Automata Theory and Formal Languages

Summer Term 2025

22nd Lecture

Program Verification

Prof. Dr. Matthias Mnich *Institute for Algorithms and Complexity*

Verification

Example

An automaton for the reachable configurations of a program

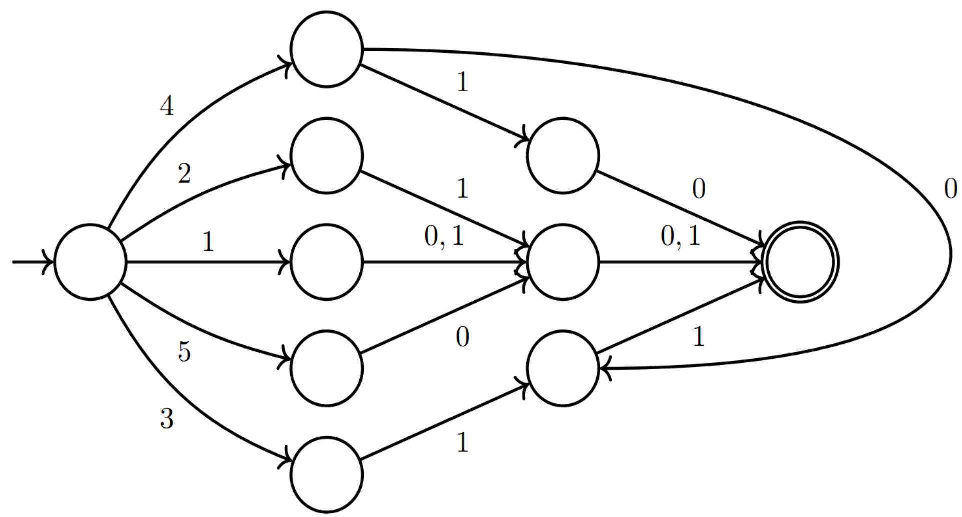
```
while x = 1 do
    if y = 1 then
        x ← 0
    y ← 1 − x
end
```

Verification

Example

An automaton for the reachable configurations of a program

while $x = 1$ ***do***
    if $y = 1$ ***then***
        $x \leftarrow 0$
    $y \leftarrow 1 - x$
end



Based on Sections 8.1-8.4 of Esparza: Automata Theory - An algorithmic approach

Verification

- We use languages to describe the implementation and the specification of a system.

Verification

- We use languages to describe the implementation and the specification of a system.

- We reduce the verification problem to language inclusion between implementation and specification

Verification

```
while x = 1 do
    if y = 1 then
        x ← 0
    y ← 1 − x
end
```

- Configuration: triple $[\ell, n_x, n_y]$ where
  - ℓ is the current value of the program counter, and
  - n_x, n_y are the current values of x, y

Verification

```
1  while x = 1 do
2     if y = 1 then
3        x ← 0
4     y ← 1 − x
5  end
```

- Configuration: triple $[\ell, n_x, n_y]$ where
  - ℓ is the current value of the program counter, and
  - n_x, n_y are the current values of x, y

Examples: $[1, 1, 1], [5, 0, 1]$

Verification

```
1  while x = 1 do
2      if y = 1 then
3          x ← 0
4      y ← 1 − x
5  end
```

- Configuration: triple $[\ell, n_x, n_y]$ where
  - ℓ is the current value of the program counter, and
  - n_x, n_y are the current values of x, y

Examples: $[1, 1, 1], [5, 0, 1]$

- Initial configuration: configuration with $\ell = 1$

Verification

$$
\begin{array}{ll}
1 & \textbf{\textit{while}}\ x = 1\ \textbf{\textit{do}} \\
2 & \quad \textbf{\textit{if}}\ y = 1\ \textbf{\textit{then}} \\
3 & \quad\quad x \leftarrow 0 \\
4 & \quad y \leftarrow 1 - x \\
5 & \textbf{\textit{end}}
\end{array}
$$

- Configuration: triple $[\ell, n_x, n_y]$ where
  - ℓ is the current value of the program counter, and
  - n_x, n_y are the current values of x, y

Examples: $[1, 1, 1], [5, 0, 1]$

- Initial configuration: configuration with $\ell = 1$
- Potential execution: finite or infinite sequence of configurations

Verification

```
1  while x = 1 do
2      if y = 1 then
3          x ← 0
4      y ← 1 − x
5  end
```

- Configuration: triple $[\ell, n_x, n_y]$ where
  - ℓ is the current value of the program counter, and
  - n_x, n_y are the current values of x, y

Examples: $[1, 1, 1], [5, 0, 1]$

- Initial configuration: configuration with $\ell = 1$
- Potential execution: finite or infinite sequence of configurations

Examples:

$[1, 1, 1][4, 1, 0]$

$[2, 1, 0][5, 1, 0]$

$[1, 1, 0][2, 1, 0][4, 1, 0][1, 1, 0]$

Verification

```
while x = 1 do
    if y = 1 then
        x ← 0
    y ← 1 − x
end
```

- Execution: potential execution starting at an initial configuration, and where configurations are followed by their “legal successors” according to the program semantics.

Verification

```
1  while x = 1 do
2      if y = 1 then
3          x ← 0
4      y ← 1 − x
5  end
```

- Execution: potential execution starting at an initial configuration, and where configurations are followed by their "legal successors" according to the program semantics.

Examples: $[1, 1, 1][2, 1, 1][3, 1, 1][4, 0, 1][1, 0, 1][5, 0, 1]$
$[1, 1, 0][2, 1, 0][4, 1, 0][1, 1, 0]$

Verification

```
1  while x = 1 do
2     if y = 1 then
3        x ← 0
4     y ← 1 − x
5  end
```

- Execution: potential execution starting at an initial configuration, and where configurations are followed by their "legal successors" according to the program semantics.

Examples: $[1, 1, 1][2, 1, 1][3, 1, 1][4, 0, 1][1, 0, 1][5, 0, 1]$
$[1, 1, 0][2, 1, 0][4, 1, 0][1, 1, 0]$

- Full execution: execution that cannot be extended (either infinite or ending at a configuration without successors)

Verification as a language problem

- Implementation: set E of executions

Verification as a language problem

- Implementation: set E of executions
- Specification:
  - subset P of the potential executions that satisfy a property, or
  - subset V of the potential executions that violate a property

Verification as a language problem

- Implementation: set E of executions
- Specification:
  - subset P of the potential executions that satisfy a property, or
  - subset V of the potential executions that violate a property
- Implementation satisfies specification if:
  - $E \subseteq P$, or
  - $E \cap V = \emptyset$

Verification as a language problem

- Implementation: set E of executions
- Specification:
  - subset P of the potential executions that satisfy a property, or
  - subset V of the potential executions that violate a property
- Implementation satisfies specification if:
  - $E \subseteq P$, or
  - $E \cap V = \emptyset$
- If E and P regular: inclusion checkable with automata
- If E and V regular: disjointness checkable with automata

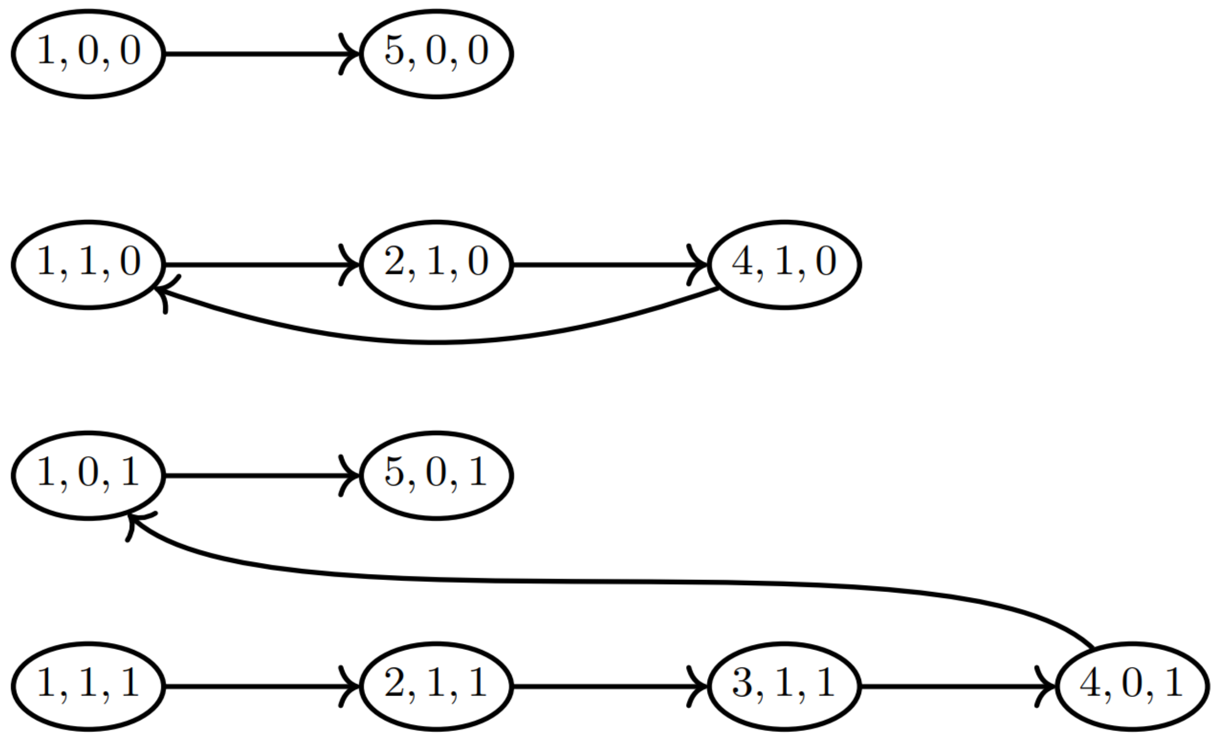
Verification as a language problem

- Implementation: set E of executions
- Specification:
  - subset P of the potential executions that satisfy a property, or
  - subset V of the potential executions that violate a property
- Implementation satisfies specification if:
  - $E \subseteq P$, or
  - $E \cap V = \emptyset$
- If E and P regular: inclusion checkable with automata
- If E and V regular: disjointness checkable with automata

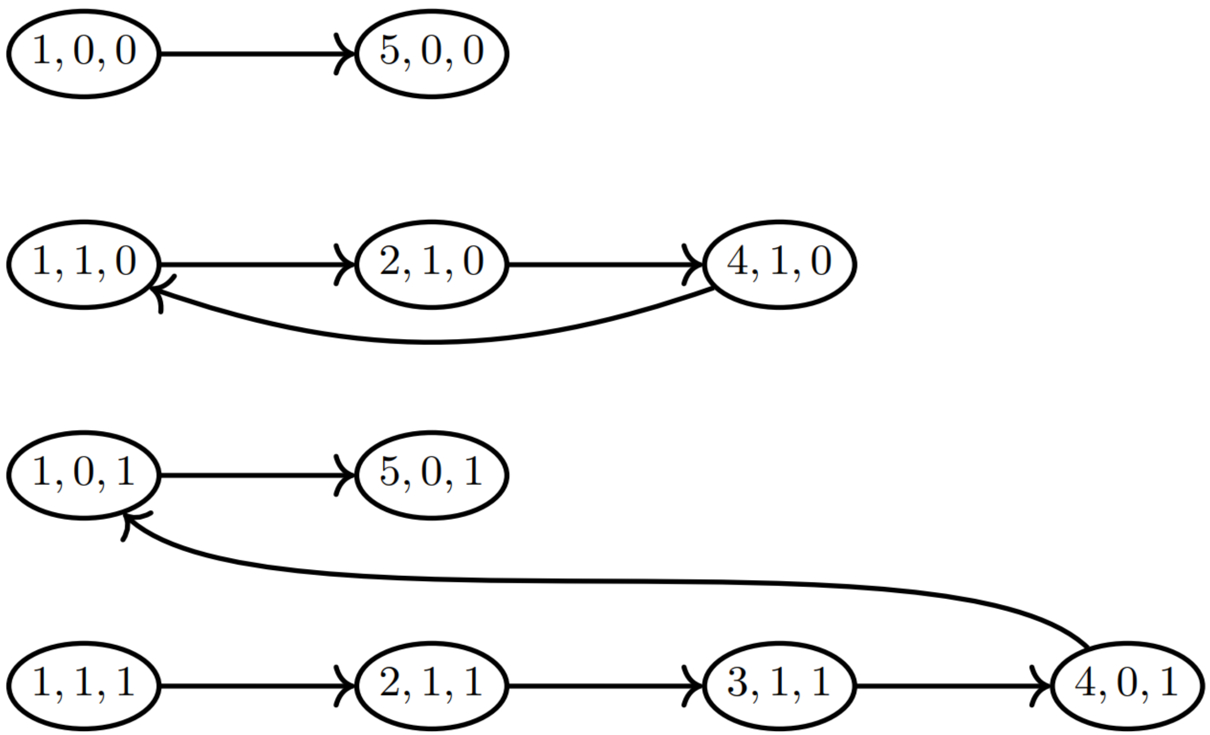
How often is the case that E, P, V are regular?

