Symbolic Computation of Strongly Connected Components Using Saturation *

Yang Zhao Gianfranco Ciardo
Department of Computer Science and Engineering, University of California, Riverside
zhaoy@cs.ucr.edu ciardo@cs.ucr.edu

Abstract

Finding strongly connected components (SCCs) in the state-space of discrete-state models is a critical task in formal verification of LTL and fair CTL properties, but the potentially huge number of reachable states and SCCs constitutes a formidable challenge. This paper is concerned with computing the sets of states in SCCs or terminal SCCs of asynchronous systems. Because of its advantages in many applications, we employ *saturation* on two previously proposed approaches: the Xie-Beerel algorithm and transitive closure. First, saturation speeds up state-space exploration when computing each SCC in the Xie-Beerel algorithm. Then, our main contribution is a novel algorithm to compute the transitive closure using saturation. Experimental results indicate that our improved algorithms achieve a clear speedup over previous algorithms in some cases. With the help of the new transitive closure computation algorithm, up to 10^{150} SCCs can be explored within a few seconds.

1 Introduction

Finding strongly connected components (SCCs) is a basic problem in graph theory. For discrete-state models, some interesting properties, such as LTL [8] and fair CTL, are related with the existence of SCCs in the state transition graph, and this is also the central problem in the language emptiness check for ω-automata. For large discrete-state models (e.g., 10^{20} states), it is impractical to find SCCs using traditional depth-first search, motivating the study of symbolic computation of SCCs. In this paper, the objective is to build the set of states in non-trivial SCCs.

The structure of SCCs in a graph can be captured by its *SCC quotient graph*, obtained by collapsing each SCC into a single node. This resulting graph is acyclic, and thus defines a partial order on the SCCs. *Terminal SCCs* are leaf nodes in the SCC quotient graph. In the context of large scale Markov chain analysis, an interesting problem is to partition the state space into *recurrent* states, which belong to terminal SCCs, and *transient* states, which are not recurrent.

The main difficulties in SCC computation are: having to explore huge state spaces and, potentially, having to deal with a large number of (terminal) SCCs. The first problem is the primary obstacle to formal verification due to the obvious limitation of computational resources. Traditional BDD-based approaches employ *image* and *preimage* computations on state-space exploration and, while quite successful in fully synchronous systems, they do not work as well for asynchronous systems. The second problem constitutes a bottleneck for one class of previous work, which enumerates SCCs one by one. Section 2.3 discusses this problem in more detail.

This paper addresses the computation of states in SCCs and terminal SCCs. We propose two approaches based on two previous ideas: the *Xie-Beerel algorithm* and *transitive closure*. Saturation, which schedules the firing of events according to their locality, is employed to overcome the complexity of state-space exploration. Pointing to the second difficulty, our efforts are devoted to an algorithm based on the transitive closure, which does not suffer from a huge numbers of SCCs but, as previously proposed, often requires large amounts of runtime and memory. We then propose to use a saturation-based algorithm to compute the transitive closure, enabling it to be a practical method of SCC computation for complex systems. We also present an algorithm for computing recurrent states based on the transitive closure.

*Work supported in part by the National Science Foundation under grant CCF-0848463.

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on data structure we use and the saturation algorithm. Section 3 introduces an improved Xie-Beerel algorithm using saturation. Section 4 introduces our transitive closure computation algorithm using saturation and the corresponding algorithms for SCC and terminal SCC computations. Section 6 compares the performance of our algorithms and that of Lockstep.

2 Preliminaries

Consider a discrete-state model $(\mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N})$ where the potential state space $\mathcal{S}$ is given by the product $\mathcal{S}_L \times \cdots \times \mathcal{S}_1$ of the local state spaces of L submodels, thus 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 $\mathcal{S}_{init} \subseteq \mathcal{S}$; 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 reached in one step when 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 inverse next-state functions, i.e., $\mathcal{N}_\alpha^{-1}(\mathbf{i})$ is the set of states that can reach $\mathbf{i}$ in one step by firing event α.

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

2.1 Symbolic encoding of discrete-state systems

We employ *multi-way decision diagrams* (MDDs) [7] to encode discrete-state systems. MDDs extend binary decision diagrams (BDDs) by allowing integer-valued variables, thus are suitable for discrete-state models with bounded but non-boolean valued state variables, such as Petri nets [10]. There are two possible terminal nodes, **0** and **1**, for all MDDs. Each MDD has a single root node.

