Symbolic Computation of Strongly Connected Components using Saturation

Yang Zhao **Gianfranco Ciardo**

Department of Computer Science and Engineering

University of California at Riverside

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

Strongly Connected Components

- Finding strongly connected components (SCCs) is a basic problem in formal verification:
  - LTL and CTL model checking
  - Language emptiness check for ω-automata
- In Markov chain analysis, we need to partition the state space into transient vs. recurrent states (recurrent states are those that belong to terminal SCCs)
- It is impractical to enumerate SCCs using explicit algorithms for large discrete-state models
  $\Rightarrow$ use symbolic computation of SCCs
- Objectives: symbolically build the set of states in non-trivial (terminal) SCCs

Difficulties and approaches

Two difficulties:

- huge state spaces: the primary obstacle to formal verification
- potentially large number of (terminal) SCCs: a bottleneck for SCC enumeration algorithms

We propose two approaches based on previous ideas: the Xie-Beerel algorithm and transitive closure

- Saturation helps cope with the complexity of state-space exploration
- To cope with a large number of SCCs, we use a transitive closure-based algorithm:
  - Computing transitive closure based on saturation
  - Can support the computation of recurrent states

Structured discrete-state models

A structured discrete-state model is specified by $\langle \widehat{\mathcal{S}}, \mathcal{S}_{init}, \mathcal{E} \rangle$:

- a potential state space $\widehat{\mathcal{S}} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1$
  - the (global) state is of the form $\mathbf{i} = (i_L, ..., i_1)$
  - $\mathcal{S}_k$ is the (discrete) local state space for submodel k or local domain for state variable x_k
  - if $\mathcal{S}_k$ is finite, we can map it to $\{0, 1, \ldots, n_k - 1\}$ n_k is known after state-space generation
- a set of initial states $\mathcal{S}_{init} \subseteq \widehat{\mathcal{S}}$
  - often there is a single initial state $\mathbf{i}_{init}$
- a set of events $\mathcal{E}$ defining disjunctively-partitioned next-state functions or transition relation
  - $\mathcal{N}_\alpha : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$ iff state $\mathbf{j}$ can be reached by firing event α in state $\mathbf{i}$
  - $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$
  - naturally extended to sets of states $\mathcal{N}_\alpha(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}_\alpha(\mathbf{i})$ and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$
  - α is enabled in $\mathbf{i}$ iff $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$, otherwise it is disabled

Multi-way decision diagrams (MDDs)

An MDD is an acyclic directed edge-labeled graph where:

- The only terminal nodes can be $\mathbf{0}$ and $\mathbf{1}$, and are at level 0 $\quad \mathbf{0}.lvl = \mathbf{1}.lvl = 0$
- A nonterminal node p is at a level k, with $L \geq k \geq 1$ $\quad p.lvl = k$
- A nonterminal node is associated with a state variable x_k, with $L \geq k \geq 1$
- For each $i_k \in \mathcal{S}_k$, a nonterminal node p at level k has an outgoing edge pointing to child $p[i_k]$
- The level of a child is lower than that of p $\quad p[i_k].lvl < p.lvl$
- A node p at level k encodes the function $v_p : \mathcal{S}_k \times \cdots \times \mathcal{S}_1 \rightarrow \mathbb{B}$ defined recursively by

$$v_p(x_k, ..., x_1) = \begin{cases} p & \text{if } k = 0 \\ v_{p[x_k]}(x_{k-1}, ..., x_1) & \text{if } k > 0 \end{cases}$$

An L-level MDD encodes a set of states $\mathcal{X} \subseteq \widehat{\mathcal{S}} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1$

$\mathbf{i} \in \mathcal{X} \Leftrightarrow$ the path $(i_L, ..., i_1)$ from the root leads to terminal $\mathbf{1}$, corresponding to $\mathbf{i}$.