System NFA

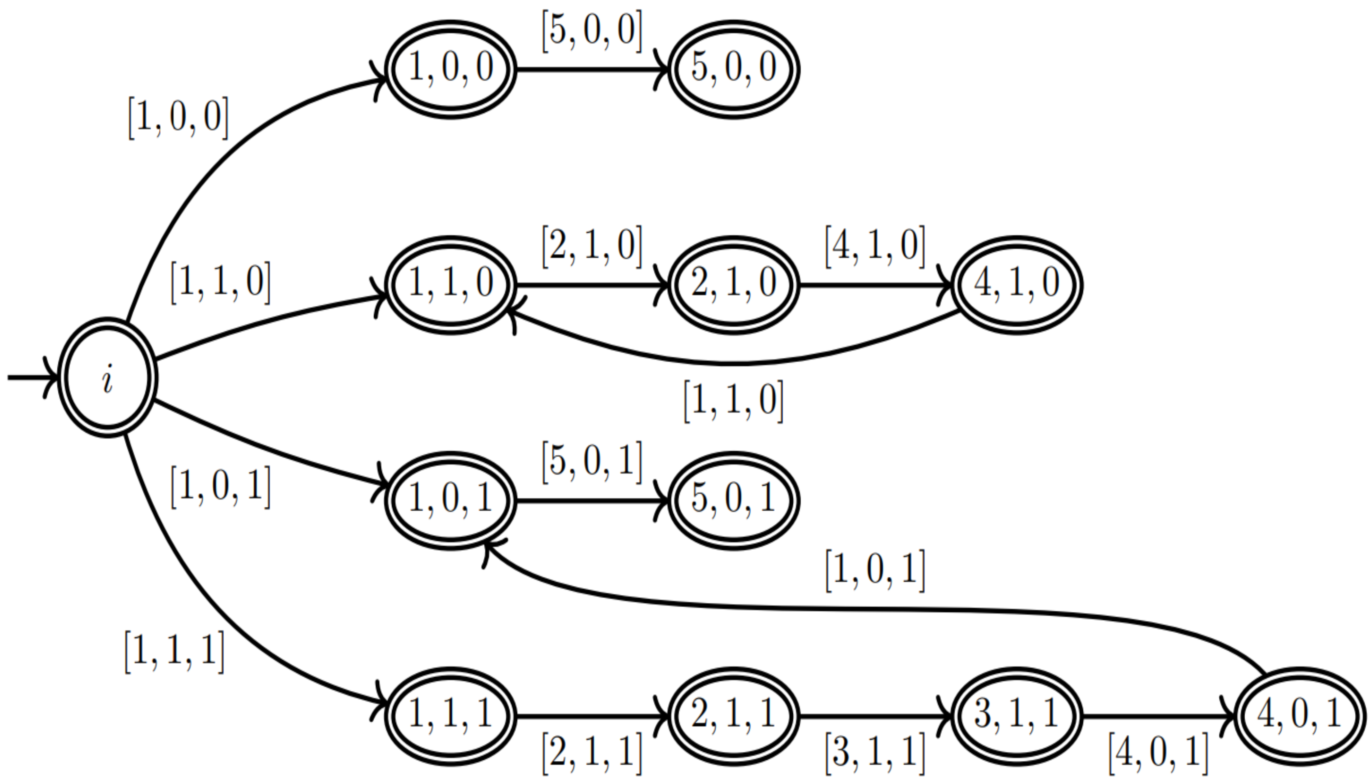
$\textbf{\textit{while}}\ x = 1\ \textbf{\textit{do}}$
$\quad \textbf{\textit{if}}\ y = 1\ \textbf{\textit{then}}$
$\qquad x \leftarrow 0$
$\quad y \leftarrow 1 - x$
$\textbf{\textit{end}}$



System NFA



System NFA


i
[1, 0, 0]
1, 0, 0
[5, 0, 0]
5, 0, 0
[1, 1, 0]
1, 1, 0
[2, 1, 0]
2, 1, 0
[4, 1, 0]
4, 1, 0
[1, 1, 0]
[1, 0, 1]
1, 0, 1
[5, 0, 1]
5, 0, 1
[1, 0, 1]
[1, 1, 1]
1, 1, 1
[2, 1, 1]
2, 1, 1
[3, 1, 1]
3, 1, 1
[4, 0, 1]
4, 0, 1

Property NFA

- Is there a full execution such that
  - initially $\ell = 1$
  - finally $y = 0$, and
  - y never increases?

Property NFA

- Is there a full execution such that
  - initially $\ell = 1$
  - finally $y = 0$, and
  - y never increases?

- Set of potential executions for this property:

$$[\ell, x, 1][\ell, x, 1]^*[\ell, x, 0]^*[5, x, 0]$$

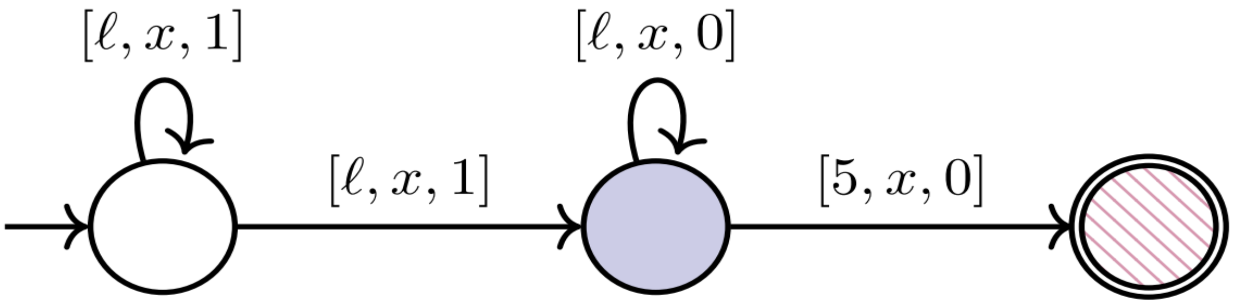
Property NFA

- Is there a full execution such that
  - initially $\ell = 1$
  - finally $y = 0$, and
  - y never increases?

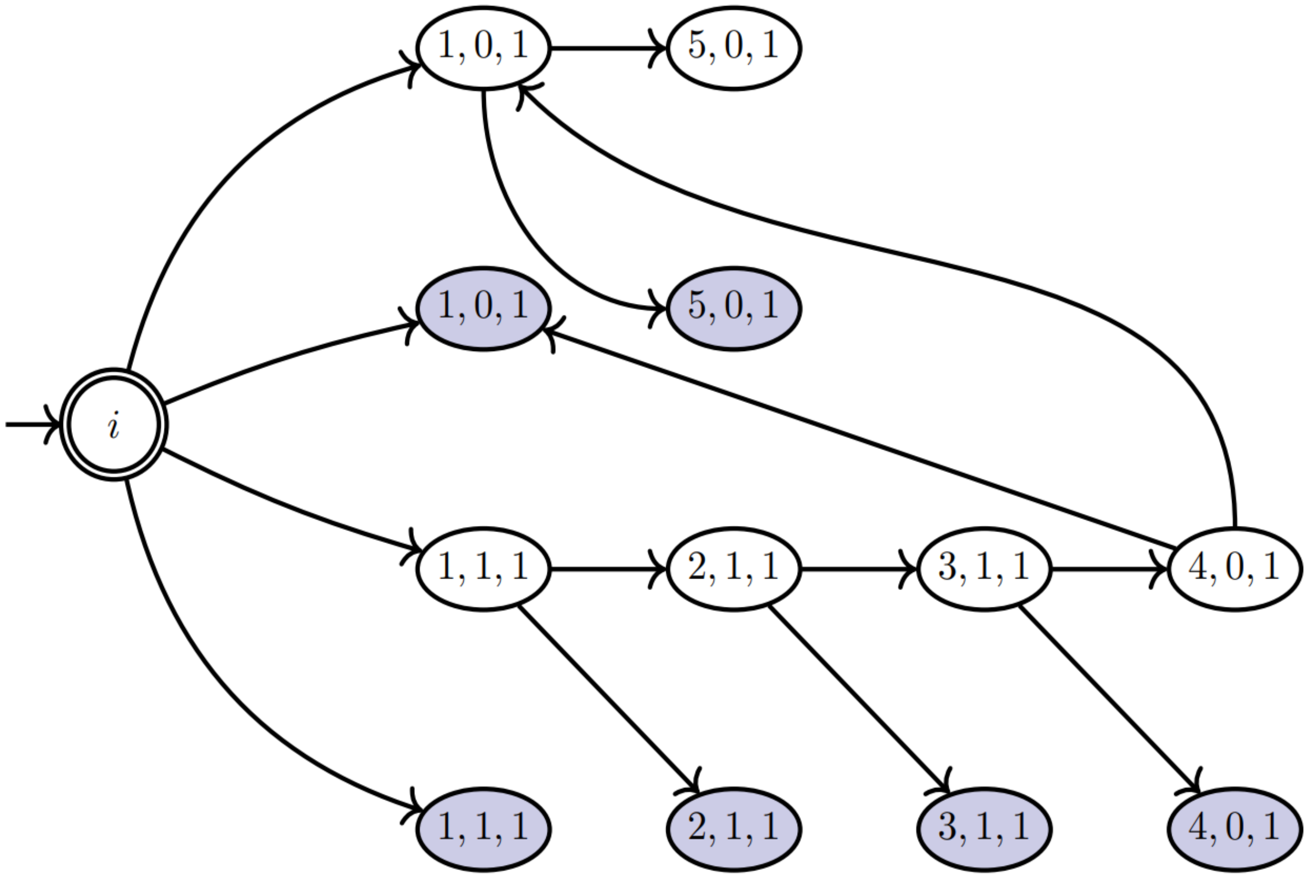
- Set of potential executions for this property:

$$[\ell, x, 1][\ell, x, 1]^*[\ell, x, 0]^*[5, x, 0]$$

- Automaton for this set:



Intersection of the system and property NFAs



Automaton is empty, and so no execution satisfies the property

Another property

- Is the assignment $y \leftarrow x - 1$ redundant?

Another property

- Is the assignment $y \leftarrow x - 1$ redundant?

- Potential executions where the assignment is executed at least once and it changes the value of y:

$$[\ell, x, y]^* \big([4, x, 0][1, x, 1] + [4, x, 1][1, x, 0]\big) [\ell, x, y]^*$$

Another property

- Is the assignment $y \leftarrow x - 1$ redundant?

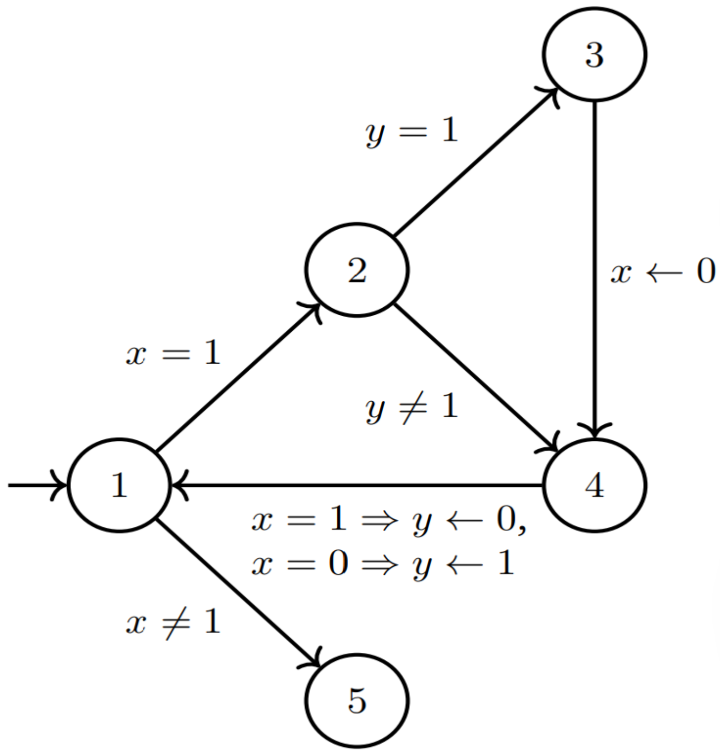
- Potential executions where the assignment is executed at least once and it changes the value of y:

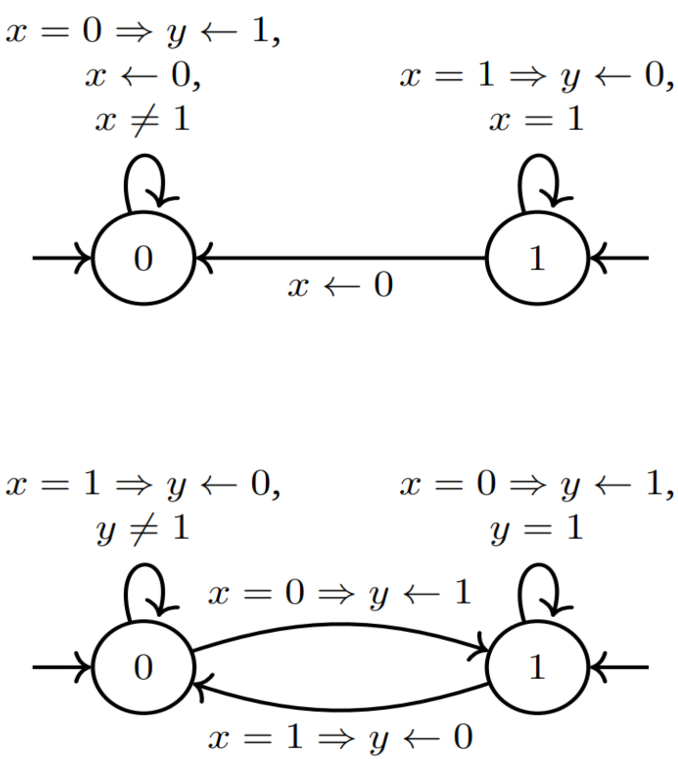
$$[\ell, x, y]^* \big([4, x, 0][1, x, 1] + [4, x, 1][1, x, 0]\big) [\ell, x, y]^*$$

- Therefore: assignment redundant iff none of these potential executions is a real execution of the program.

Networks of Automata

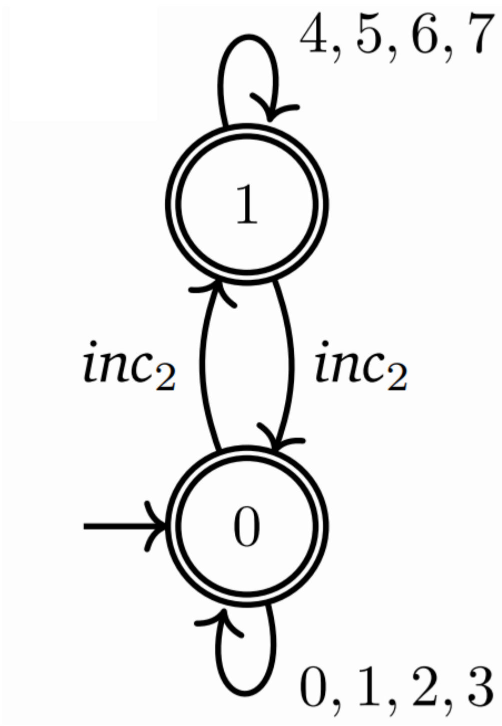
while $x = 1$ **do**
  if $y = 1$ **then**
    $x \leftarrow 0$
  $y \leftarrow 1 - x$
end



