future work

We focused on improving two previous approaches for SCC computation, the Xie-Beerel (XB) and the transitive closure (TC) algorithms, using saturation. For the asynchronous models we study, the improved XB algorithm using saturation achieves a clear speedup, and our new algorithm to compute TC using saturation is experimentally shown to be capable of handling models with up to 10^{150} of SCCs. We argue that the TC-based approach is worth further research because of its ability to deal with models having huge numbers of SCCs.

The SCC analysis can be an important step to simplify Markov chain analysis and stochastic model checking, as shown in [1,10]. Our future work is to apply the algorithms in this paper to these topics, where symbolic methods are only scarcely used.

Acknowledgments This work was supported in part by the National Science Foundation under Grant CCF-1018057 and by a UC MEXUS-CONACYT Collaborative Research Grant.

References

1. Ábrahám E, Jansen N, Wimmer R, Katoen J-P, Becker B (2010) DTMC model checking by SCC reduction. In: QEST, pp 37–46
2. Bloem R, Gabow HN, Somenzi F (2000) An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In: Formal methods in computer aided design. Springer, Berlin, pp 37–54
3. Bollig B, Wegener I (1996) Improving the variable ordering of OBDDs is NP-complete. IEEE Trans Comput 45(9):993–1002
4. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691

5. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98: 142–170
6. Ciardo G et al (2003) SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. http://www.cs.ucr.edu/~ciardo/SMART/.
7. Ciardo G, Jones RL, Miner AS, Siminiceanu R (2006) Logical and stochastic modeling with SMART. Perf Eval 63: 578–608
8. Ciardo G, Marmorstein R, Siminiceanu R (2006) The saturation algorithm for symbolic state space exploration. Softw Tools Technol Transf 8(1): 4–25
9. Ciardo G, Siminiceanu R (2003) Structural symbolic CTL model checking of asynchronous systems. In: Hunt WA Jr, Somenzi F (eds) Proceedings of CAV, Boulder, CO. LNCS, vol 2725. Springer, Berlin, pp 40–53
10. Ciesinski F, Baier C, Grösser M, Klein J (2008) Reduction techniques for model checking Markov decision processes. In: Proceedings of the 2008 5th international conference on quantitative evaluation of systems. IEEE Computer Society, Washington, DC, pp 45–54
11. Cimatti A, Clarke E, Giunchiglia F, Roveri M (1999) NuSMV: a new symbolic model checker. http://nusmv.irst.itc.it/
12. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge, MA
13. Emerson EA, Lei C-L (1986) Efficient model checking in fragments of the propositional mu-Calculus. In: Proceedings, symposium on logic in computer science, 16–18 Jun 1986. IEEE Computer Society, Cambridge, MA, pp 267–278
14. Fisler K, Fraer R, Kamhi G, Vardi MY, Yang Z (2001) Is there a best symbolic cycle-detection algorithm? In: Proceedings of the 7th international conference on tools and algorithms for the construction and analysis of systems, TACAS 2001, London, UK. Springer, Berlin, pp 420–434
15. Hachtel G, Macii E, Pardo A, Somenzi F (1996) Markovian analysis of large finite state machines. IEEE Trans CAD Integr Circ Syst 15(12): 1479–1493
16. Hardin RH, Kurshan RP, Shukla SK, Vardi MY (2001) A new heuristic for bad cycle detection using bdds. Formal Methods Syst Des 18(2): 131–140
17. Hojati R, Touati HJ, Kurshan RP, Brayton RK (1992) Efficient ω-regular language containment. In: CAV, pp 396–409
18. Kam T, Villa T, Brayton RK, Sangiovanni-Vincentelli A (1998) Multi-valued decision diagrams: theory and applications. Multiple Valued Logic 4(1–2): 9–62
19. Kesten Y, Pnueli A, Raviv L-o (1998) Algorithmic verification of linear temporal logic specifications. In: ICALP '98: Proceedings of the 25th international colloquium on automata, languages and programming, London, UK. Springer, Berlin, pp 1–16
20. Matsunaga Y, McGeer PC, Brayton RK (1993) On computing the transitive closure of a state transition relation. In: Proceedings of the 30th international design automation conference, pp 260–265
21. Murata T (1989) Petri nets: properties, analysis and applications. Proc IEEE 77(4): 541–579
22. Ravi K, Bloem R, Somenzi F (2000) A comparative study of symbolic algorithms for the computation of fair cycles. In: FMCAD '00: Proceedings of the 3rd international conference on formal methods in computer-aided design, London, UK. Springer, Berlin, pp 143–160
23. Safra S, Vardi MY (1989) On omega-automata and temporal logic. In: Proceedings of the 21st annual ACM symposium on theory of computing. ACM, Seattle, pp 127–137
24. Somenzi F, Ravi K, Bloem R (2002) Analysis of symbolic SCC hull algorithms. In: FMCAD '02: Proceedings of the 4th international conference on formal methods in computer-aided design, London, UK. Springer, Berlin, pp 88–105
25. Stewart WJ (1994) Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton
26. Tarjan R (1971) Depth-first search and linear graph algorithms. In: Proceedings of the 12th annual symposium on switching and automata Theory (swat 1971). IEEE Computer Society, Washington, DC, pp 114–121
27. Wan M, Ciardo G (2009) Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In: Nielsen M et al (eds) Proceedings of 35th international conference on current trends in theory and practice of computer science (SOFSEM), Špindlerův Mlýn, Czech Republic. LNCS, vol 5404. Springer, Berlin, pp 582–594
28. Xie A, Beerel PA (1998) Efficient state classification of finite-state Markov chains. IEEE Trans CAD Integr Circuit Syst 17(12): 1334–1339
29. Xie A, Beerel PA (2000) Implicit enumeration of strongly connected components and an application to formal verification. IEEE Trans CAD Integr Circuit Syst 19(10): 1225–1230
30. Zhao Y, Ciardo G (2009) Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Liu Z, Ravn AP (eds) Proceedings of 7th international symposium on automated technology for verification and analysis (ATVA), Macao SAR, China. LNCS, vol 5799. Springer, Berlin, pp 368–381