Using MDDs to encode next-state functions

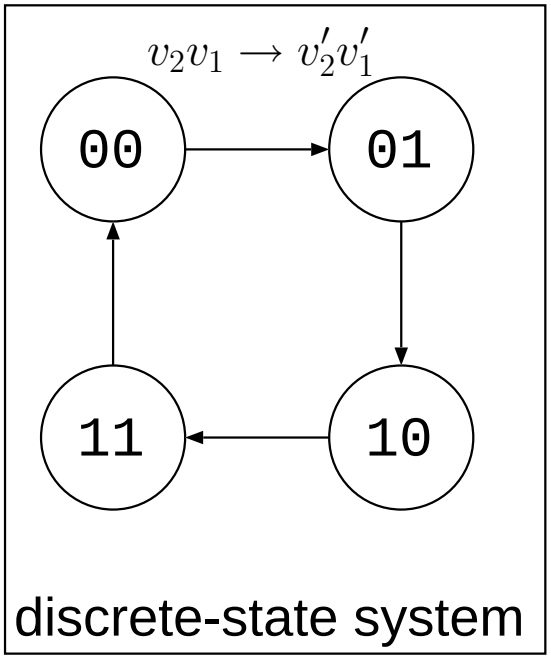
A $2L$-level MDD encodes the next-state function $\mathcal{N} : \widehat{\mathcal{S}} \rightarrow 2^{\widehat{\mathcal{S}}}$

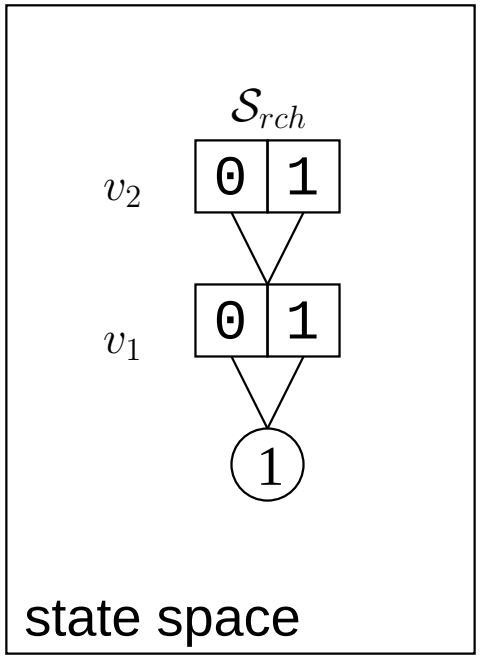
$\mathbf{j} \in \mathcal{N}(\mathbf{i}) \Leftrightarrow$ the path $(i_L, j_L, ..., i_1, j_1)$ from the root leads to terminal $\mathbf{1}$.

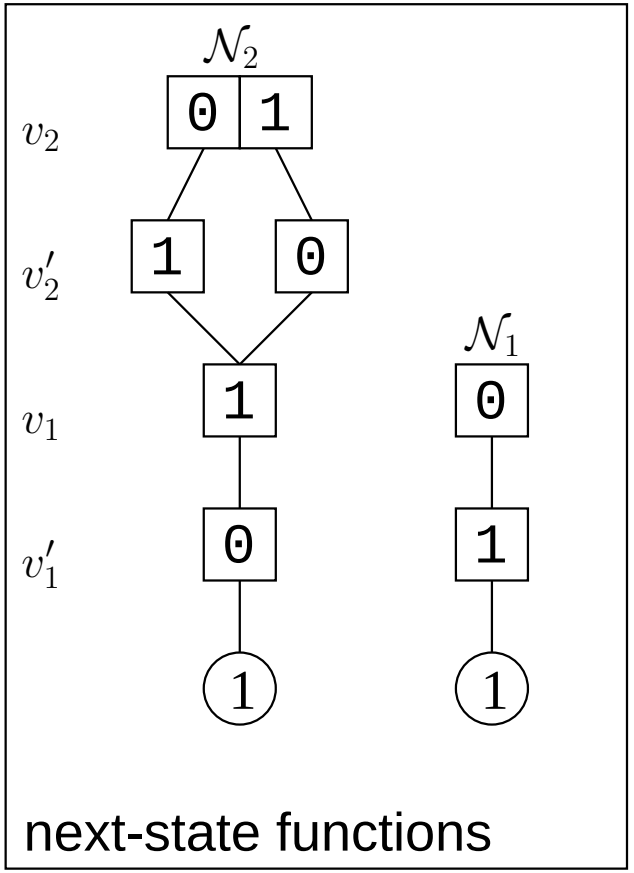
- α is independent of the k^{th} submodel if:
  - its enabling does not depend on i_k,
  - and its firing does not change the value of i_k.
- A level k belongs to $supp(\alpha)$, if α is not independent of k.
- Let $Top(\alpha)$ be the highest-numbered level in $supp(\alpha)$.
- Let $\mathcal{E}_k$ be the set of events $\{\alpha \in \mathcal{E} : Top(\alpha) = k\}$.
- Let $\mathcal{N}_k$ be the next-state function corresponding to all events in $\mathcal{E}_k$:

$$\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$$

Example: 2-bit counter







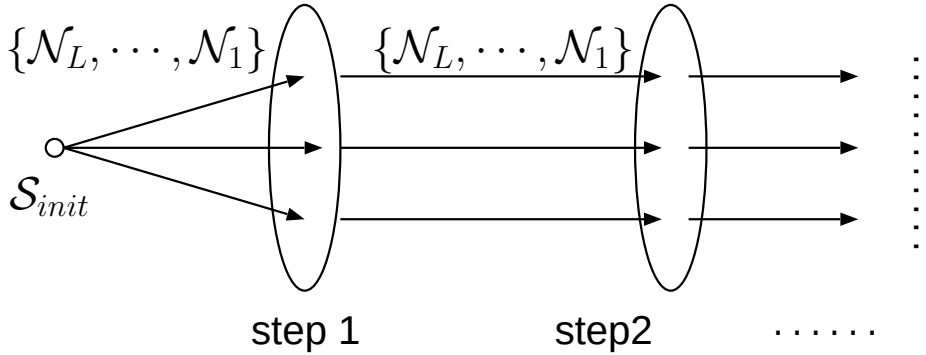
Saturation: an iteration strategy based on the model structure

MDD node p at level k is ***saturated*** if it encodes a fixpoint w.r.t. any event α s.t. $Top(\alpha) \leq k$

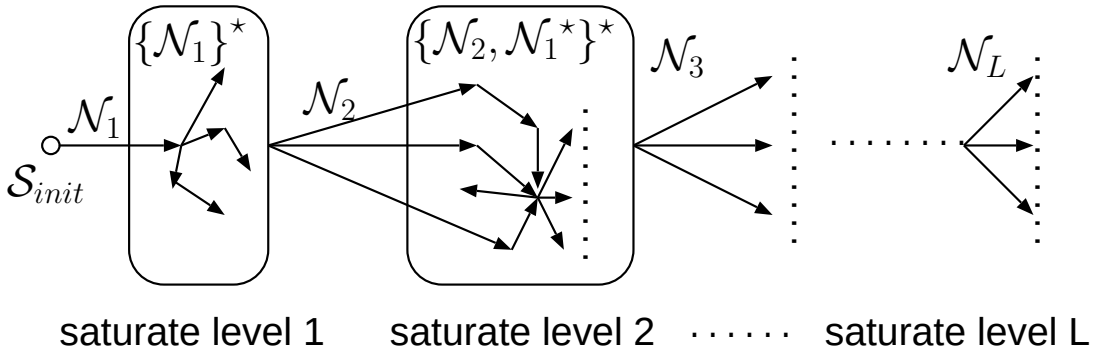
- build the L-level MDD encoding of $\mathcal{S}_{init}$ if $|\mathcal{S}_{init}| = 1$, there is one node per level
- saturate each node at level 1: fire in them all events α s.t. $Top(\alpha) = 1$
- saturate each node at level 2: fire in them all events α s.t. $Top(\alpha) = 2$
  (if this creates nodes at level 1, saturate them immediately upon creation)
- saturate each node at level 3: fire in them all events α s.t. $Top(\alpha) = 3$
  (if this creates nodes at levels 2 or 1, saturate them immediately upon creation)
- ...
- saturate the root node at level L: fire in it all events α s.t. $Top(\alpha) = L$
  (if this creates nodes at levels $L-1$, $L-2$, ..., 1, saturate them immediately upon creation)

Saturation vs. BFS

Breadth-first search (BFS):



Saturation:



- states are **not** discovered in breadth-first order
- enormous time and memory savings for asynchronous systems