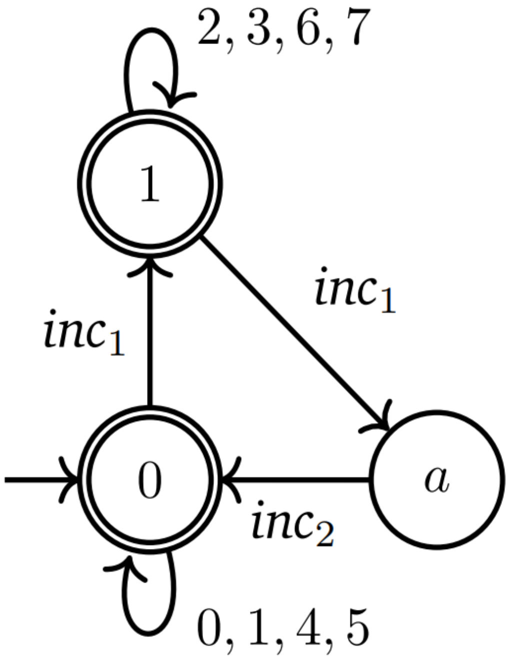


Networks of Automata

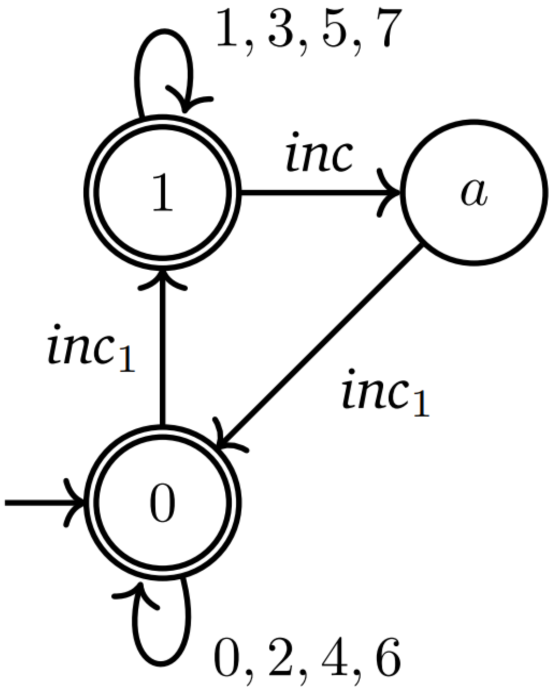
A_2



A_1



A_0

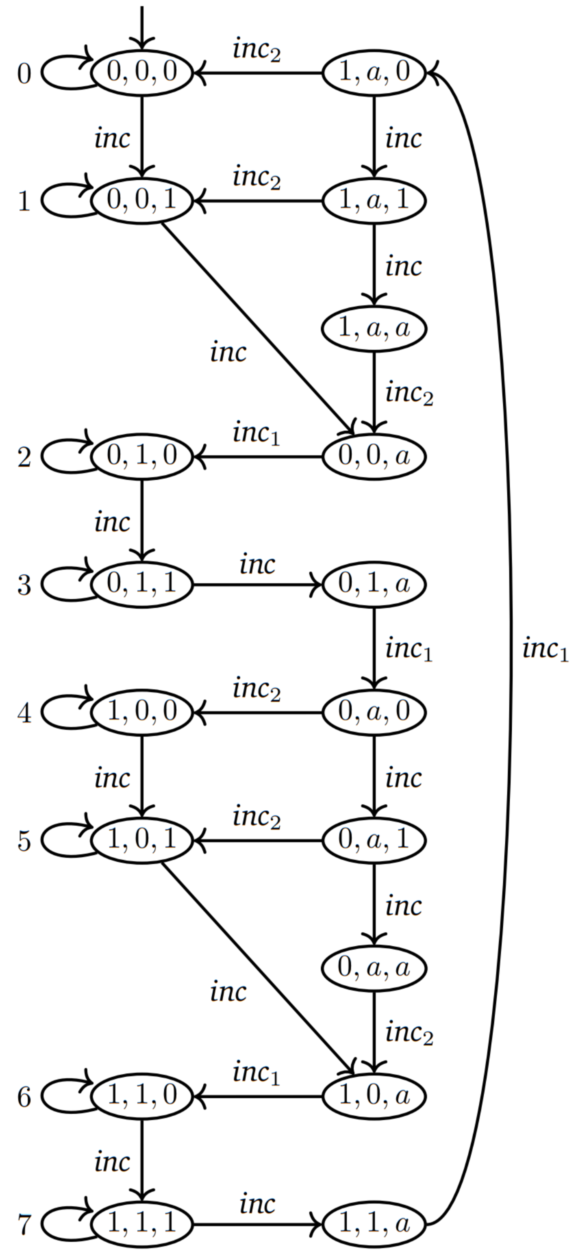


Networks of Automata

- Tuple $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$ of NFAs.
- Each NFA has its own alphabet Σ_i of actions.
- Alphabets usually not disjoint!
- A_i participates in action a if $a \in \Sigma_i$.
- A configuration is a tuple $\langle q_1, \ldots, q_n \rangle$ of states, one for each automaton of the network.
- $\langle q_1, \ldots, q_n \rangle$ enables a if every participant in a is in a state from which an a-transition is possible.
- Enabled actions can occur, and their occurrence simultaneously changes the states of their participants. Non-participants stay idle and don't change their states.

Networks of Automata

Configuration graph of the network



Asynchronous product

AsyncProduct$(A_1, \ldots, A_n)$

Input: a network of automata $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$, where
$A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$ for every $i \in \{1, \ldots, n\}$

Output: NFA $A_1 \otimes \cdots \otimes A_n = (Q, \Sigma, \delta, Q_0, F)$ recognizing $\mathcal{L}(\mathcal{A})$

$Q, \delta, F \leftarrow \emptyset$
$Q_0 \leftarrow Q_{01} \times \cdots \times Q_{0n}$
$W \leftarrow Q_0$
while $W \neq \emptyset$ **do**
    pick $[q_1, \ldots, q_n]$ **from** W
    add $[q_1, \ldots, q_n]$ **to** Q
    if $\bigwedge_{i=1}^{n} q_i \in F_i$ **then add** $[q_1, \ldots, q_n]$ **to** F
    for all $a \in \Sigma_1 \cup \ldots \cup \Sigma_n$ **do**
        for all $i \in [1..n]$ **do**
            if $a \in \Sigma_i$ **then** $Q'_i \leftarrow \delta_i(q_i, a)$ **else** $Q'_i = \{q_i\}$
        for all $[q'_1, \ldots, q'_n] \in Q'_1 \times \ldots \times Q'_n$ **do**
            if $[q'_1, \ldots, q'_n] \notin Q$ **then add** $[q'_1, \ldots, q'_n]$ **to** W
            add $([q_1, \ldots, q_n], a, [q'_1, \ldots, q'_n])$ **to** δ
return $Q, \Sigma_1 \cup \cdots \cup \Sigma_n, \delta, Q_0, F$

Networks of Automata

Concurrent programs as networks of automata: Lamport's 1-bit algorithm

Two processes $i = 0, 1$ with shared variables $b_0, b_1 \in \{0, 1\}$

Networks of Automata

Concurrent programs as networks of automata: Lamport's 1-bit algorithm

Two processes $i = 0, 1$ with shared variables $b_0, b_1 \in \{0, 1\}$

Process 0

  repeat

nc_0: $b_0 \leftarrow 1$

t_0: **while** $b_1 = 1$ **do skip**

c_0: $b_0 \leftarrow 0$

  forever

Networks of Automata

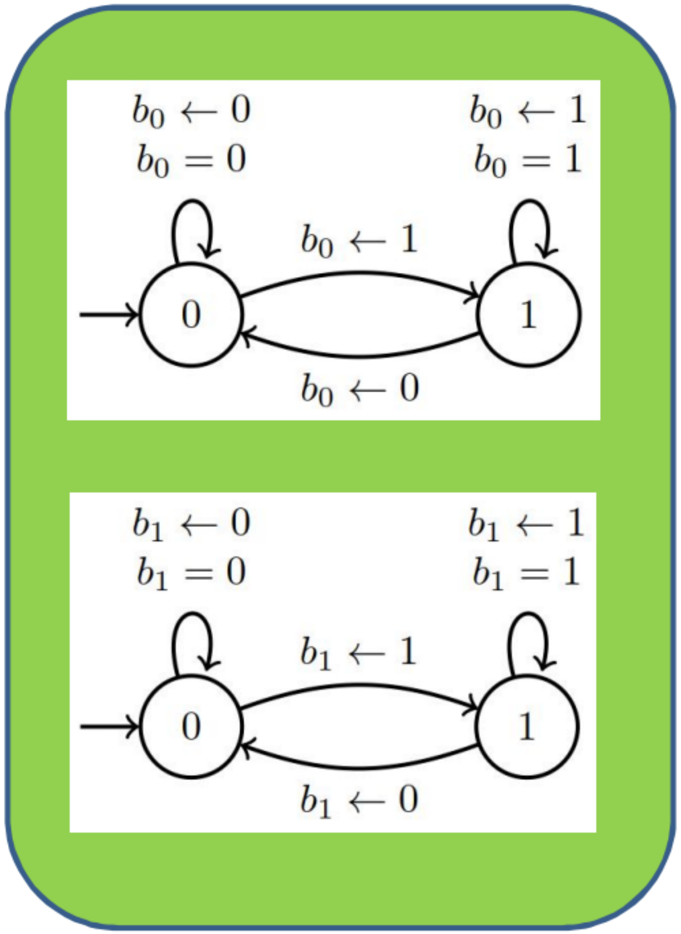
Concurrent programs as networks of automata: Lamport's 1-bit algorithm

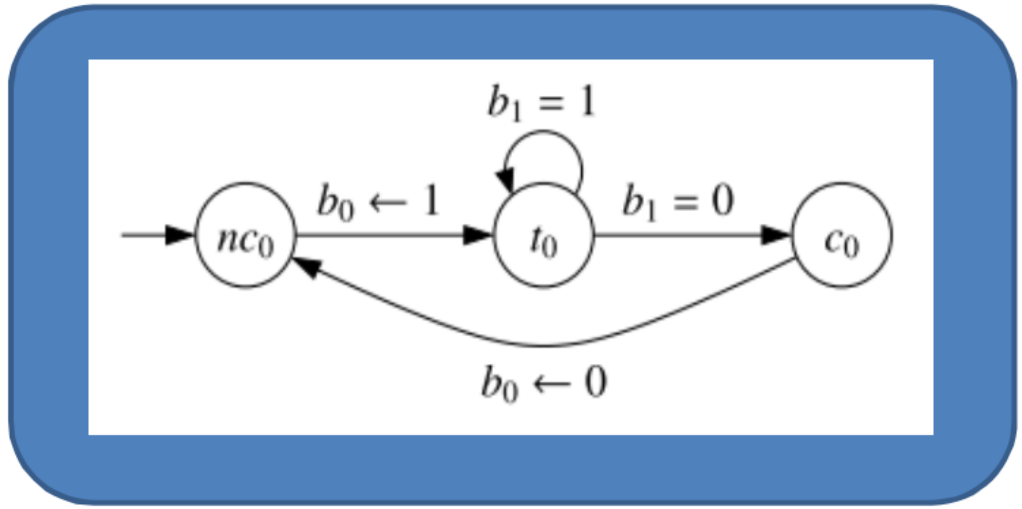
Two processes $i = 0, 1$ with shared variables $b_0, b_1 \in \{0, 1\}$

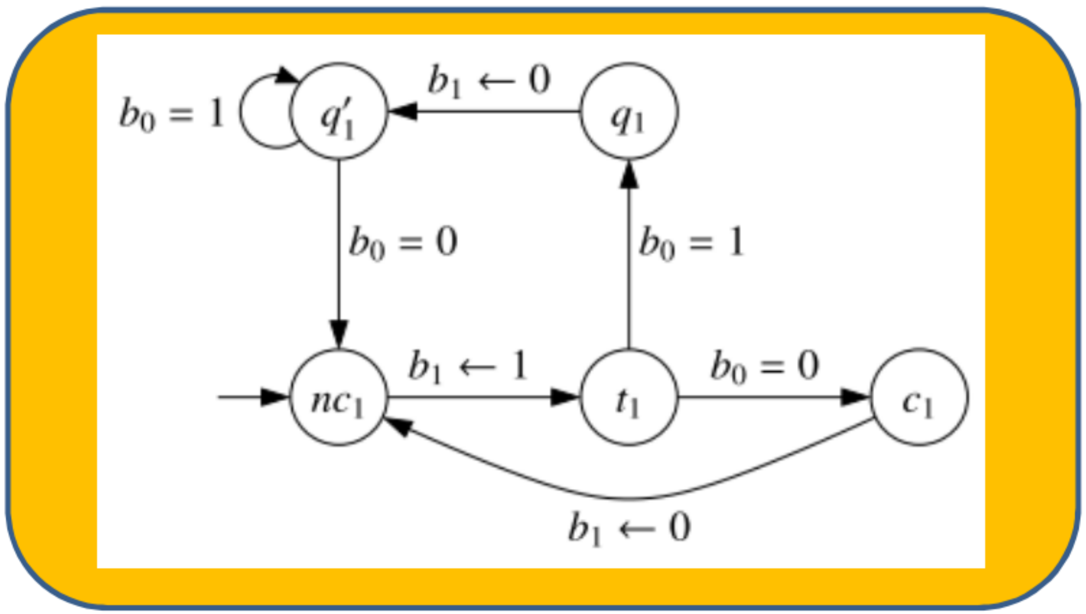
Process 0
  repeat
nc_0: $b_0 \leftarrow 1$
t_0: **while** $b_1 = 1$ **do skip**
c_0: $b_0 \leftarrow 0$
  forever

Process 1
  repeat
nc_1: $b_1 \leftarrow 1$
t_1: **if** $b_0 = 1$ **then**
q_1: $b_1 \leftarrow 0$
q_1': **while** $b_0 = 1$ **do skip**
  goto nc_1
c_1: $b_1 \leftarrow 0$ **forever**

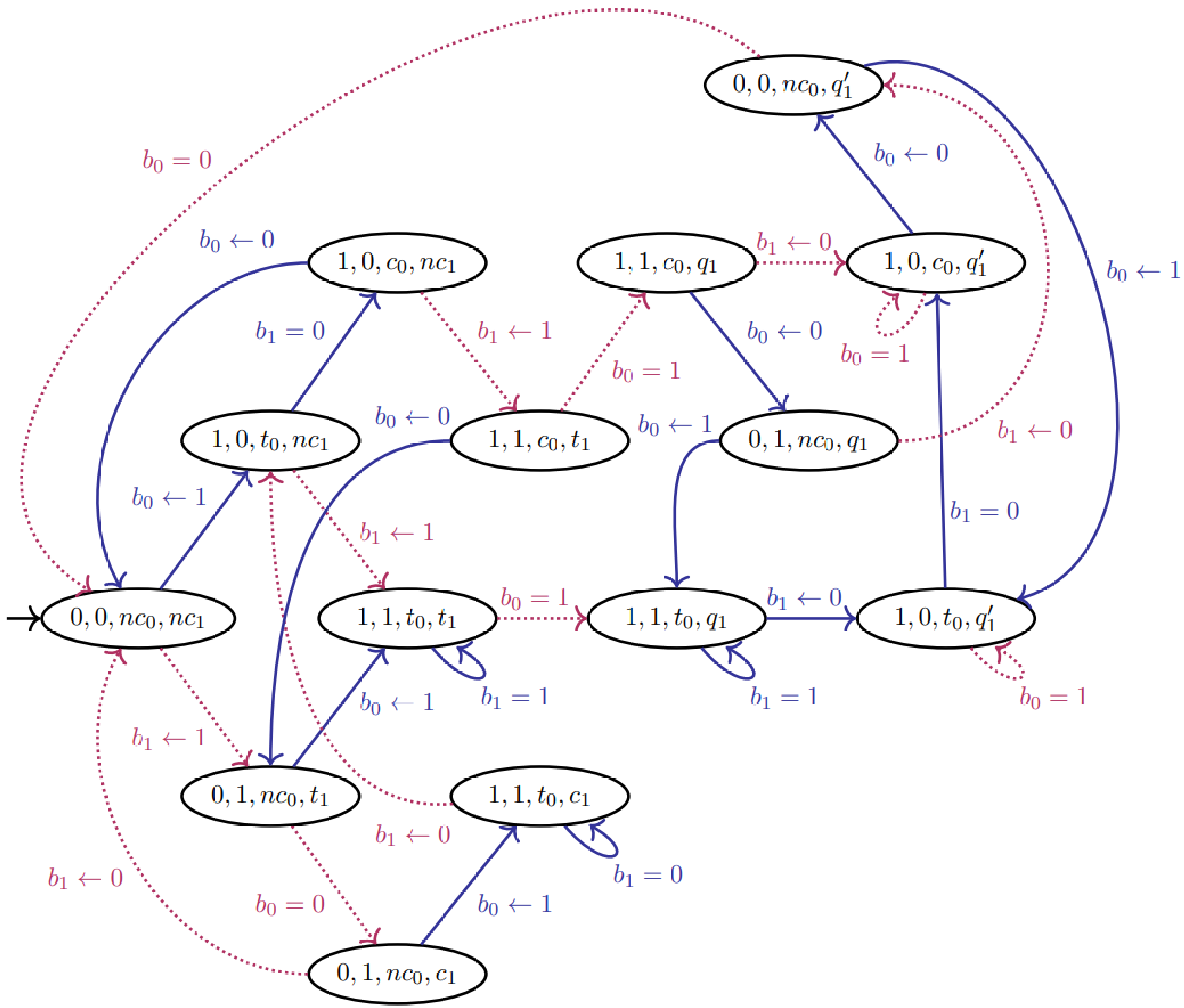
Network for the two-process case







Asynchronous product



Checking properties of the algorithm

- Deadlock freedom: every configuration has at least one successor.