We encode a set of states 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 it is a root node. If a is nonterminal and $a.lvl = k$, then a has n_k outgoing edges labeled with $\{0, ..., n_k - 1\}$, each of which corresponds to a local state in $\mathcal{S}_k$. The node pointed by the edge labeled with i_k is denoted with $a[i_k]$. If $a[i_k] \neq \mathbf{0}$, it must be a node at level $k-1$. Finally, let $\mathcal{B}(a) \subseteq \mathcal{S}_k \times \cdots \times \mathcal{S}_1$ be the set of paths from node a to **1**.

Turning to the encoding of the next-state functions, most asynchronous systems enjoy *locality*, which can be exploited to obtain a compact symbolic expression. An event α is *independent* of the k^{th} submodel if its enabling does not depend on i_k and its firing does not change the value of i_k. A level k belongs to the *support* set of event α, denoted $supp(\alpha)$, if α is not independent of k. We define $Top(\alpha)$ to be the highest-numbered level in $supp(\alpha)$, and $\mathcal{E}_k$ to be the set of events $\{\alpha \in \mathcal{E} : Top(\alpha) = k\}$. Also, we let $\mathcal{N}_k$ be the next-state function corresponding to all events in $\mathcal{E}_k$, i.e., $\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$.

We encode the next-state function using $2L$-level MDDs with level order $L, L', ..., 1, 1'$, where unprimed and primed levels correspond to "from" and "to" states, respectively, and we let $Unprimed(k) = Unprimed(k') = k$. We use the *quasi-identity-fully (QIF)* reduction rule [13] for MDDs encoding next-state functions. For an event α with $Top(\alpha) = k$, $\mathcal{N}_\alpha$ is encoded with a $2k$-level MDD since it does not affect state variables corresponding to nodes on levels $L, \ldots, k+1$; these levels are skipped in this MDD. The advantage of the QIF reduction rule is that the application of $\mathcal{N}_\alpha$ only needs to start at level $Top(\alpha)$, and not at level L. We refer interested readers to [13] for more details about this encoding.

2.2 State-space generation using saturation

All symbolic approaches to state-space generation use some variant of symbolic image computation. The simplest approach is the breadth-first iteration, directly implementing the definition of the state space

```
mdd Saturate({N_L,...,N_1}, mdd s)
 1  if InCache_Saturate(s,t) then return t;
 2  level k ← s.lvl;
 3  mdd t ← NewNode(k);   mdd r ← N_k;
 4  foreach i ∈ S_k s.t. s[i] ≠ 0 do
      • First saturate all its children
 5    t[i] ← Saturate({N_L,...,N_1}, s[i]);
 6  endfor
 7  repeat      • Get a local fixed point on the root
 8    foreach i,i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 9      mdd u ←
        RelProdSat({N_L,...,N_1}, t[i], r[i][i']);
10      t[i'] ← Or(t[i'], u);
11    endfor
12  until t does not change;
13  t ← UniqueTablePut(t);
14  CacheAdd_Saturate(s,t);
15  return t;
```

```
mdd RelProdSat({N_L,...,N_1}, mdd s, mdd r)
 1  if s = 1 and r = 1 then return 1; endif
 2  if InCache_ConsRelProd(s,r,t) then return t; endif
 3  level k ← s.lvl;   mdd t ← 0;
 4  foreach i,i' ∈ S_k s.t. r[i][i'] ≠ 0 do
 5    mdd u ← RelProdSat({N_L,...,N_1}, s[i], r[i][i']);
 6    if u ≠ 0 then
 7      if t = 0 then t ← NewNode(k); endif
 8      t[i'] ← Or(t[i'], u);
 9    endif
10  endfor
11  t ← Saturate({N_L,...,N_1}, UniqueTablePut(t));
      • Return a saturated MDD
12  CacheAdd_RelProdSat(s,r,t);
13  return t;
      • UniqueTable guarantees the uniqueness of each node
      • Cache reduces complexity through dynamic programming
```

Figure 1: Saturation algorithms.

$\mathcal{S}_{rch}$ as the fixed point of $\mathcal{S}_{init} \cup \mathcal{N}(\mathcal{S}_{init}) \cup \mathcal{N}^2(\mathcal{S}_{init}) \cup \mathcal{N}^3(\mathcal{S}_{init}) \cup \cdots$. Given a set of states $\mathcal{X}$, their forward and backward reachable sets are $forward(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}(\mathcal{X}) \cup \mathcal{N}^2(\mathcal{X}) \cup \mathcal{N}^3(\mathcal{X}) \cup \cdots$ and $backward(\mathcal{X}) = \mathcal{X} \cup \mathcal{N}^{-1}(\mathcal{X}) \cup (\mathcal{N}^{-1})^2(\mathcal{X}) \cup (\mathcal{N}^{-1})^3(\mathcal{X}) \cup \cdots$.