Previous work

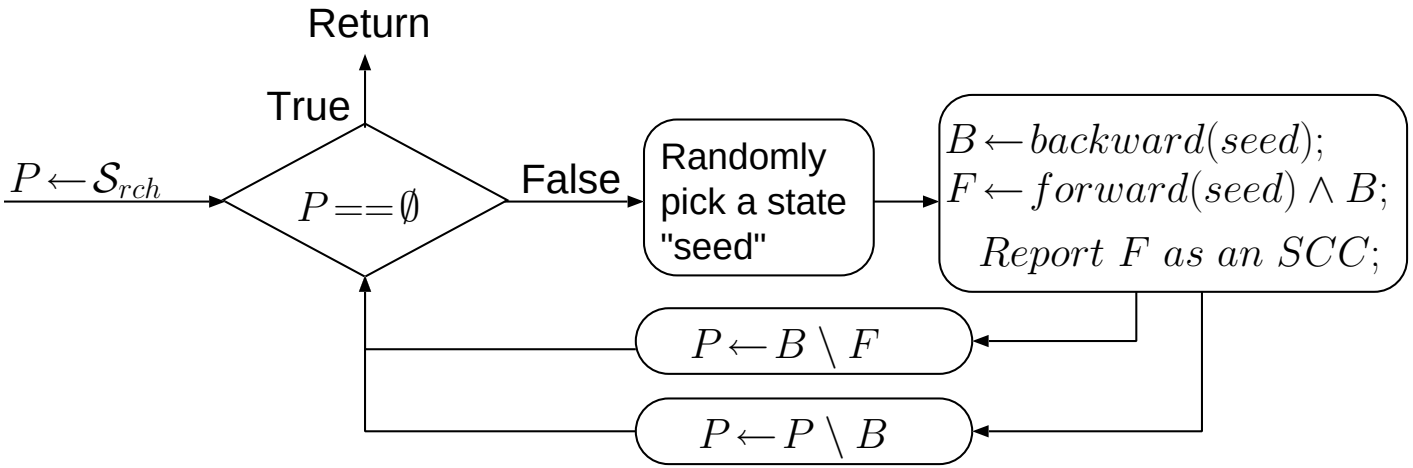
Two categories of related work: transitive closure and the SCC enumeration

- transitive closure: Hojati et al. presented as fully symbolic algorithm for testing ω-regular language containment by computing the transitive closure:

$$\mathcal{N}^+ = \mathcal{N} \cup \mathcal{N}^2 \cup \mathcal{N}^3 \cup \cdots$$

  - Due to the high complexity of computing the transitive closure, this approach has long been considered infeasible for complex systems.

- SCC enumeration: the Xie-Beerel algorithm combines both explicit state enumeration and symbolic state-space exploration.



The improved Xie-Beerel algorithm: Lockstep

Lockstep reduces the number of image computations w.r.t. the Xie-Beerel algorithm.

- It interleaves one forward and one backward step to compute forward and backward reachable states.
- It uses the earlier converged set of states to bound the other.
- Lockstep achieves $O(n \log n)$ complexity in the terms of steps.