Checking properties of the algorithm

- Deadlock freedom: every configuration has at least one successor.
- Mutual exclusion: no configuration of the form $[b_0, b_1, c_0, c_1]$ is reachable

Checking properties of the algorithm

- Deadlock freedom: every configuration has at least one successor.
- Mutual exclusion: no configuration of the form $[b_0, b_1, c_0, c_1]$ is reachable
- Bounded overtaking (for process 0): after process 0 signals interest in accessing the critical section, process 1 can enter the critical section at most once before process 0 enters.
  - Let NC_i, T_i, C_i be the configurations in which process i is non-critical, trying, or critical
  - Set of potential executions violating the property:

$$\Sigma^* T_0 (\Sigma \setminus C_0)^* C_1 (\Sigma \setminus C_0)^* NC_1 (\Sigma \setminus C_0)^* C_1 \Sigma^*$$

The issue of state explosion

In sequential programs, the number of reachable configurations grows exponentially in the number of variables.

The issue of state explosion

In sequential programs, the number of reachable configurations grows exponentially in the number of variables.

Proposition

The following problem is PSPACE-complete.

Given: a boolean program π (program with only boolean variables), and a NFA A_V recognizing a set of potential executions

Decide: Is $E_\pi \cap A_V$ empty?

The issue of state explosion

In concurrent programs, the number of reachable configurations also grows exponentially in the number of components.

The issue of state explosion

In concurrent programs, the number of reachable configurations also grows exponentially in the number of components.

Proposition

The following problem is PSPACE-complete.

Given: a network of automata $\mathcal{A} = \langle A_1, \ldots, A_n \rangle$ and a NFA A_V recognizing a set of potential executions of $\mathcal{A}$

Decide: Is $L(A_1 \otimes \cdots \otimes A_n \otimes A_V) = \emptyset$?

On-the-fly verification

CheckViol$(A_1, \ldots, A_n, r_V)$

Input: a network $\mathcal{A} = \langle A_1, \ldots A_n \rangle$, where $A_i = (Q_i, \Sigma_i, \delta_i, Q_{0i}, F_i)$
a regular expression r_V over the configurations of $\mathcal{A}$

Output: **true** if $\mathcal{L}(A_1 \otimes \cdots \otimes A_n) \cap \mathcal{L}(r_V)$ is nonempty, **false** otherwise

$(Q_V, \Sigma_V, \delta_V, Q_{0V}, F_V) \leftarrow \mathit{REtoNFA}(r_V)$
$Q \leftarrow \emptyset$; $Q_0 \leftarrow Q_{01} \times \cdots \times Q_{0n} \times Q_{0V}$
$W \leftarrow Q_0$
while $W \neq \emptyset$ **do**
  pick $[q_1, \ldots, q_n, q]$ **from** W
  add $[q_1, \ldots, q_n, q]$ **to** Q
  for all $a \in \Sigma_1 \cup \ldots \cup \Sigma_n$ **do**
    for all $i \in [1..n]$ **do**
      if $a \in \Sigma_i$ **then** $Q'_i \leftarrow \delta_i(q_i, a)$ **else** $Q'_i = \{q_i\}$
    for all $[q'_1, \ldots, q'_n] \in Q'_1 \times \ldots \times Q'_n$ **do**
      $Q' \leftarrow \delta_V(q, [q'_1, \ldots, q'_n])$
      for all $q' \in Q'$ **do**
        if $\bigwedge_{i=1}^{n} q'_i \in F_i$ **and** $q' \in F_V$ **then return true**
        if $[q'_1, \ldots, q'_n, q'] \notin Q$ **then add** $[q'_1, \ldots, q'_n, q']$ **to** W
return false

Compositional verification

To check emptiness of an asynchronous product
$A_1 \otimes \cdots \otimes A_n$ we can

Compositional verification

To check emptiness of an asynchronous product $A_1 \otimes \cdots \otimes A_n$ we can

- Replace A_1 by an automaton A'_1 recognizing $\mathrm{proj}_{\Sigma \setminus \Sigma_1}(L(A_1))$ and compute $A_{12} = A'_1 \otimes A_2$;
- Replace A_{12} by an automaton A'_{12} recognizing $\mathrm{proj}_{\Sigma \setminus (\Sigma_1 \cup \Sigma_2)}(L(A_{12}))$ and compute $A_{13} = A'_{12} \otimes A_3$;
- $\cdots$
- Replace $A_{1(n-1)}$ by an automaton $A'_{1(n-1)}$ recognizing $\mathrm{proj}_{\Sigma \setminus (\Sigma_1 \cup \cdots \cup \Sigma_{n-1})}(L(A_{1(n-1)}))$ and compute $A_{1n} = A'_{1(n-1)} \otimes A_n$.

Compositional verification

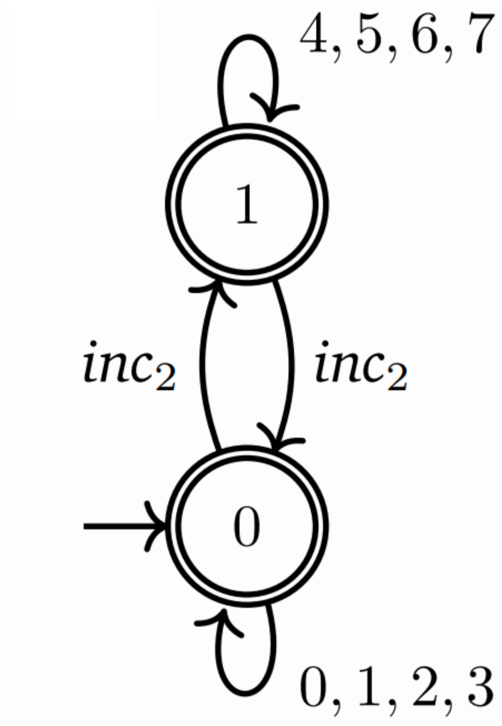
To check emptiness of an asynchronous product $A_1 \otimes \cdots \otimes A_n$ we can

- Replace A_1 by an automaton A_1' recognizing $\mathrm{proj}_{\Sigma \setminus \Sigma_1}(L(A_1))$ and compute $A_{12} = A_1' \otimes A_2$;
- Replace A_{12} by an automaton A_{12}' recognizing $\mathrm{proj}_{\Sigma \setminus (\Sigma_1 \cup \Sigma_2)}(L(A_{12}))$ and compute $A_{13} = A_{12}' \otimes A_3$;
- $\cdots$
- Replace $A_{1(n-1)}$ by an automaton $A_{1(n-1)}'$ recognizing $\mathrm{proj}_{\Sigma \setminus (\Sigma_1 \cup \cdots \cup \Sigma_{n-1})}(L(A_{1(n-1)}))$ and compute $A_{1n} = A_{1(n-1)}' \otimes A_n$.

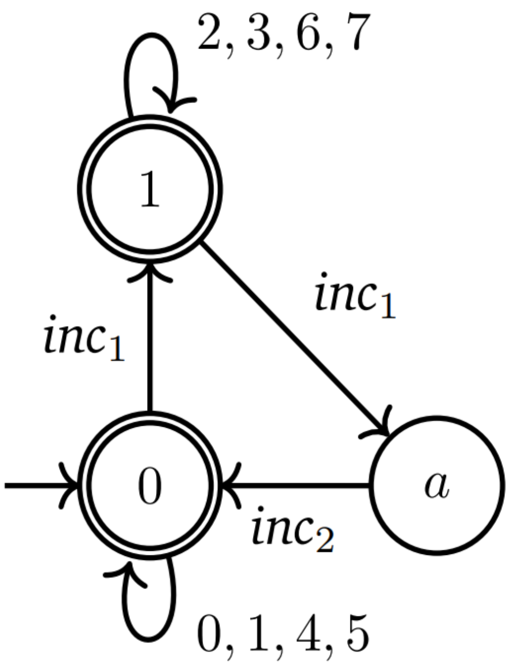
This can save space w.r.t. the direct computation.