Locality and disjunctive partition of the next-state function form the basis of the saturation algorithm. The key idea is to apply the event firings in an order consistent with their *Top*. An event in $\mathcal{E}_k$ will not be fired until the events in $\mathcal{E}_h$ where $h < k$ do not further grow the explored state space. We say that a node a at level k is *saturated* if it is a fixed point with respect to firing any event that is independent of all levels above k: $\forall h, k \geq h \geq 1, \forall \alpha \in \mathcal{E}_h, \forall \mathbf{i} \in \mathcal{S}_L \times \ldots \times \mathcal{S}_{k+1}$, $\{\mathbf{i}\} \times \mathcal{B}(a) \supseteq \mathcal{N}_\alpha(\{\mathbf{i}\} \times \mathcal{B}(a))$

Figure 1 shows the pseudocode of the saturation algorithm. In function *Saturate*, the nodes in MDD s are saturated in order, from the bottom level to the top level. Different from the traditional relational product operation, *RelProdSat* always returns a saturated MDD. Saturation can also be applied to computing $backward(\mathcal{X})$ by using inverse next-state functions $\{\mathcal{N}_L^{-1}, \ldots, \mathcal{N}_1^{-1}\}$.

2.3 Previous work

Symbolic SCC analysis has been widely explored. Almost all of these algorithms employ BDD-based manipulation of sets of states. Many efforts have been made on computing the *SCC hull*. The SCC hull contains not only states in nontrivial SCCs, but also states on the paths between them. A family of SCC hull algorithms [12] with the same upper bound of complexity is available. We review two categories of previous work on the same problem as ours: *transitive closure* and the *Xie-Beerel algorithm*.

Hojati et al. [6] presented a symbolic algorithm for testing ω-regular language containment by computing the transitive closure, namely, $\mathcal{N}^+ = \mathcal{N} \cup \mathcal{N}^2 \cup \mathcal{N}^3 \cup \cdots$. Matsunaga et al. [9] proposed a recursive procedure for computing the transitive closure. While it is a fully symbolic algorithm, due to the unacceptable complexity of computing the transitive closure, this approach has long been considered infeasible for complex systems.

Xie et al. [15] proposed an algorithm, referred as the Xie-Beerel algorithm in this paper, combining both explicit state enumeration and symbolic state-space exploration. This algorithm explicitly picks a state as a "seed", computes the forward and backward reachable states from the seed and finds the

```
mdd Lockstep(mdd P)
 1 if(P = ∅) then return ∅;
 2 mdd ans ← ∅;   mdd seed ← Pick(P);   mdd C ← 0;
 3 mdd F_front ← N(seed) ∩ P;   mdd B_front ← N^-1(seed) ∩ P;
 4 mdd F ← F_front;   mdd B ← B_front;
 5 while(F_front ≠ ∅ and B_front ≠ ∅)
 6    F_front ← N(F_front) ∩ P \ F; B_front ← N(B_front) ∩ P \ B;
 7    F ← F ∪ F_front;   B ← B ∪ B_front;
 8 endwhile
 9 if(F_front = ∅) then
10    mdd conv ← F;
11    while(B_front ∩ F ≠ ∅) do B_front ← N(B_front) ∩ P \ B;  B ← B ∪ B_front; endwhile
12 else
13    mdd conv ← B;
14    while(F_front ∩ B ≠ ∅) do F_front ← N(F_front) ∩ P \ F;  F ← F ∪ F_front; endwhile
15 endif
16 if(F ∩ B ≠ ∅ ) then C ← (F ∩ B) ∪ seed;   ans ← C; endif
17 ans ← ans ∪ Lockstep(conv \ C) ∪ Lockstep(P \ (conv ∪ seed));
18 return ans;
```

```
mdd XB_TSCC(mdd S)
 1 mdd ans ← ∅;   mdd P ← S;   mdd seed, F, B;
 2 while (P ≠ ∅)
 3    seed ← Pick(P);   F ← forward(seed) ∩ P;   B ← backward(seed) ∩ P;
 4    if F \ B = ∅ then ans ← ans ∪ F; endif                    • Find a terminal SCC
 5    P ← P \ B;
 6 endwhile
 7 return ans;
```

Figure 2: Lockstep for SCC computation and Xie-Beerel's algorithm for terminal SCC computation.