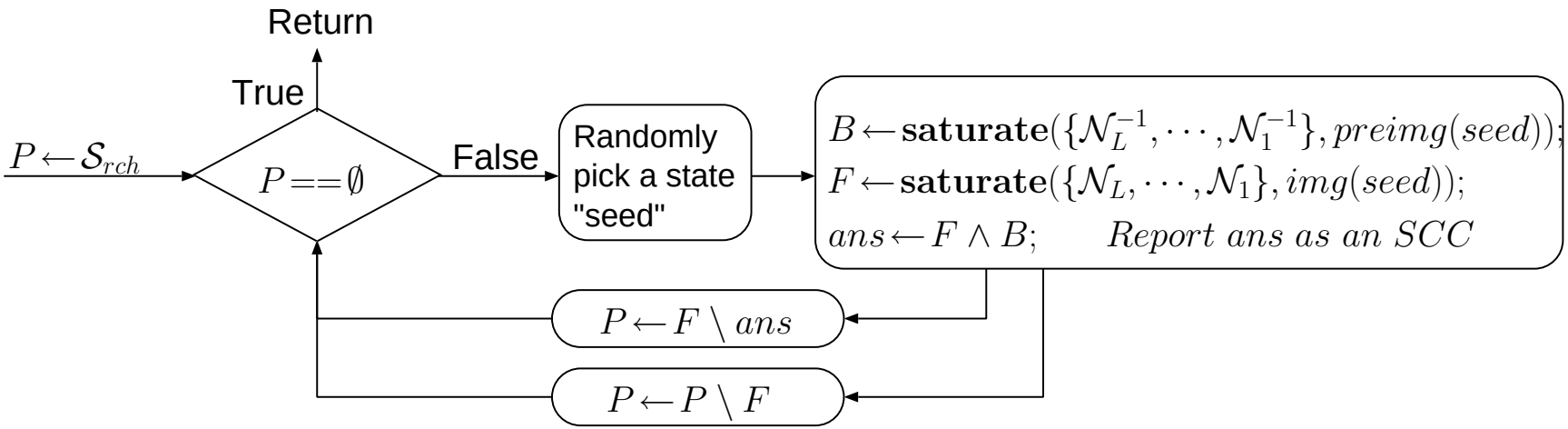
$mdd\ \ Lockstep(mdd\ P)$

```
1  ...
2  while(F_front ≠ ∅ and B_front ≠ ∅)
3      F_front ← N(F_front) ∩ P \ F; B_front ← N^-1(B_front) ∩ P \ B;
4      F ← F ∪ F_front;     B ← B ∪ B_front;
5  endwhile
6  if(F_front = ∅) then                          F converges earlier than B
7      mdd Conv ← F;
8      while(B_front ∩ F ≠ ∅) do
9          B_front ← N^-1(B_front) ∩ P \ B;
10         B ← B ∪ B_front;
11     endwhile
12 else
13 ...
```

Our contributions

Improving the Xie-Beerel algorithm using saturation

We employ saturation for the state-space exploration in the Xie-Beerel algorithm.



- Our algorithms compute B and F separately, unlike $Lockstep$, which uses the set that converges first to bound the other.
- The complexity of our algorithm and of $Lockstep$ are hard to compare (one saturation run vs. a bounded number of BFS steps)
  - Saturation executes a series of lightweight firings instead of global image computations, its complexity cannot be captured as a number of steps.
  - Saturation results in more compact decision diagrams during state-space exploration, often greatly reducing runtime and memory.

Improving the Xie-Beerel algorithm using saturation

$mdd\ \ XBSaturation(mdd\ P)$

1 if($P = \emptyset$) then return $\emptyset$;
2 $mdd\ ans \leftarrow \emptyset$; $mdd\ seed \leftarrow Pick(P)$;
3 $mdd\ F_{front} \leftarrow \mathcal{N}(seed) \cap P$; $mdd\ B_{front} \leftarrow \mathcal{N}^{-1}(seed) \cap P$;
4 $mdd\ F \leftarrow Saturate(\{\mathcal{N}_L \cdots \mathcal{N}_1\}, F_{front}) \cap P$;
5 $mdd\ B \leftarrow Saturate(\{\mathcal{N}_L^{-1} \cdots \mathcal{N}_1^{-1}\}, B_{front}) \cap P$;

---

6 $mdd\ C \leftarrow F \cap B$; if $C \neq \emptyset$ then $ans \leftarrow C$; endif — *Line $6-8$ are for computing SCCs*
7 $ans \leftarrow ans \cup XBSaturation(F \setminus C) \cup XBSaturation(P \setminus F)$;
8 return ans;

6' if $F \setminus B = \emptyset$ then $ans \leftarrow ans \cup F$; endif — *Line $6'-8'$ are for computing terminal SCCs*
7' $ans \leftarrow ans \cup XBSaturation(P \setminus B)$;
8' return ans;

Experimental results show that, for most models, the saturation-based Xie-Beerel algorithm outperforms $Lockstep$, sometimes by orders of magnitude.