Compositional verification

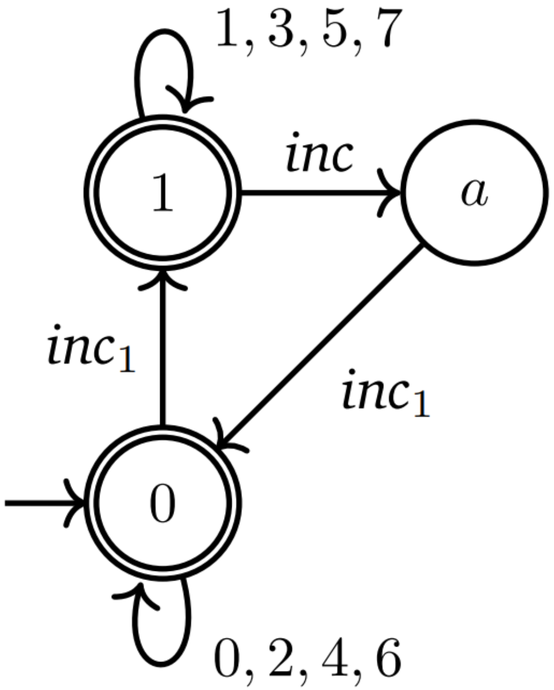
A_2



A_1

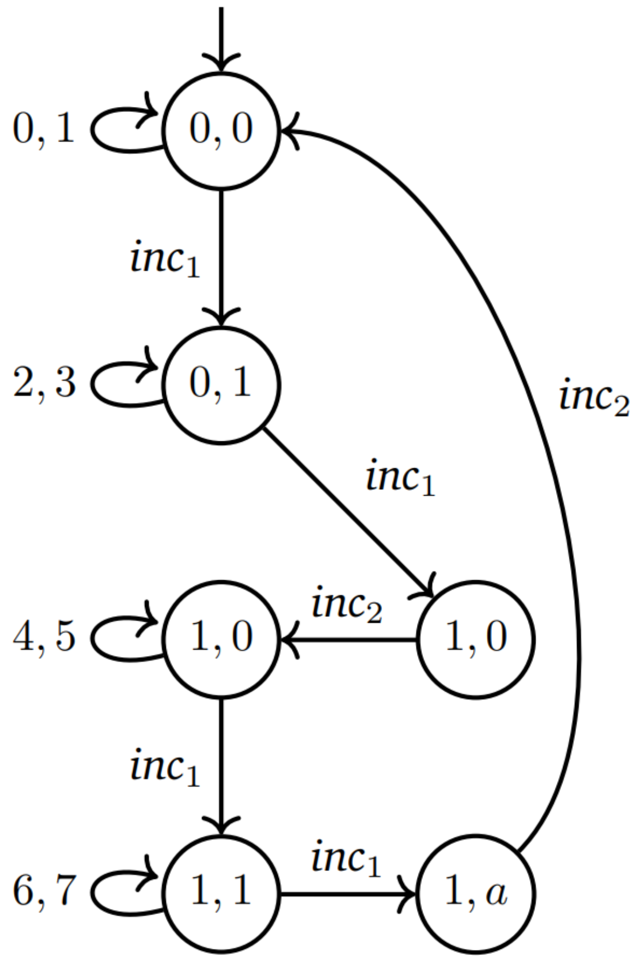


A_0

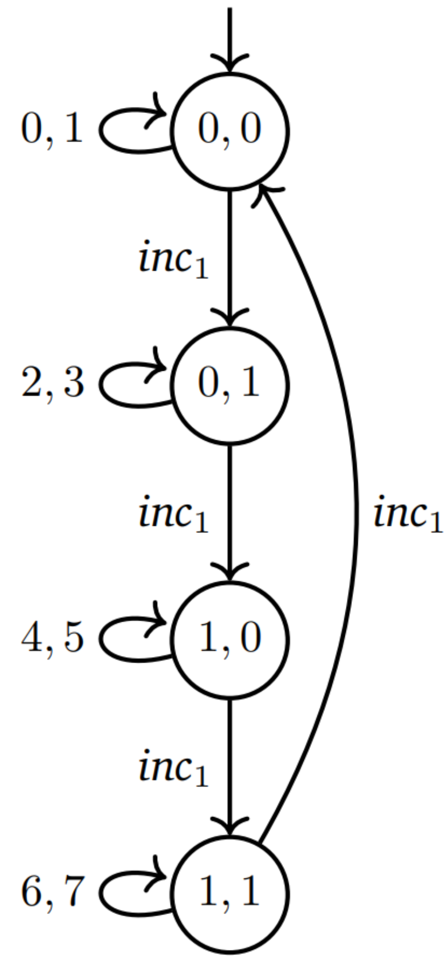


Compositional verification

A_{21}

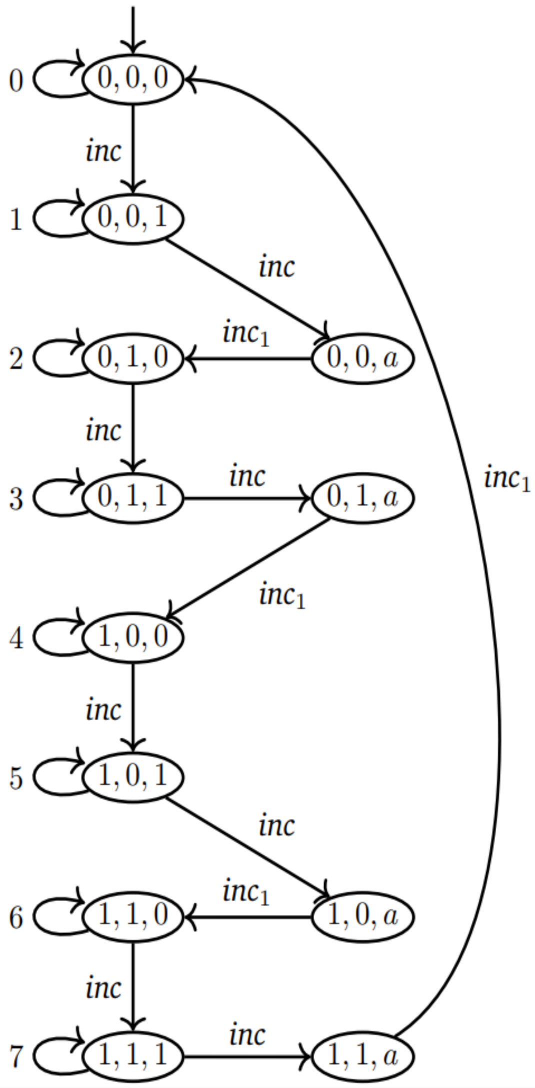


A'_{21}

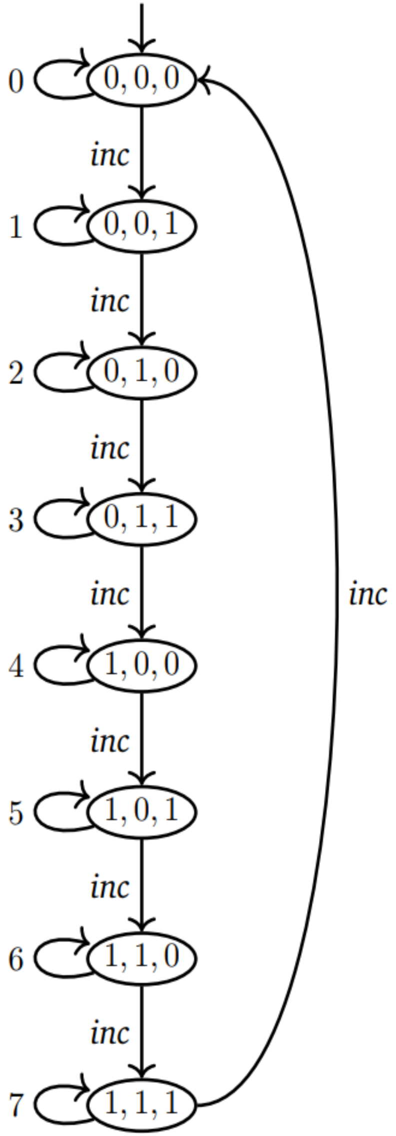


Compositional verification

A_{21}



A'_{21} (proj. on visible actions)

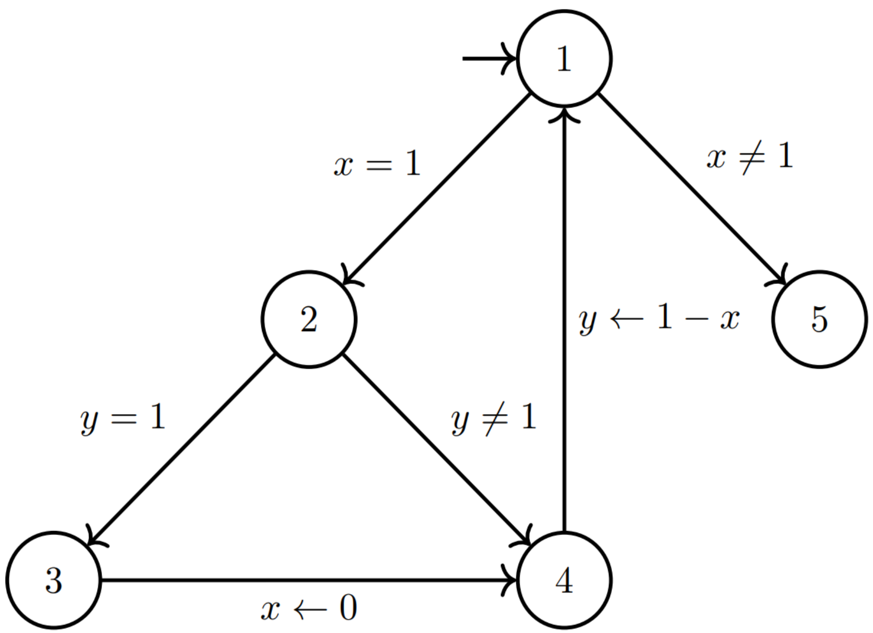


Symbolic exploration

- A technique to mitigate the issue of state explosion
- Configurations can be encoded as words.
- The set of reachable configurations of a program can be encoded as a language.
- We use automata to compactly store the set of reachable configurations.

Flowgraphs

```
while x = 1 do
   if y = 1 then
      x ← 0
   y ← 1 − x
end
```



Step relations

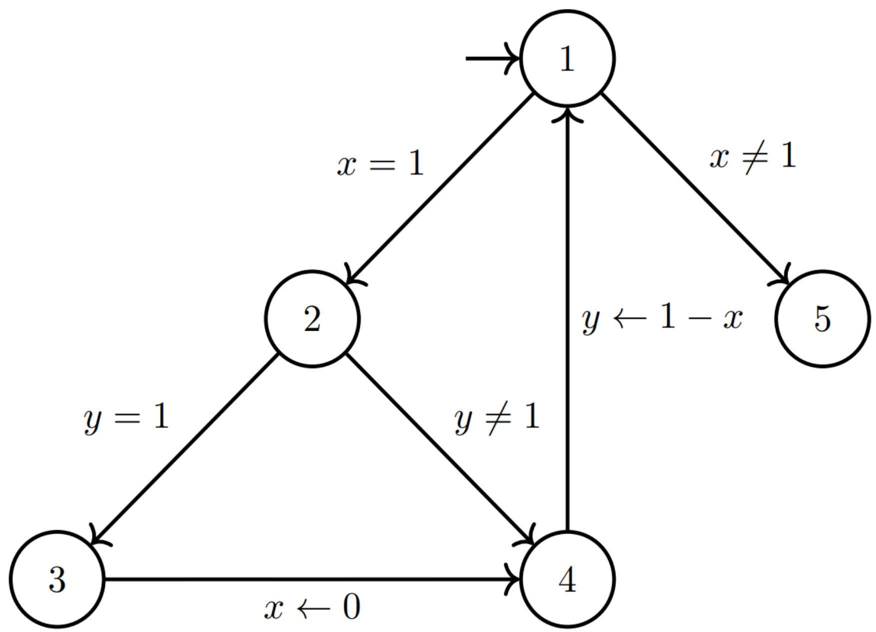
Let ℓ, ℓ' be two control points of a flowgraph.

The step relation $S_{\ell,\ell'}$ contains all pairs

$$([\ell, x_0, y_0], [\ell', x_0', y_0'])$$

of configurations such that: if at point ℓ the current values of x, y are x_0, y_0, then the program can take a step, after which the new control point is ℓ' and the new values of x, y are x_0', y_0'.

Step relations



$$S_{\ell,\ell'} = \{([4, x_0, y_0], [1, x_0, 1 - x_0]) \mid x_0, y_0 \in \{0, 1\}\}$$

The global step relation S is the union of the step relations $S_{\ell,\ell'}$ for all pairs ℓ, ℓ' of control points.

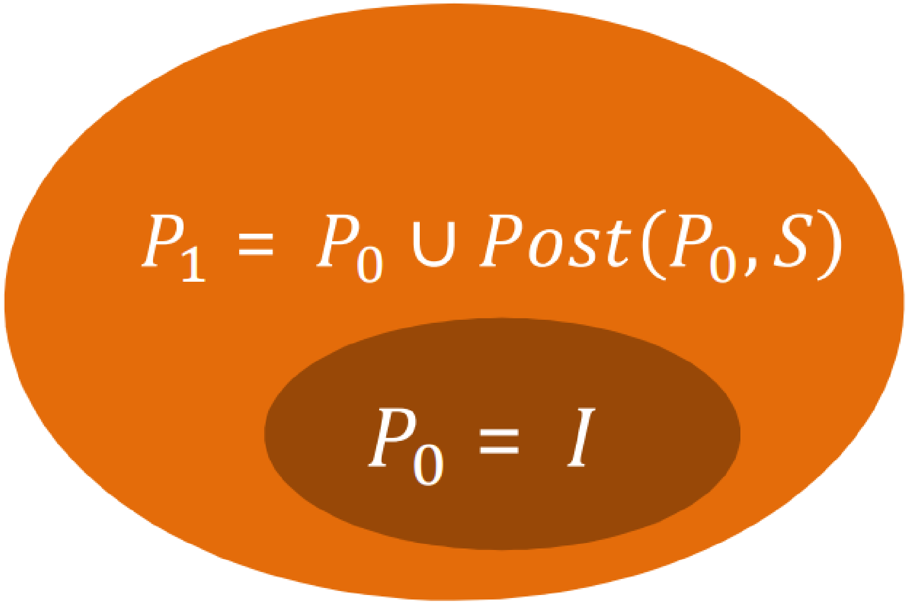
Computing reachable configurations

- Start with the set of initial configurations
- Iteratively: add the set of successors of the current set of configurations until a fixed point is reached

