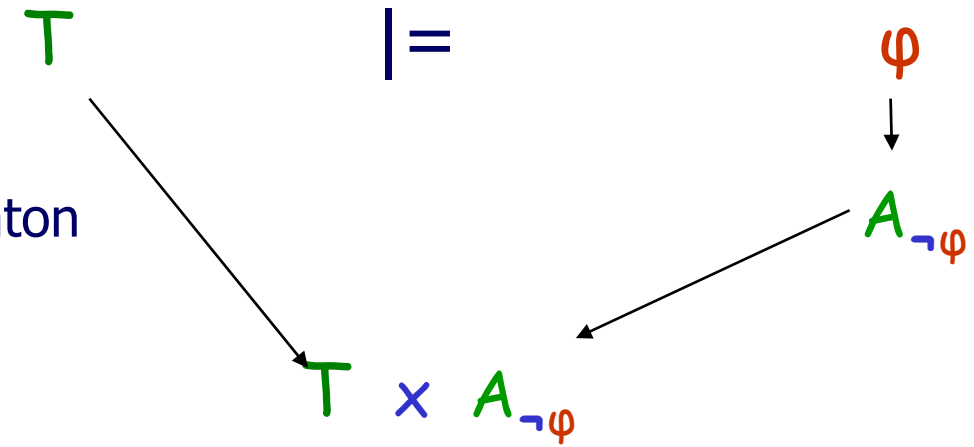
Lecture 7

Verification of Linear-Time Properties

Model checking Linear Temporal logic

Automata-Theoretic Approach [Vardi Wolper 1986]

- Question



- Construct

  Buchi Automaton
- Combine

- Find

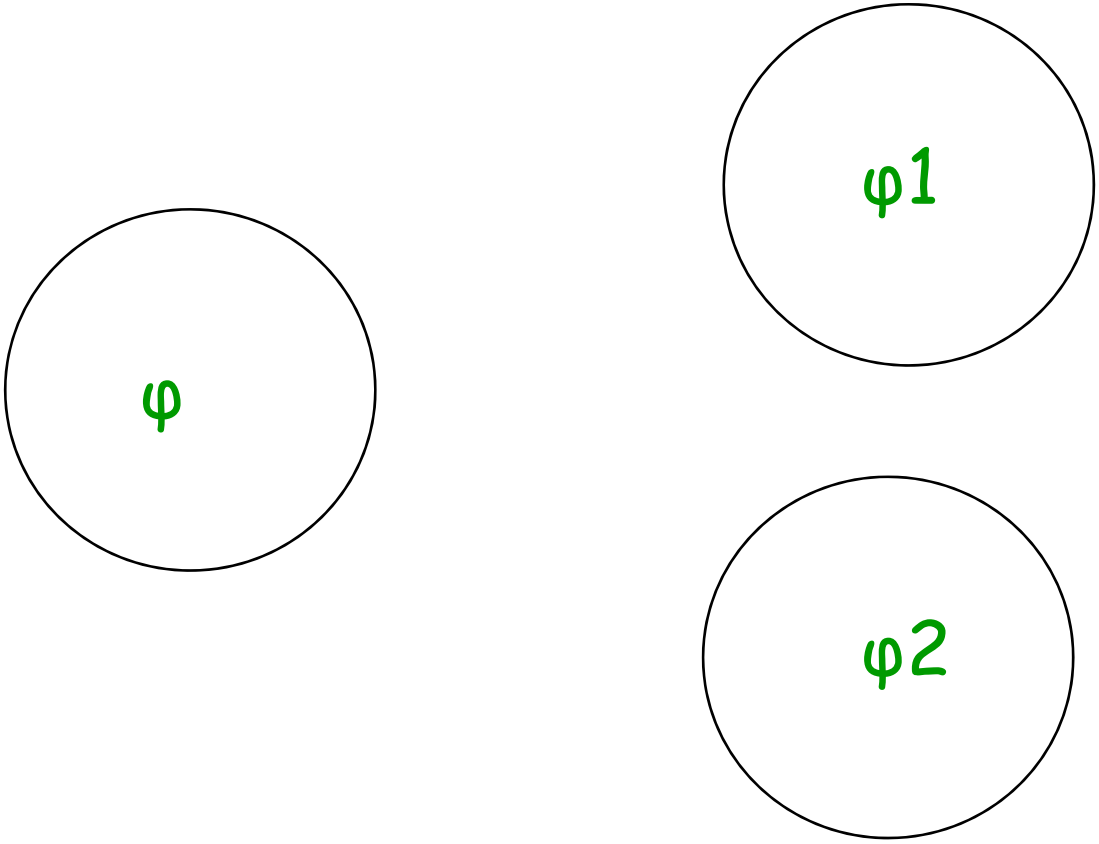
  accepting computation (error trace)