SCC containing this seed as the intersection of these two sets of states. Bloem et al. [2] presented a improved algorithm called *Lockstep*, shown in Figure 2. $Lockstep(\mathcal{S}_{rch})$ returns the set of states belonging to non-trivial SCCs. It has been proven that Lockstep requires in $O(n \log n)$ image and preimage computations (Theorem 2 in [2]), where n is the number of reachable states. As shown in Figure 2, given a "seed" state, instead of computing sets of forward and backward reachable states separately, it uses the set which converges earlier to bound the other. This optimization constitutes the key point in achieving $O(n \log n)$ complexity. Ravi et al. [11] compared the SCC-hull algorithms and Lockstep. According to our experimental results, Lockstep often works very well for systems with few SCCs. However, as the number of SCCs grows, the exhaustive enumeration of SCCs becomes a problem. In this paper, we compare our algorithms to Lockstep.

Xie et al. [14] proposed a similar idea in computing recurrent states in large scale Markov chains. The pseudocode of that algorithm is shown as *XB_TSCC* in Figure 2. From a randomly picked seed state, if the forward reachable states (F) is a subset of backward reachable states (B), F is a terminal SCC; otherwise ($F \not\subseteq B$), no terminal SCC exists in B, and B can be eliminated from future exploration.

The main ideas of our two approaches belong to these two categories of previous work. In the Xie-Beerel algorithm, BFS-based state-space exploration can be replaced with saturation. For transitive closure computation, we propose a new algorithm using saturation.

```
mdd XBSaturation(mdd P)
 1  if(P = ∅) then return ∅;
 2  mdd ans ← ∅; mdd seed ← Pick(P);
 3  mdd F_front ← N(seed) ∩ P;  mdd B_front ← N^-1(seed) ∩ P;
 4  mdd F ← Saturate({N_L ··· N_1}, F_front) ∩ P;
 5  mdd B ← Saturate({N_L^-1 ··· N_1^-1}, B_front) ∩ P;
 6  mdd C ← F ∩ B;    if C ≠ ∅ then ans ← C; endif     • Line 6−8 are for computing SCCs
 7  ans ← ans ∪ XBSaturation(F \ C) ∪ XBSaturation(P \ F);
 8  return ans;
 6' if F \ B = ∅ then ans ← ans ∪ F; endif     • Line 6'−8' are for computing terminal SCCs
 7' ans ← ans ∪ XBSaturation(P \ B);
 8' return ans;
```

Figure 3: Improved Xie-Beerel algorithm using saturation.

3 Improving the Xie-Beerel algorithm using saturation

A straightforward idea is to employ saturation on the state-space exploration in the Xie-Beerel algorithm. The pseudocode of our algorithms for computing SCCs and terminal SCCs is shown as *XBSaturation* in Figure 3. The merit of our algorithms comes from the higher efficiency of saturation in computing forward and backward reachable states (B and F). However, our algorithms need to compute B and F separately, while Lockstep can use the set that converges first to bound the other, which may reduce the number of image computations (steps). Thus, there is a trade-off between the advantages of BFS and saturation. From a theoretical point of view, the complexity of our algorithm can hardly be compared directly with the result in [2], which measures the complexity by the number of steps. Since saturation executes a series of light-weight events firing instead of global image computations, its complexity cannot be captured as a number of steps. Furthermore, saturation results in more compact decision diagrams during state-space exploration, often greatly reducing runtime and memory. Performance is also affected by which seed is picked in each iteration. For a fair comparison, we pick the same seed in both algorithms at each iteration. The experimental results in Section 6 show that, for most models, the improved Xie-Beerel algorithm using saturation outperforms Lockstep, sometimes by orders of magnitude.

4 Applying saturation to computing transitive closure

We define the backward transitive closure (TC^{-1}) of a discrete-state model as follows:

Definition 4.1. *A pair of states* $(\mathbf{i},\mathbf{j}) \in TC^{-1}$ *iff there exists a non-trivial (i.e., positive length) path* π *from* $\mathbf{j}$ *to* $\mathbf{i}$*, denoted by* $\mathbf{j} \to \mathbf{i}$*. Symmetrically, we can define TC where* $(\mathbf{i},\mathbf{j}) \in TC$ *iff* $\mathbf{i} \to \mathbf{j}$*.*