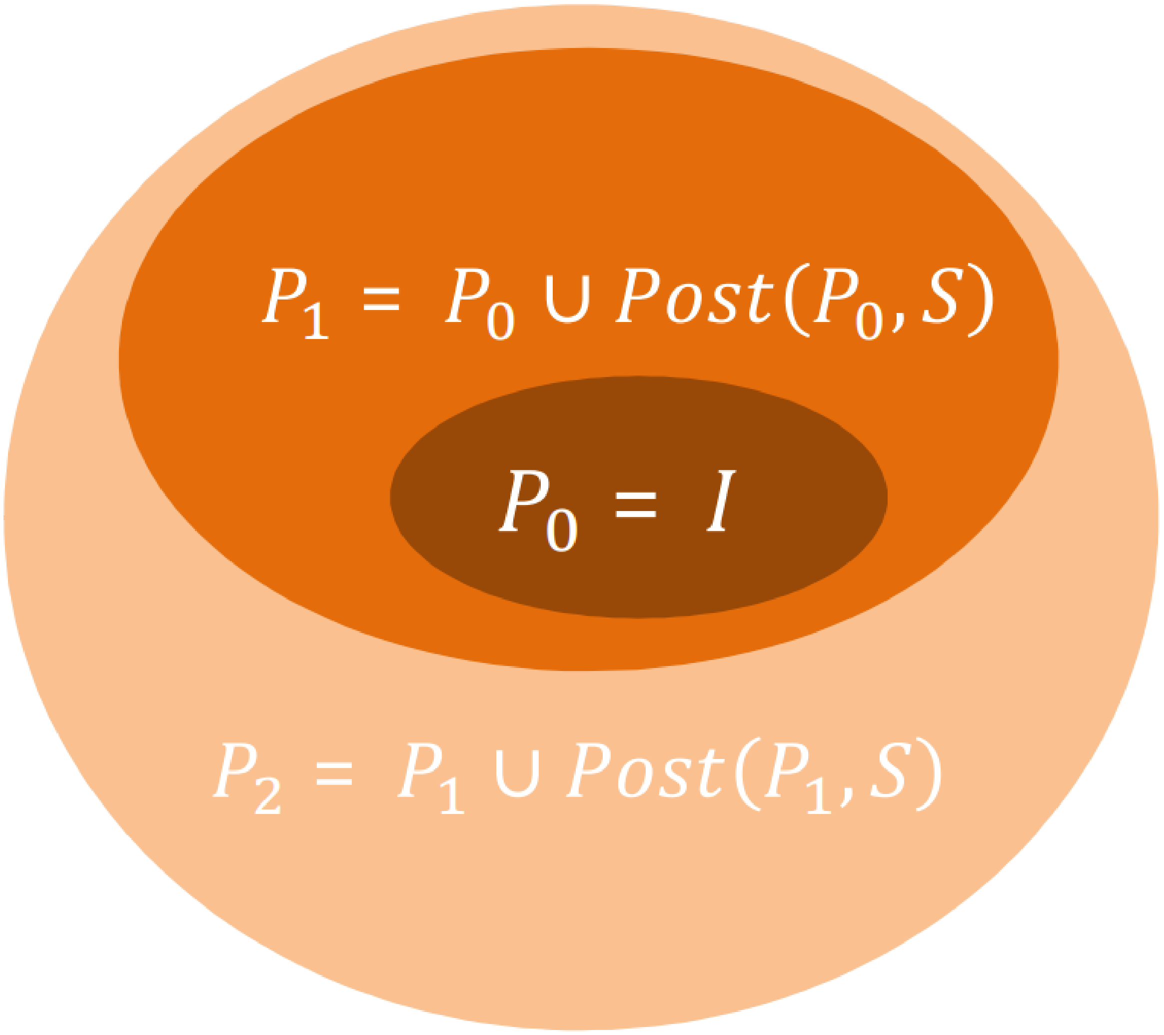
Computing reachable configurations

$P_0 = I$

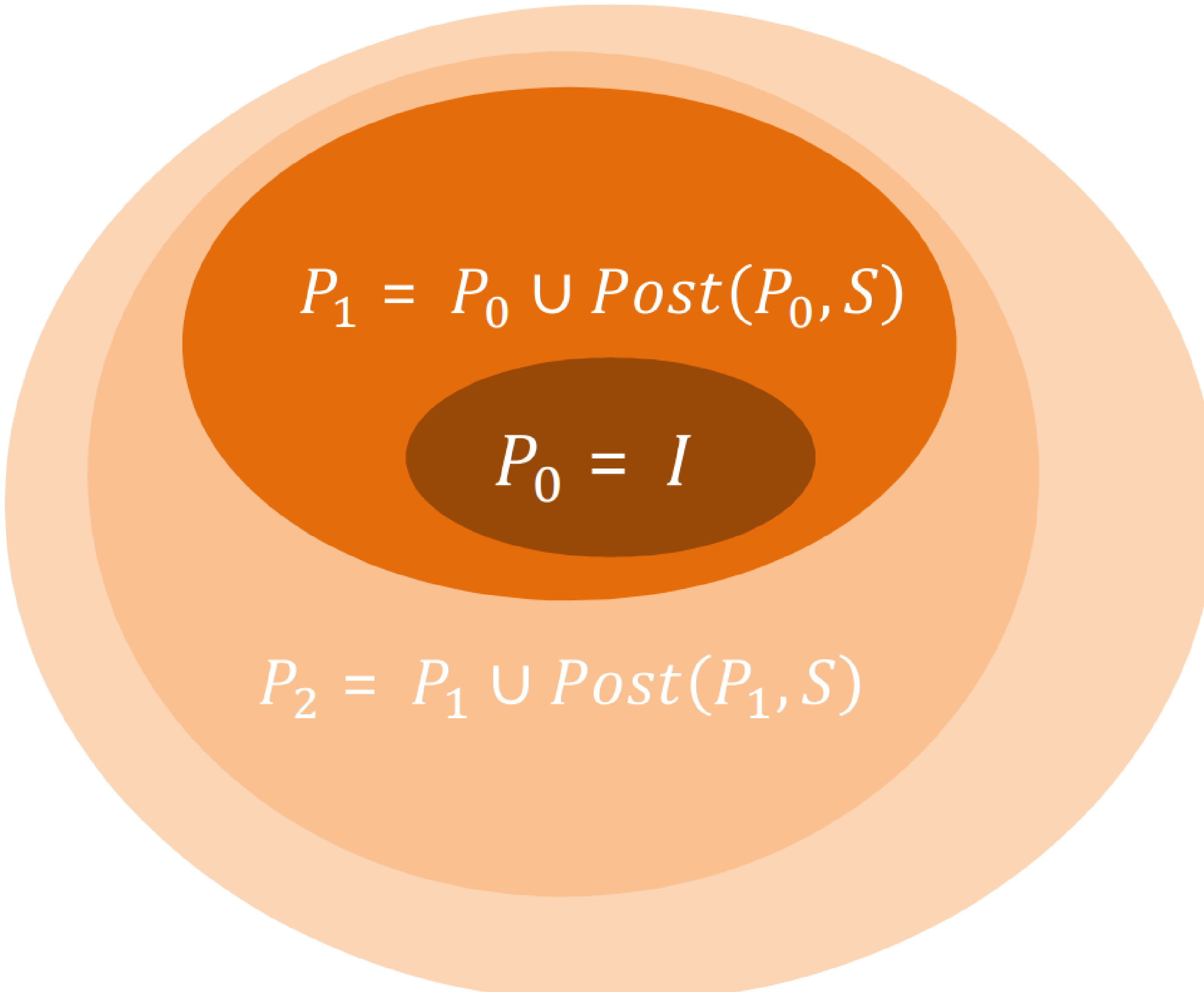
Computing reachable configurations



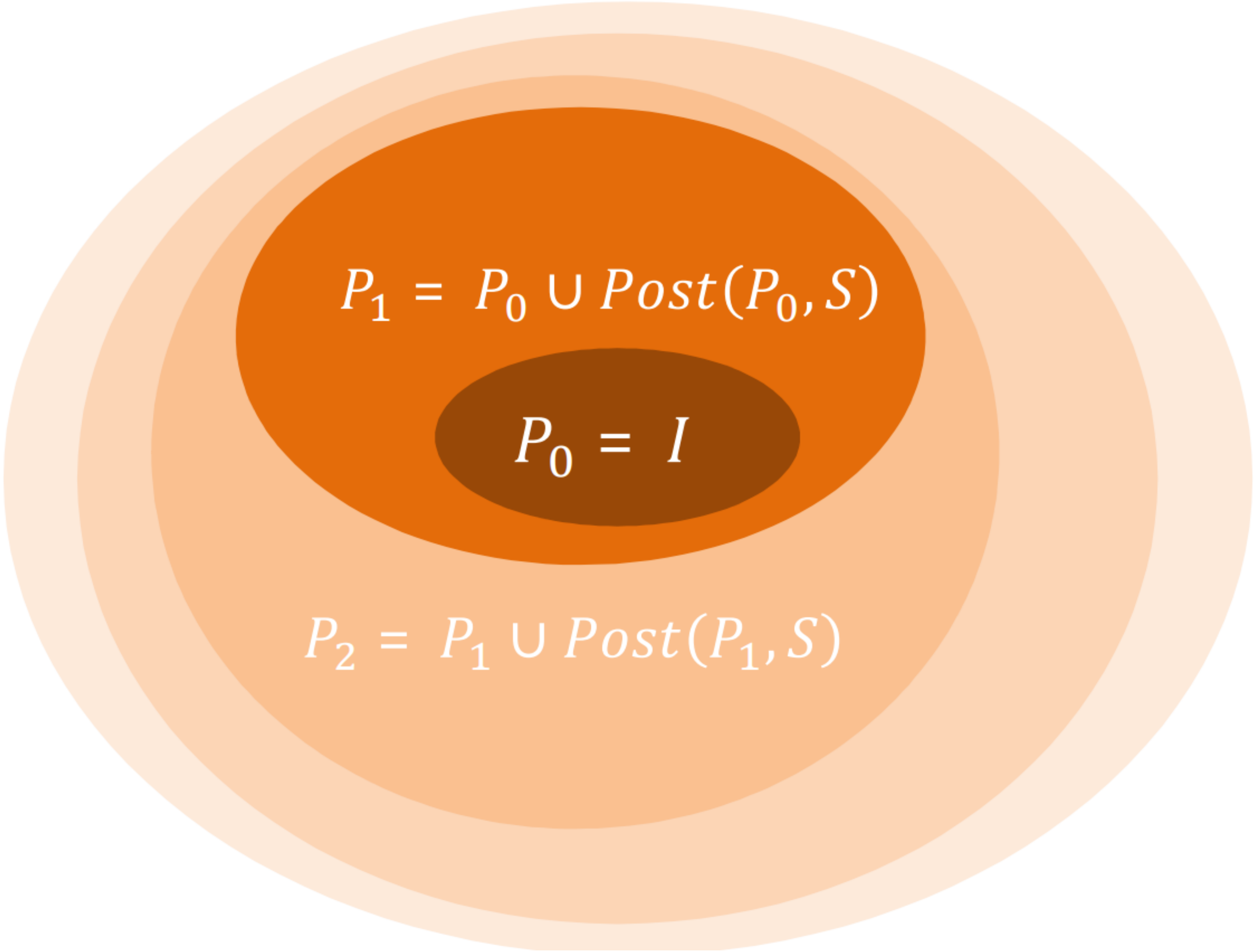
Computing reachable configurations



Computing reachable configurations



Computing reachable configurations

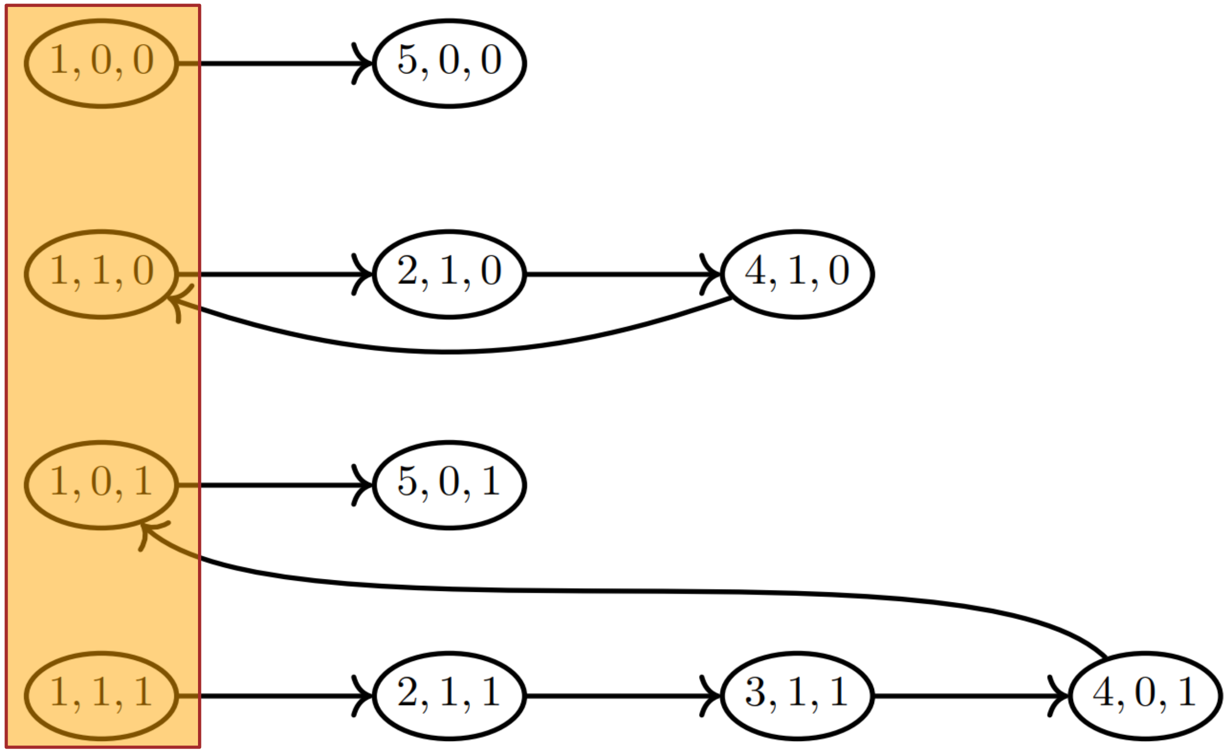


Computing reachable configurations

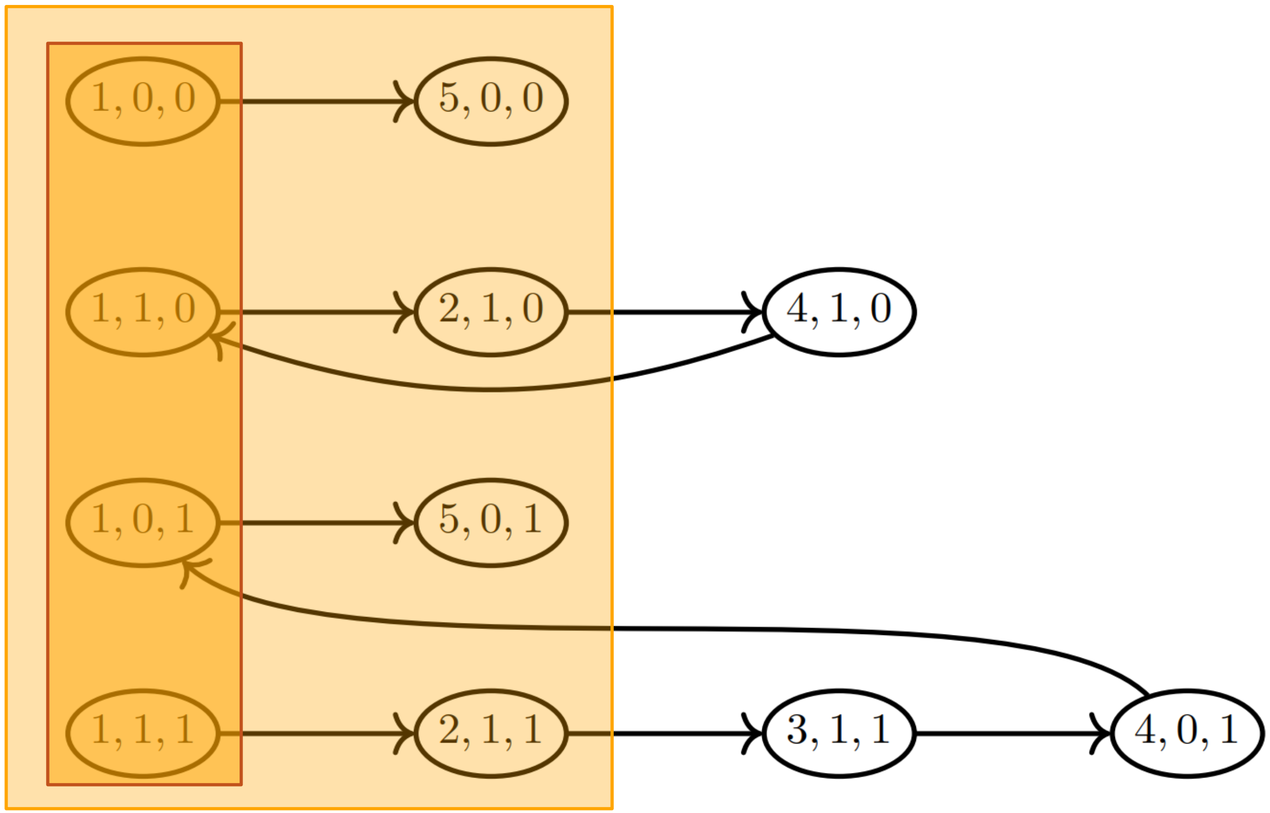
Reach(I, R)
Input: set I of initial configurations; step relation S
Output: set of configurations reachable from I

```
OldP ← ∅; P ← I
while P ≠ OldP do
    OldP ← P
    P ← Union(P, Post(P, S))
return P
```