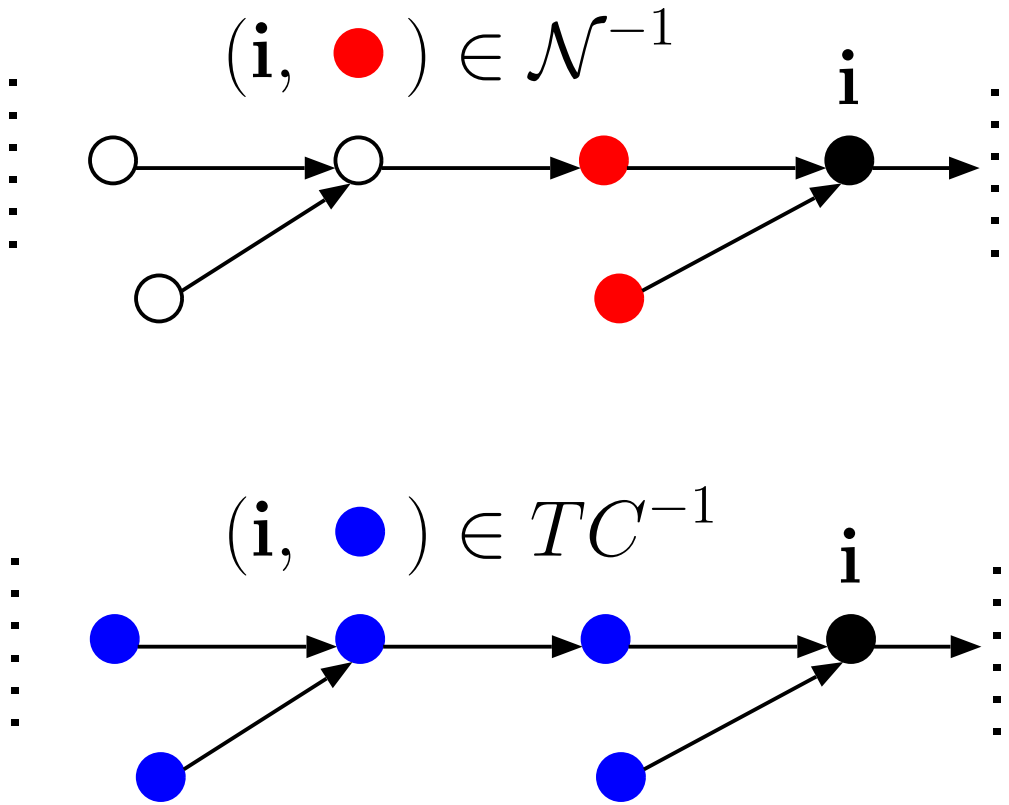
Our algorithm and Lockstep improve the Xie-Beerel algorithm in different ways

- Lockstep aims at reducing the number of image computations.
- Our algorithm aims as improving state-space exploration by scheduling event firings based on locality.

Transitive closure

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

Definition: 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 $\mathbf{j} \overset{+}{\rightarrow} \mathbf{i}$. Symmetrically, we can define TC where $(\mathbf{i}, \mathbf{j}) \in TC$ iff $\mathbf{i} \overset{+}{\rightarrow} \mathbf{j}$.



Computing the transitive closure

Can be described as a new state-space exploration problem:

- Potential state space: $(\mathbf{i}, \mathbf{j})$ where $\mathbf{i}, \mathbf{j} \in \mathcal{S}_{rch}$.
- Initial states: $\{(\mathbf{i}, \mathbf{j}) | (\mathbf{i}, \mathbf{j}) \in \mathcal{N}^{-1}\}$.
- Next-state function $\mathcal{N}'$:

$$\mathcal{N}'(\ (\mathbf{i}, \mathbf{j})\ ) = \{(\mathbf{i}, \mathbf{k}) | \mathbf{k} \in \mathcal{N}^{-1}(\mathbf{j})\}$$

Our algorithm using saturation is based on the following observation:

if $(\mathbf{i}, \mathbf{k}) \in \mathcal{N}^{-1}$ then $(\mathbf{i}, \mathbf{j}) \in TC^{-1}$ where $\mathbf{j} \in Saturate(\{\mathcal{N}_L^{-1}, \cdots, \mathcal{N}_1^{-1}\}, \{\mathbf{k}\})$

Top-level pseudocode:

$mdd\ \ SCC\_TC(\mathcal{N}^{-1})$

1 $mdd\ TC^{-1} \leftarrow TransClosureSat(\mathcal{N}^{-1})$;
2 $mdd\ SCC \leftarrow TCtoSCC(TC^{-1})$;
3 return SCC;