As TC and TC^{-1} are symmetric to each other, we focus on the computation of TC^{-1}. TC can then be obtained from TC^{-1} by simply swapping the unprimed and primed levels. Our algorithm is based on the following observation:

$$(\mathbf{i},\mathbf{j}) \in TC^{-1} \text{ iff } \exists \mathbf{k} \in \mathcal{N}^{-1}(\mathbf{i}) \text{ and } \mathbf{j} \in Saturate(\{\mathcal{N}_L^{-1}, \cdots, \mathcal{N}_1^{-1}\}, \{\mathbf{k}\})$$

Instead of executing saturation on $\mathbf{j}$ for each pair of $(\mathbf{i},\mathbf{j})$, we propose an algorithm that executes on the $2L$-level MDD encoding $\mathcal{N}^{-1}$. In function $SCC\_TC(\mathcal{N}^{-1})$ of Figure 4, TC^{-1} is computed in line 1 using function *TransClosureSat*, which runs bottom-up recursively. Similar to the idea of saturation shown in Figure 1, this function runs node-wise on primed level and fires lower level events exhaustively until the local fixed point is obtained. This procedure guarantees the following Lemma.

Lemma: Given a $2k$-level MDD n, *TransClosureSat*(n) returns an $2k$-level MDD t that for any $(\mathbf{i},\mathbf{j}) \in \mathscr{B}(n)$, all $(\mathbf{i},\mathbf{k})$ where $\mathbf{k} \in (\mathscr{N}_{\leq k}^{-1})^*(\mathbf{j})$ belong to $\mathscr{B}(t)$.

Theorem: *TransClosureSat*$(\mathscr{N}^{-1})$ returns TC^{-1}.

This theorem can be proved directly from Lemma and the definition of TC^{-1}. The pseudocode of the SCC computation using TC^{-1} is shown in *SCC_TC* in Figure 4. Then, function *TCtoSCC* extracts all states $\mathbf{i}$ such that $(\mathbf{i},\mathbf{i}) \in TC^{-1}$.

Unlike SCC enumeration algorithms like Xie-Beerel's or Lockstep, the TC-based approach does not necessarily suffer when the number of SCCs is large. Nevertheless, due to the complexity of building TC^{-1}, this approach is considered not feasible for complex systems. Thanks to the idea of saturation, our algorithm of computing TC^{-1} completes on some large models, such as the dining philosopher problem with 1000 philosophers. For some models containing large numbers of SCCs, the TC-based approach shows its advantages. While the TC-based approach is not as robust as Lockstep, it can be used as the substitute for Lockstep when Lockstep fails to exhaustively enumerate all SCCs.

TC^{-1} can also be employed to find recurrent states, i.e., terminal SCCs. As the other SCCs are not reachable from terminal SCCs, state $\mathbf{j}$ belongs to a terminal SCC iff $\forall \mathbf{i},\ \mathbf{j} \to \mathbf{i} \Longrightarrow \mathbf{i} \to \mathbf{j}$. Given states $\mathbf{i},\mathbf{j}$, let $\mathbf{j} \mapsto \mathbf{i}$ denote that $\mathbf{j} \to \mathbf{i}$ *and* $\neg(\mathbf{i} \to \mathbf{j})$. We can encode this relation with a $2L$-level MDD, which can be obtained as $TC^{-1} \setminus TC$. The pseudocode of this algorithm is shown as *TSCC_TC* in Figure 5. The set of $\{(\mathbf{i},\mathbf{j}) | \mathbf{j} \mapsto \mathbf{i}\}$ is encoded with a $2L$-level MDD L. Then, the set of states $\{\mathbf{j} | \exists \mathbf{i}, \mathbf{j} \mapsto \mathbf{i}\}$, which do *not* belong to terminal SCCs, is computed by quantifying out the unprimed levels and can be stored in MDD *nontscc*. The remaining states in SCCs are recurrent states belonging to terminal SCCs.

To the best of our knowledge, this is the first symbolic algorithm for terminal SCC computation using the transitive closure. This algorithm is more expensive in both runtime and memory than SCC computation because of the computation of the $\mapsto$ relation. With the help of *TransClosureSat*, this algorithm works for most of the models we study. Moreover, for models with many terminal SCCs, this algorithm also shows its unique benefits.

5 Fairness

One application of the SCC computation is to decide language emptiness for an ω-automaton. The language of an ω-automaton is nonempty if there is a nontrivial *fair loop* satisfying a certain fair constraint. Thus, it is necessary to extend the SCC computation to finding fair loops. Büchi fairness (weak fairness) [5] is a widely used fair condition specified as a set of sets of states $\{\mathscr{F}_1,\ldots,\mathscr{F}_n\}$. A fair loop satisfies Büchi fairness iff, for each $i = \{1,\ldots,n\}$, some state in $\mathscr{F}_i$ is included in the loop.