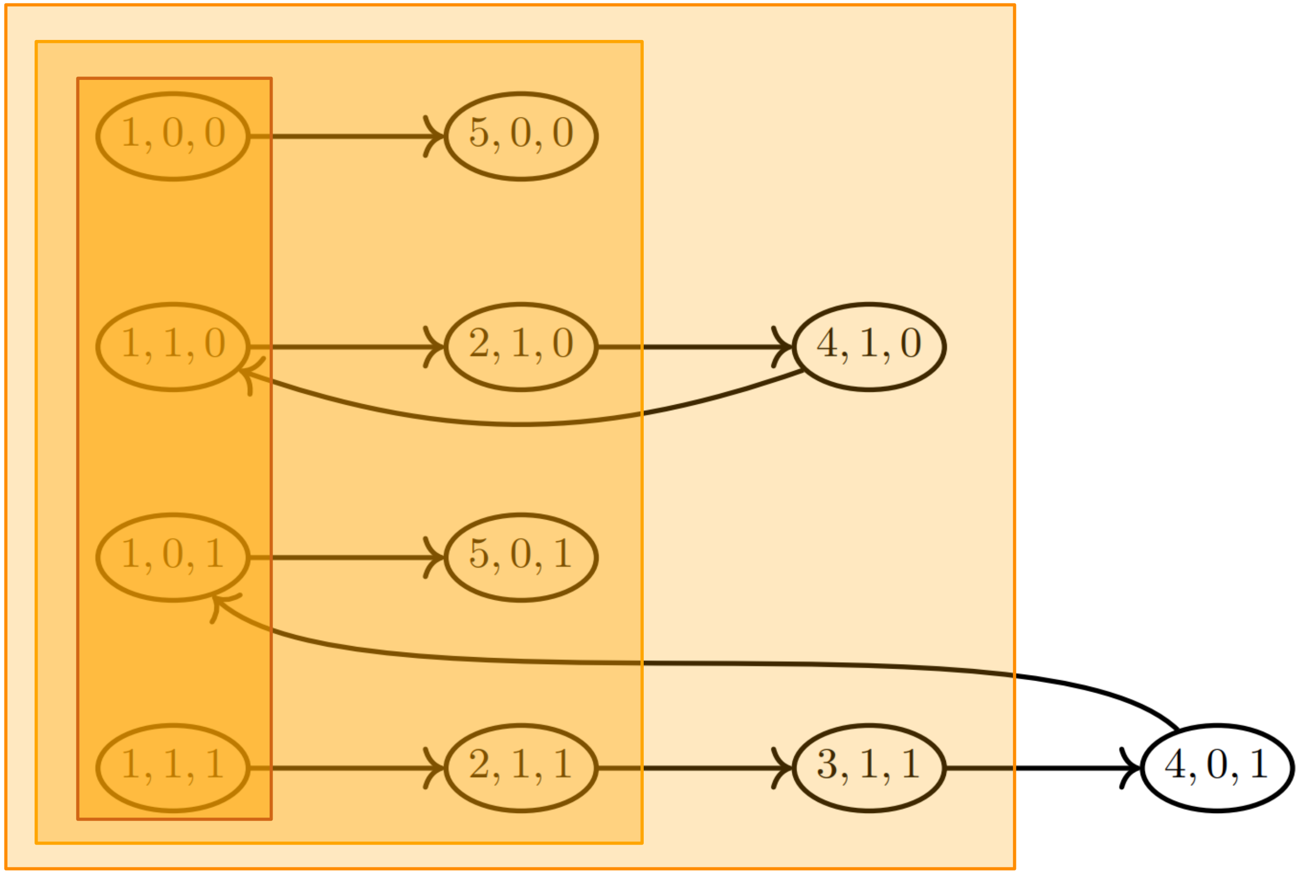
Computing reachable configurations



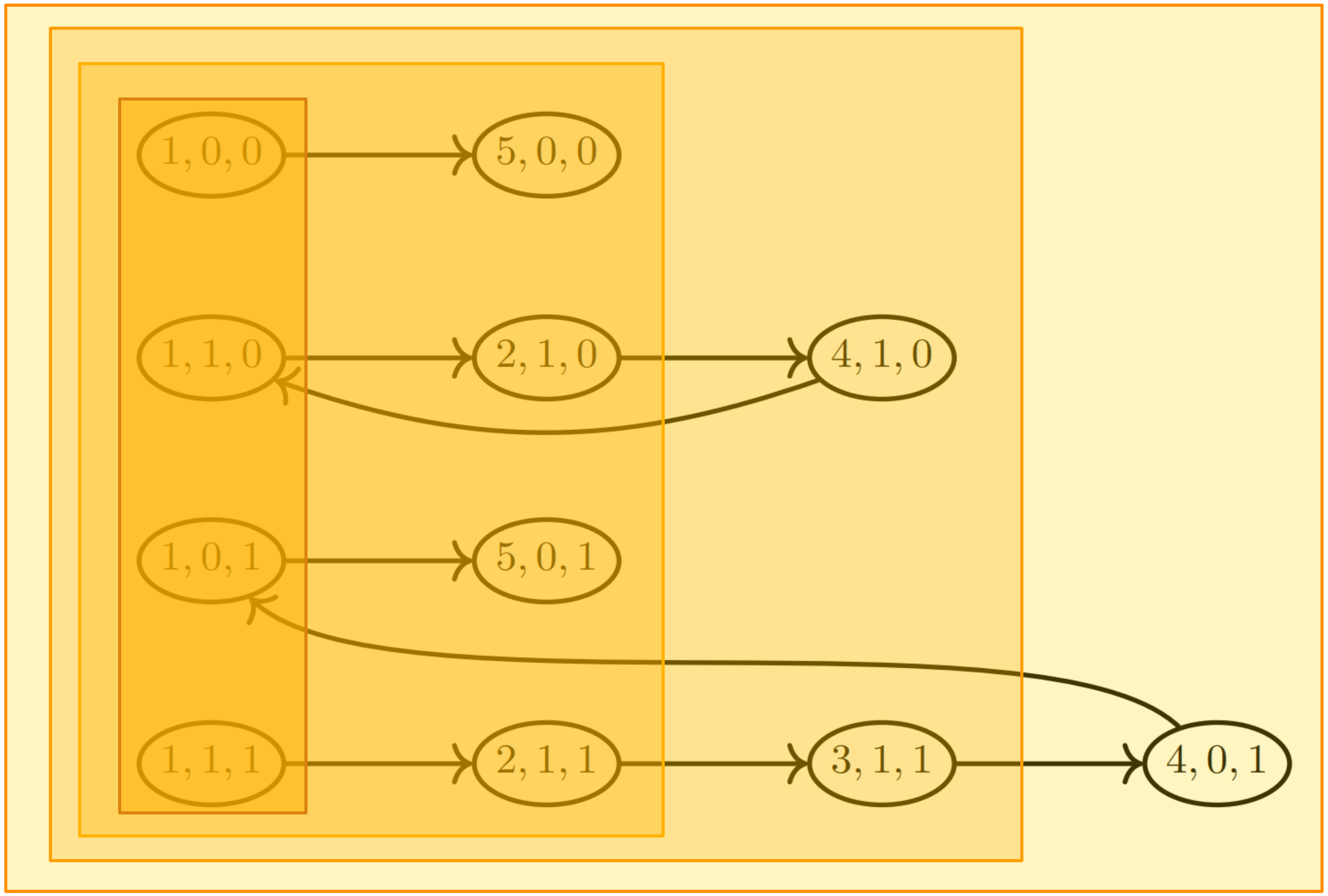
Computing reachable configurations



Computing reachable configurations

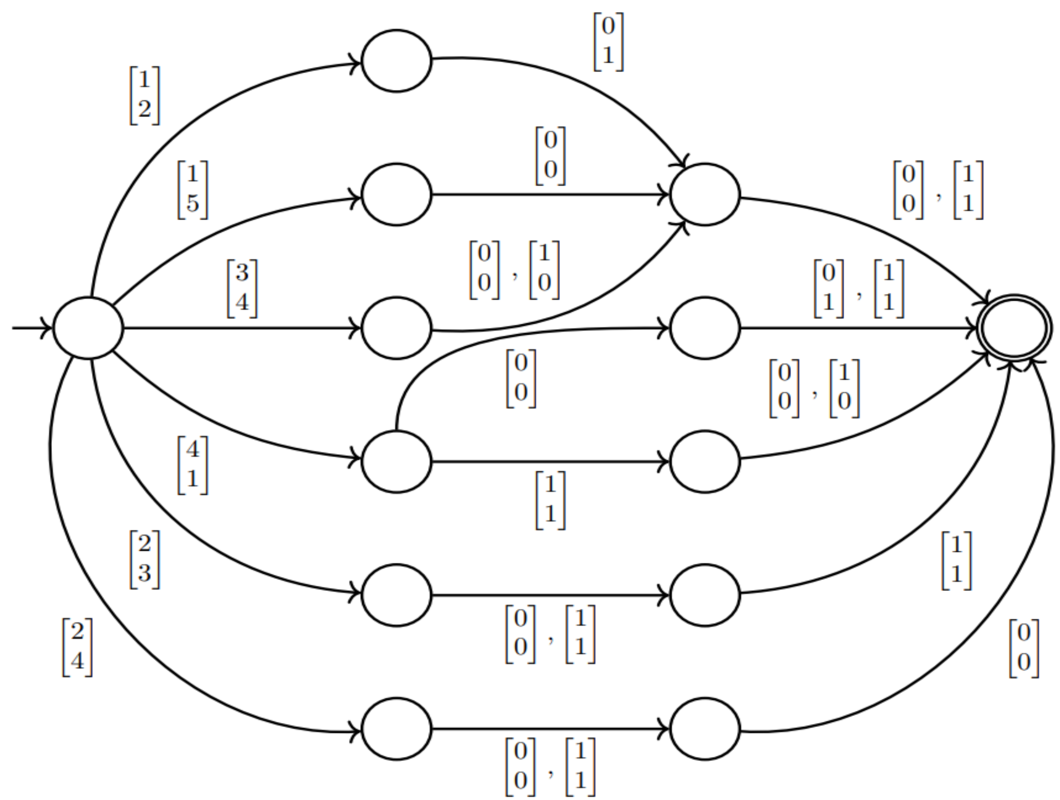


Computing reachable configurations

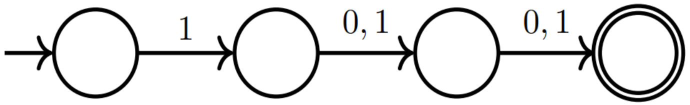


Example: Transducer for the global step relation

while $x = 1$ ***do***
  if $y = 1$ ***then***
    $x \leftarrow 0$
  $y \leftarrow 1 - x$
end