Finding all $(\mathbf{i}, \mathbf{i}) \in TC^{-1}$

Computing the transitive closure (cont.)


```
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
 4     t[i][j] ← TransClosureSat(n[i][j]);
 5 endfor
 6 foreach i ∈ S_{Unprimed(k)} s.t. n[i] ≠ 0
 7     repeat                                          Build a local fixed point
 8         foreach j, j' ∈ S_{Unprimed(k)} s.t. n[i][j] ≠ 0 and r[j][j'] ≠ 0 do
 9             mdd u ← TCRelProdSat(t[i][j], r[j][j']);  t[i][j'] ← Or(t[i][j'], u);
10         endfor
11     until t does not change;
12 endfor
13 t ← UniqueTablePut(t);  CacheAdd_TransClosureSat(n, t);
14 return t;
```

Similar to the idea of saturation, this function runs node-wise on primed level and fires lower level events exhaustively until the local fixed point is obtained.

Computing terminal SCCs using transitive closure

$\mathbf{j}$ belongs to a terminal SCC iff

$$\forall \mathbf{i}, \mathbf{j} \xrightarrow{+} \mathbf{i} \Longrightarrow \mathbf{i} \xrightarrow{+} \mathbf{j}$$

Given states $\mathbf{i}, \mathbf{j}$, let $\mathbf{j} \mapsto \mathbf{i}$ denote that $\mathbf{j} \xrightarrow{+} \mathbf{i}\ and\ \neg(\mathbf{i} \xrightarrow{+} \mathbf{j})$.

Encode this relation with a $2L$-level MDD, which can be obtained as $TC^{-1} \setminus TC$.

$mdd\ \ TSCC\_TC(\mathcal{N}^{-1})$

1 $mdd\ TC^{-1} \leftarrow TransClosureSat(\mathcal{N}^{-1})$; $\quad mdd\ TC \leftarrow Inverse(TC^{-1})$;
2 $mdd\ SCC \leftarrow TCtoSCC(TC^{-1})$;
3 $mdd\ L \leftarrow TC^{-1} \setminus TC$;
4 $mdd\ nontscc \leftarrow QuantifyUnprimed(L)$;
5 $mdd\ recurrent \leftarrow SCC \setminus nontscc$;
6 return $recurrent$;

- To the best of our knowledge, this is the first symbolic algorithm for terminal SCC computation using 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.
  It is the only known algorithm applicable to models with a huge number of terminal SCCs.

Büchi fairness

Büchi fairness (weak fairness) can be specified as a set of sets of states $\{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$.

A fair loop satisfies Büchi fairness iff it contains a state in $\mathcal{F}_i$, for each $i = \{1, \ldots, n\}$

TC-based approach: Assume TC and TC^{-1} have been built, let

$$\mathcal{S}_{weak} = \left\{ \mathbf{i} \;\middle|\; \bigcap_{m=1,\ldots,n} [\exists \mathbf{f}_m \in \mathcal{F}_m . (TC(\mathbf{f}_m, \mathbf{i}) \wedge TC^{-1}(\mathbf{f}_m, \mathbf{i}))] \right\}$$

$\mathcal{S}_{weak}$ contains all the states in fair loops.

Experimental results of SCC computations

Model		SCCs	States	TC		XBSat		Lockstep	
name	N		in SCCs	mem(MB)	time(sec)	mem(MB)	time(sec)	mem(MB)	time(sec)
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
arbiter1	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	–
arbiter2	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

Experimental results of terminal SCC computations

Model		TSCCs	States	TC		XBSat		XBBFS	
name	N		in TSCCs	mem(MB)	time(sec)	mem(MB)	time(sec)	mem(MB)	time(sec)
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
arbiter1	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
arbiter2	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

Conclusion

- Saturation is effective in speeding up the SCC and terminal SCC computations within the framework of the Xie-Beerel algorithm.
- Our new saturation-based TC computation can tackle some complex models with up to 10^{150} states.
- For models with huge numbers of SCCs, the TC-based SCC computation has advantages over Lockstep, which symbolically explores one SCC at a time.

Our TC-based approach is not a replacement for Lockstep, but is an alternative worth further research.

For modelds with unknown number of SCCs, employing both approaches concurrently could be ideal.

Future work: 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.