Lockstep is able to handle the computation of fair loops as proposed in [2]. Here we present a TC-based approach. Assume TC and TC^{-1} have been built, let $\mathscr{S}_{weak}$ be the set of states $\mathbf{i}$ satisfying:

$$\bigcap_{m=1,\ldots,n} [\exists \mathbf{f}_m \in \mathscr{F}_m.(TC(\mathbf{f}_m,\mathbf{i}) \wedge TC^{-1}(\mathbf{f}_m,\mathbf{i}))]$$

According to the definition of weak fairness, it can be proved that $\mathscr{S}_{weak}$ contains all states in the fair loops. The pseudocode of computing $\mathscr{S}_{weak}$ is shown in Figure 6. $\mathscr{F}_i \times \mathscr{S}_{rch}$ returns a $2L$-level MDD encoding all pairs of states $(\mathbf{i},\mathbf{j})$ where $\mathbf{i} \in \mathscr{F}_i$ and $\mathbf{j} \in \mathscr{S}_{rch}$. The main complexity lies in computing $TC(\mathbf{i},\mathbf{j}) \wedge TC^{-1}(\mathbf{i},\mathbf{j})$, which is similar to computing the $\mapsto$ relation in the terminal SCC computation.

6 Experimental results

We implement the proposed approaches in SMART [4] and report experimental results obtained on an Intel Xeon 3.0Ghz workstation with 3GB RAM under SuSE Linux 9.1. All the models are described as the Petri nets expressed in the input language of SMART. These models include a closed queue networks

```
mdd SCC_TC(N^-1)
 1 mdd TC^-1 ← TransClosureSat(N^-1);
 2 mdd SCC ← TCtoSCC(TC^-1);
 3 return SCC;
```

```
mdd TransClosureSat(mdd n)
 1 if InCache_TransClosureSat(n,t) then return t;
 2 level k ← n.lvl;  mdd t ← NewNode(k);  mdd r ← N^-1_Unprimed(k)
 3 foreach i,j ∈ S_k s.t. n[i][j] ≠ 0 do  t[i][j] ← TransClosureSat(n[i][j]); endfor
 4 foreach i ∈ S_Unprimed(k) s.t. n[i] ≠ 0
 5   repeat                                              • Build a local fixed point
 6     foreach j,j' ∈ S_Unprimed(k) s.t. n[i][j] ≠ 0 and r[j][j'] ≠ 0 do
 7       mdd u ← TCRelProdSat(t[i][j],r[j][j']);  t[i][j'] ← Or(t[i][j'],u);
 8     endfor
 9   until t does not change;
10 endfor
11 t ← UniqueTablePut(t);  CacheAdd_TransClosureSat(n,t);
12 return t;
```

```
mdd TCRelProdSat(mdd n,mdd r)
 1 if n = 1 and r = 1 then return 1;
 2 if InCache_TCRelProdSat(n,r,t) then return t;
 3 level k ← n.lvl;  mdd t ← 0;
 4 foreach i ∈ S_Unprimed(k) s.t. n[i] ≠ 0 do
 5   foreach j,j' ∈ S_Unprimed(k) s.t. n[i][j] ≠ 0 and r[j][j'] ≠ 0 do
 6     mdd u ← TCRelProdSat(n[i][j],r[j][j']);
 7     if u ≠ 0 then
 8       if t = 0 then t ← NewNode(k); endif
 9       t[i][j'] ← Or(t[i][j'],u);
10     endif
11   endfor
12 endfor
13 t ← TransClosureSat(UniqueTablePut(t));  CacheAdd_TCRelProdSat(n,r,t);
14 return t;
```

```
mdd TCtoSCC(mdd n)
 1 if n = 1 return 1;   if InCache_TCtoSCC(n,t) then return t;
 2 mdd t ← 0; level k ← n.lvl;
 3 foreach i ∈ S_Unprimed(k) s.t. n[i][i] ≠ 0 do
 4   if TCtoSCC(n[i][i]) ≠ 0 then
 5     if t = 0 then t ← NewNode(k); endif
 6     t[i] ← TCtoSCC(n[i][i]);
 7   endif
 8 endfor
 9 t ← UniqueTablePut(t);   CacheAdd_TCtoSCC(n,t);
10 return t;
```

Figure 4: Building the transitive closure using saturation.