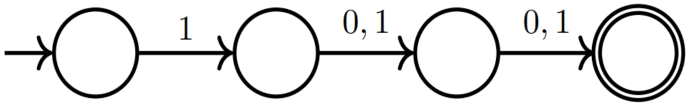


Example: DFAs generated by Reach

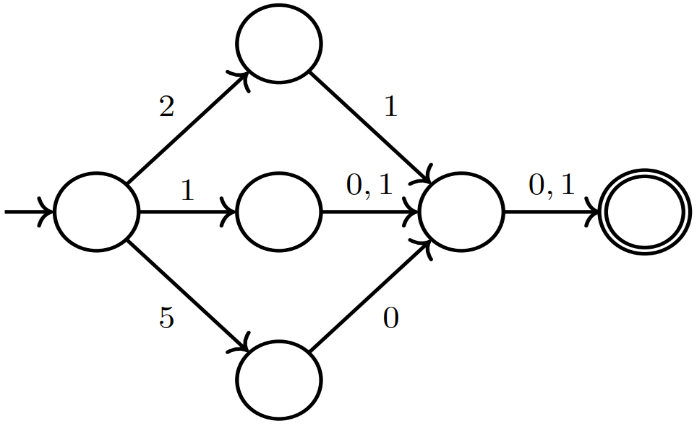


Initial configurations

Example: DFAs generated by Reach

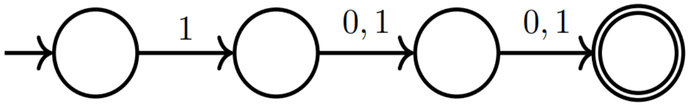


Initial configurations

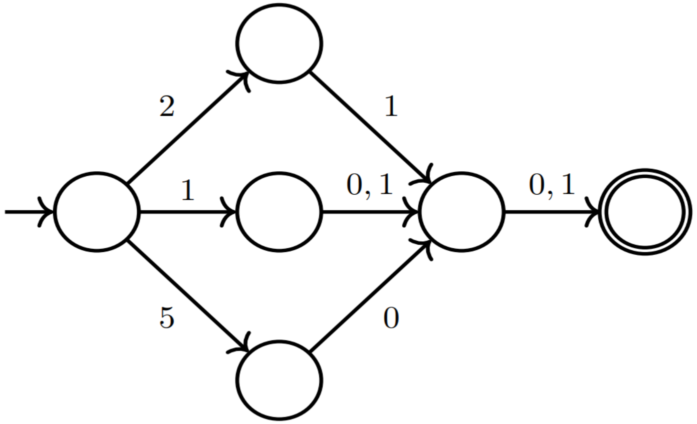


Configurations reachable in at most 1 step

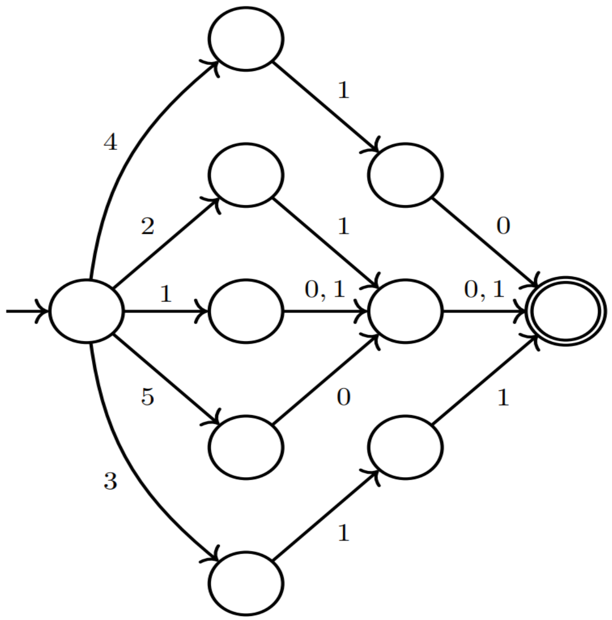
Example: DFAs generated by Reach



Initial configurations

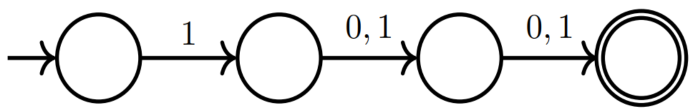


Configurations reachable in at most 1 step

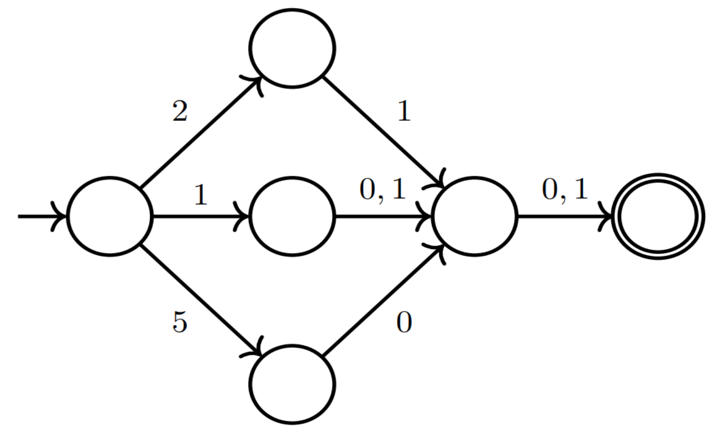


Configurations reachable in at most 2 steps

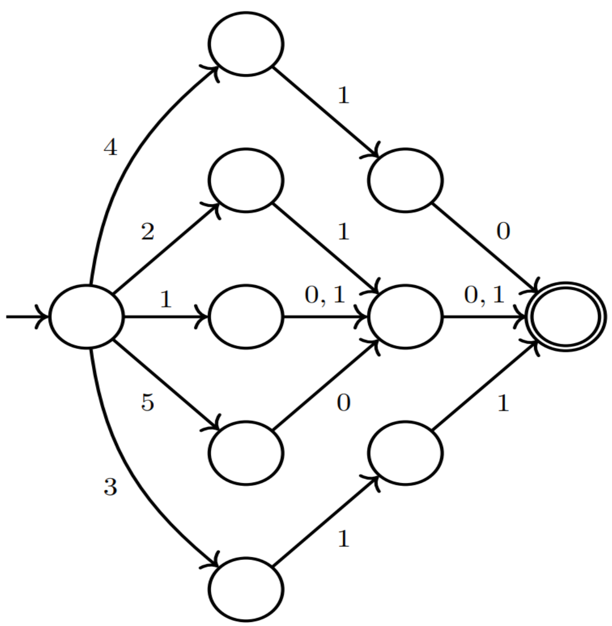
Example: DFAs generated by Reach



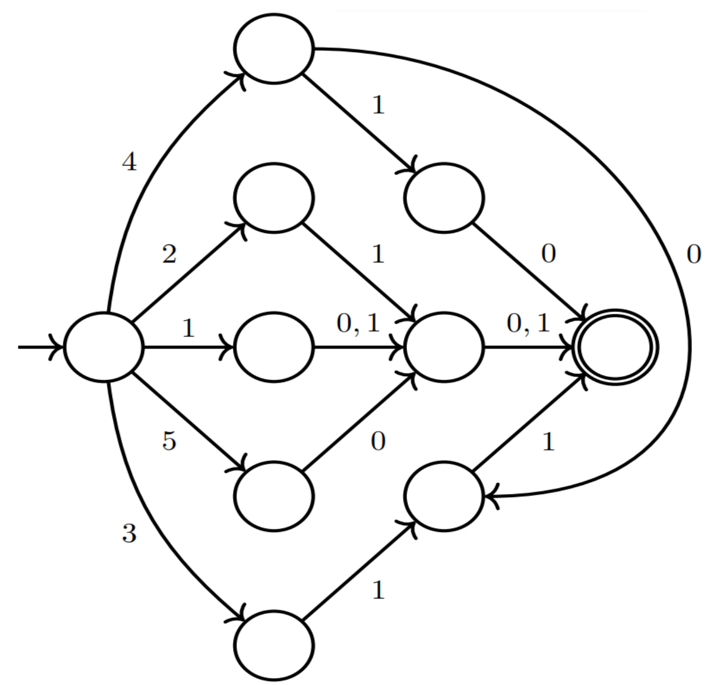
Initial configurations



Configurations reachable in at most 1 step



Configurations reachable in at most 2 steps



Configurations reachable in at most 3 steps

Variable orders

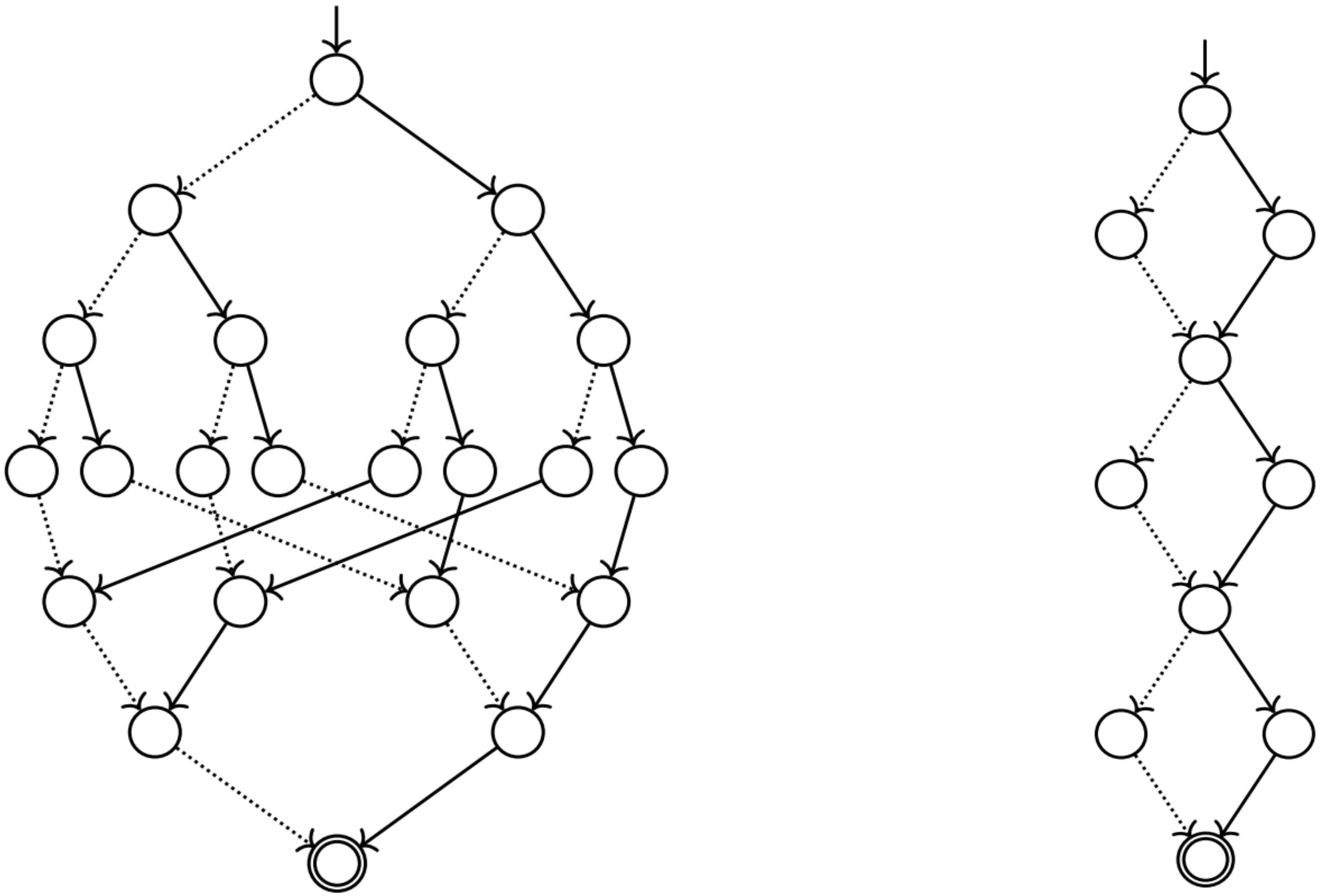
- Consider the set Y of tuples $[x_1, \ldots, x_{2k}]$ of booleans such that $x_1 = x_{k+1}, x_2 = x_{k+2}, \ldots, x_k = x_{2k}$

- A tuple $[x_1, \ldots, x_{2k}]$ can be encoded by the word $x_1 x_2 \cdots x_{2k-1} x_{2k}$ but also by the word $x_1 x_{k+1} \cdots x_k x_{2k}$

- For $k = 3$, the encodings of Y are then, respectively $\{000000, 001001, 010010, 011011, 100100, 101101, 110110, 111111\}$, $\{000000, 000011, 001100, 001111, 110000, 110011, 111100, 111111\}$.

- The minimal DFAs for these languages have very different sizes!

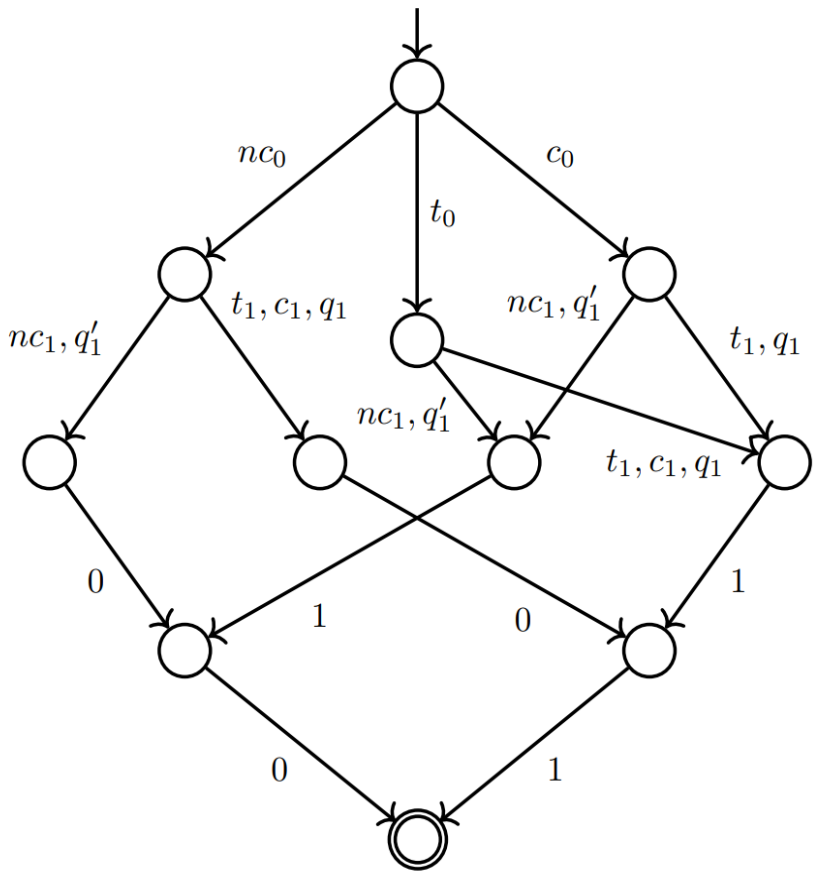
Minimal DFAs for Reachable Configurations



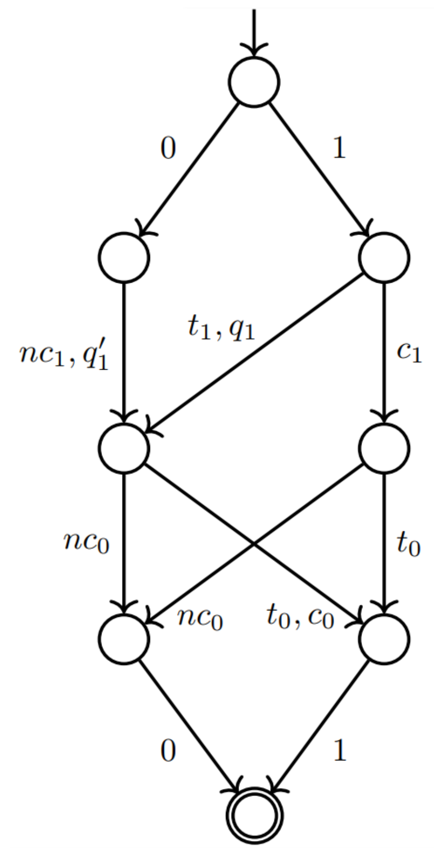
0 and 1 are respectively represented by solid and dotted arcs

Minimal DFAs for Reachable Configurations

Another example: Lamport's algorithm



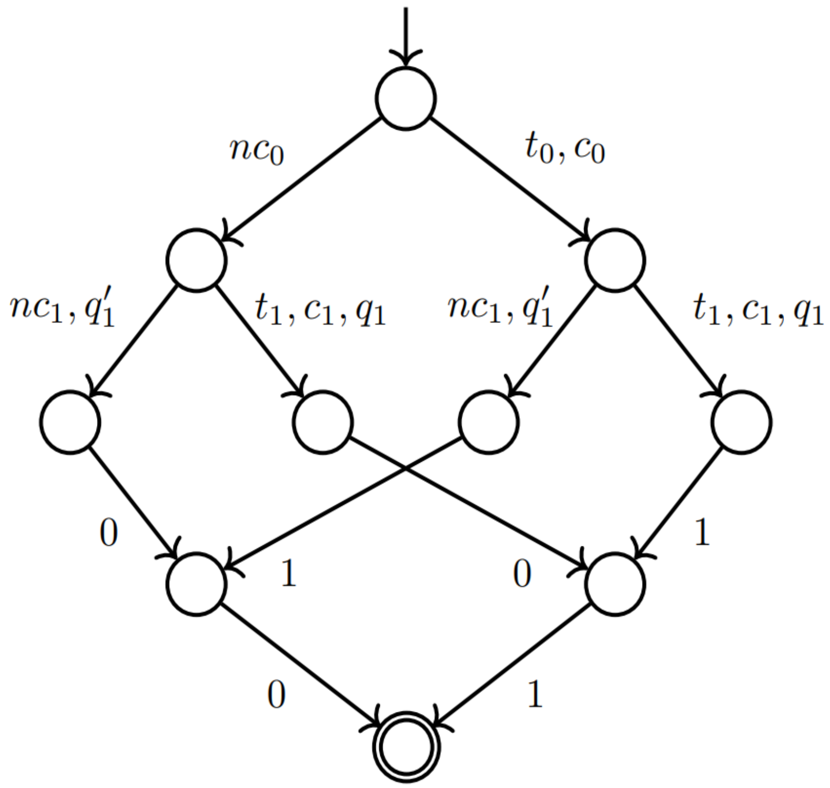
$\langle v_0, v_1, s_0, s_1 \rangle$ ensoded by $s_0 s_1 v_0 v_1$

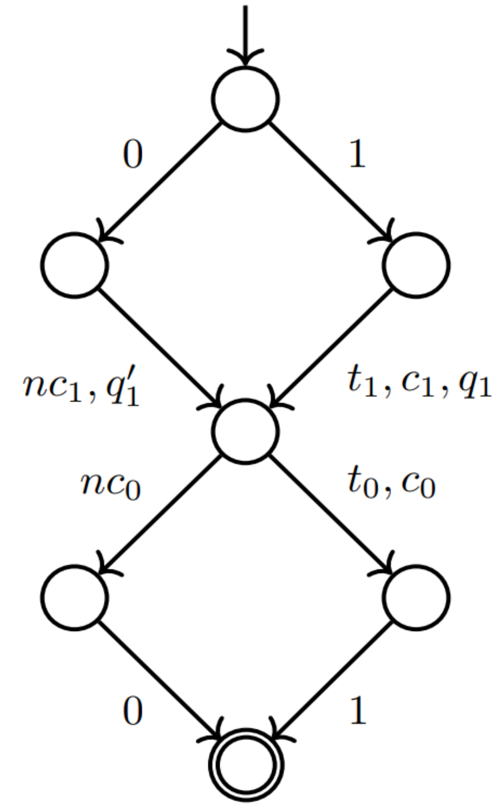


$\langle v_0, v_1, s_0, s_1 \rangle$ ensoded by $v_1 s_1 s_0 v_0$

Minimal DFAs for Reachable Configurations

Larger sets can yield smaller DFAs!





DFAs after adding the configuration $\langle c_0, c_1, 1, 1\rangle$ to the set

Reachable Configurations

- When encoding configurations, good variable orders can lead to much smaller automata.

- Unfortunately, the problem of finding an optimal encoding for a language represented by a DFA is NP-complete