Translating formulas to Automata

$\Diamond(p \wedge \Box \neg q)$

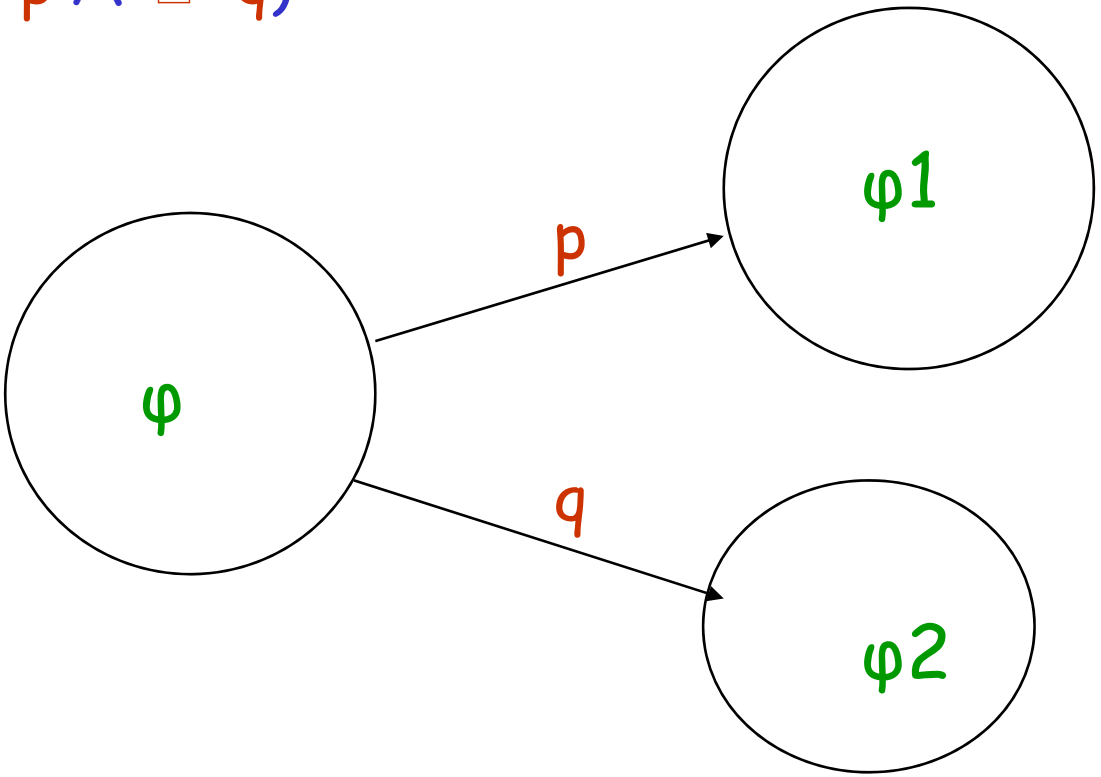
Translating formulas to Automata

◊(p ∧ □¬q)