(*cqn*) discussed in [15], two implementations of arbiters (*arbiter*1, *arbiter*2)[1], one which guarantees fairness and the other which does not, the N-queen problem (*queens*), the dining philosopher problem (*phil*) and the leader selection protocol (*leader*) [3]. The size for each model is parameterized with N.

```
mdd TSCC_TC(N^-1)
  1 mdd TC^-1 ← TransClosureSat(N^-1);     mdd TC ← Inverse(TC^-1);
  2 mdd SCC ← TCtoSCC(TC^-1);
  3 mdd L ← TC^-1 \ TC;
  4 mdd nontscc ← QuantifyUnprimed(L);
  5 mdd recurrent ← SCC \ nontscc;
  6 return recurrent;
```

Figure 5: Computing recurrent states using transitive closure.

```
mdd FairLoop_TC(S_rch, N^-1, {F_1, ..., F_n})
  1 mdd TC^-1 ← TransClosureSat(N^-1);     mdd TC ← Inverse(TC^-1);
  2 mdd S_weak ← S_rch;
  3 foreach m ∈ {1, ..., n}
  4    mdd p ← QuantifyUnprimed(TC^-1 ∧ TC ∧ (F_m × S_rch));
  5    S_weak ← S_weak ∩ p;
  6 endfor
  7 return S_weak;
```

Figure 6: Computing fair loops using transitive closure.

The number of SCCs (terminal SCCs) and states in SCCs (terminal SCCs) for each model obtained from (terminal) SCC enumeration is listed in column "SCC" ("TSCC") and column "States" respectively. The upper bounds for runtime and size of unique table (i.e., the storage for the MDD nodes) are set to 2 hours and 1GB respectively. The main metrics of our comparison are runtimes and peak memory consumption (for the unique table, storing the MDD nodes, plus the cache).

The top part of Table 1 compares three algorithms for SCC computation: the TC-based algorithm (column "TC") presented in Section 4, the improved Xie-Beerel algorithm (column "XBSat") presented in Section 3, and Lockstep (column "Previous algorithm") in Section 2.3. Coupled with saturation, the improved Xie-Beerel algorithm is better than Lockstep for most of the models in both runtime and memory. Compared with Lockstep, the TC-based algorithm is often more expensive. However, for two models, *queens* and $arbiter2$, the TC-based algorithm completes within the time limit while the other two algorithms fail. For $arbiter2$, our TC-based algorithm can explore over 10^{150} SCCs in a few seconds, while it is obviously not feasible for SCC enumeration algorithms to exhaustively enumerate all SCCs. To the best of our knowledge, this is the best result of SCC computation reported, stressing that the TC-based algorithm is not sensitive to the number of SCCs. With our new algorithm, the transitive closure can be built for some large systems, such as the dining philosopher problem with 1000 philosophers.

The bottom part of Table 1 compares the improved Xie-Beerel algorithm, *XBSaturation*, (column "XBSat") and algorithm *TSCC_TC_Sat* (column "TC"), presented in Section 3 and 4, respectively, for terminal SCC computation, with *XB_TSCC* (column "Previous algorithm") in Section 2.3. The basic trends are similar to the results of SCC computations, *XBSaturation* works consistently better than the original method, while *TSCC_TC* is less efficient for most models. In the Xie-Beerel framework, it is faster to compute terminal SCCs than all SCCs because a larger set of states is pruned in each recursion. On the contrary, *TSCC_TC* is more expensive than *SCC_TC* due to the computation of the $\mapsto$ relation, which has large memory and runtime requirements. Nevertheless, for models with large numbers of terminal SCCs, such as *queens*, *TSCC_TC* shows its advantage over the Xie-Beerel algorithm.

We conclude that saturation is effective in speeding up the SCC and terminal SCC computations within the framework of the Xie-Beerel algorithm. Also, our new saturation-based TC computation can

Model		SCC/TSCC	States	TC		XBSat		Previous algorithm	
name	N			mem(MB)	time(sec)	mem(MB)	time(sec)	mem(MB)	time(sec)
Results for the SCC computation									
cqn	10	11	2.09e+10	34.2	13.6	3.4	< 0.1	4.0	3.9
	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
phil	100	1	4.96e+62	5.0	0.5	3.2	< 0.1	52.0	4.5
	500	1	3.03e+316	33.0	4.0	24.5	0.1	–	to
	1000	1	9.18e+626	40.5	7.8	29.1	0.3	–	to
queens	10	3.22e+4	3.23e+4	8.2	1.6	64.4	14.5	63.9	12.4
	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
leader	3	4	6.78e+2	6.0	1.4	20.8	< 0.1	20.8	< 0.1
	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
*arbiter*1	10	1	2.05e+4	24.1	1.2	21.4	< 0.1	21.8	0.1
	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	–
*arbiter*2	10	1024	1.02e+4	20.3	< 0.1	26.2	0.7	31.1	1.1
	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
Results for the terminal SCC computation									
cqn	10	10	2.09e+10	37.9	15.5	21.4	< 0.1	33.5	3.4
	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
phil	100	2	2	26.5	0.5	20.9	< 0.1	39.2	8.7
	500	2	2	34.3	4.1	23.2	< 0.1	–	to
	1000	2	2	44.4	11.3	26.5	0.2	–	to
queens	10	1.28e+04	1.28e+4	36.2	3.0	46.7	2.8	62.3	35.1
	11	6.11e+04	6.11e+4	76.5	19.3	70.6	24.5	145.2	364.2
	12	3.14e+05	3.14e+5	244.1	205.4	98.8	179.4	mo	–
	13	1.72e+06	1.72e+6	mo	–	269.0	1940.81	mo	–
leader	3	3	3	26.6	1.5	20.7	< 0.1	21.4	0.1
	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
*arbiter*1	10	1	2.05e+4	24.1	1.2	20.4	< 0.1	22.4	0.4
	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
*arbiter*2	10	1	1	20.4	< 0.1	20.9	< 0.1	39.6	6.4
	15	1	1	20.5	< 0.1	40.6	4.6	–	to
	20	1	1	20.5	< 0.1	450.0	2897.8	–	to

Table 1: Results for SCC and terminal SCC computations.

tackle some complex models with up to 10^{150} states. Finally, for models with huge numbers of SCCs, the *TC*-based SCC computation has advantages over Lockstep, which detects SCCs one-by-one.

While our *TC*-based approach is not a replacement for Lockstep, we argue that it is an alternative worth further research. For a model with an unknown number of existing SCCs, employing both of these approaches at the same time could be ideal. Given current trends in multi-core processors, it is reasonable to run the two algorithms concurrently, possibly sharing some of the common data structures, such as the MDDs encoding the state space and next-state functions.

7 Conclusion

In this paper, we focus on improving two previous approaches to SCC computation, the Xie-Beerel algorithm and TC, using saturation. We first employ the saturation on the framework of the Xie-Beerel algorithm. In the context of the asynchronous models we study, the improved Xie-Beerel algorithm using saturation achieves a clear speedup. We also propose a new algorithm to compute TC using saturation. The experimental results demonstrate that our TC-based algorithm is capable of handling models with up to 10^{150} of SCCs. As we argue, the TC-based approach is worth further research because of its advantages when used on models with large numbers of SCCs.

References

[1] NuSMV: a new symbolic model checker. Available at http://nusmv.irst.itc.it/.

[2] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. In *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, 2000.

[3] Gianfranco Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at http://www.cs.ucr.edu/∼ciardo/SMART/.

[4] Gianfranco Ciardo, Robert L. Jones, Andrew S. Miner, and Radu Siminiceanu. Logical and stochastic modeling with SMART. *Perf. Eval.*, 63:578–608, 2006.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[6] Ramin Hojati, Hervé J. Touati, Robert P. Kurshan, and Robert K. Brayton. Efficient ω-regular language containment. In *CAV*, pages 396–409, 1992.

[7] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[8] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 1–16, London, UK, 1998. Springer-Verlag.

[9] Y. Matsunaga, P.C. McGeer, and R.K. Brayton. On computing the transitive closure of a state transition relation. In *Design Automation, 1993. 30th Conference on*, pages 260–265, June 1993.

[10] Tadao Murata. Petri nets: properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–579, April 1989.

[11] Kavita Ravi, Roderick Bloem, and Fabio Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 143–160, London, UK, 2000. Springer-Verlag.

[12] Fabio Somenzi, Kavita Ravi, and Roderick Bloem. Analysis of symbolic SCC hull algorithms. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 88–105, London, UK, 2002. Springer-Verlag.

[13] Min Wan and Gianfranco Ciardo. Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In M. Nielsen et al., editors, *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404, pages 582–594, Špindlerův Mlýn, Czech Republic, February 2009. Springer-Verlag.

[14] Aiguo Xie and Peter A. Beerel. Efficient state classification of finite-state Markov chains. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 17(12):1334–1339, 1998.

[15] Aiguo Xie and Peter A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(10):1225–1230, 2000.