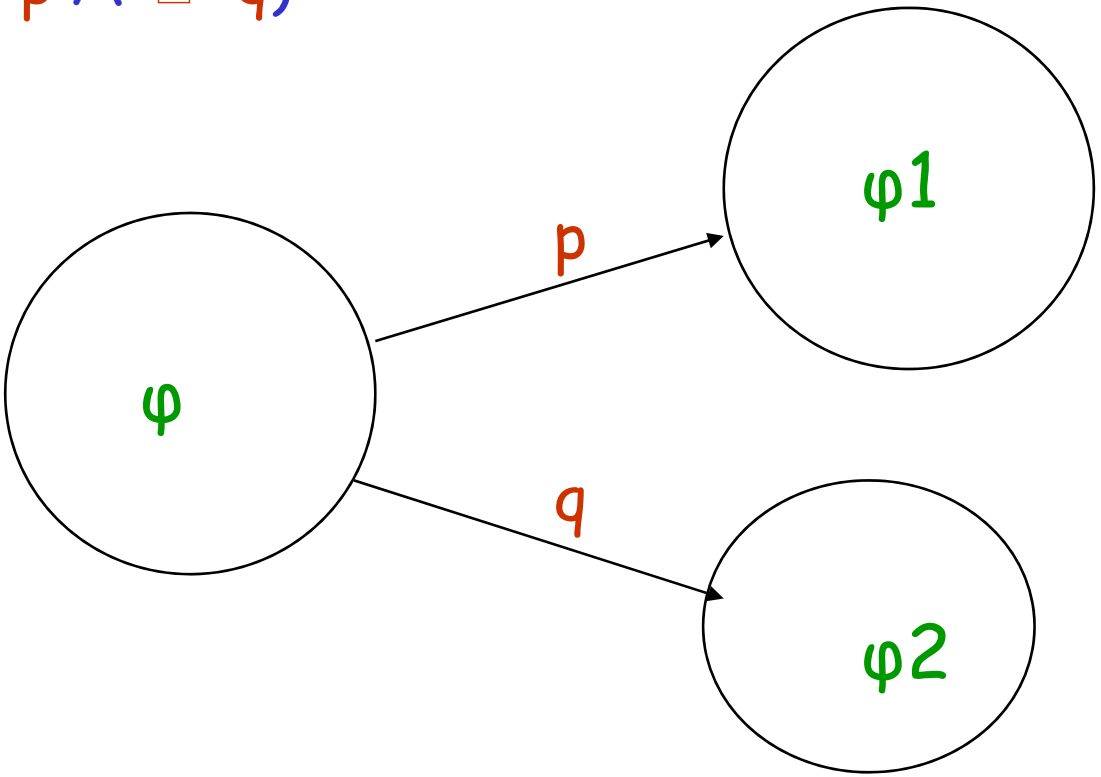
Translating formulas to Automata

◊(p ∧ □¬q)



Translating formulas to Automata

$\Diamond(p \wedge \Box \neg q)$



$\varphi \equiv p \wedge \circ\varphi 1 \quad \vee \quad q \wedge \circ\varphi 2$

Decomposing temporal operators

Pushing negations

$\neg \Box \varphi \equiv \Diamond \neg \varphi$

$\neg \Diamond \varphi \equiv \Box \neg \varphi$

$\neg (\varphi \, U \, \psi) \equiv \neg\psi \, W \, (\neg\varphi \wedge \neg\psi)$

$\neg (\varphi \, W \, \psi) \equiv \neg\psi \, U \, (\neg\varphi \wedge \neg\psi)$

Putting on Normal form

$\Box \varphi \equiv \varphi \wedge \circ \Box \varphi$

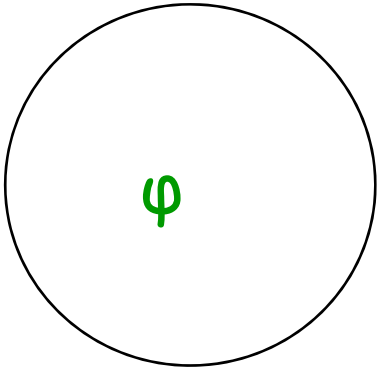
$\Diamond \varphi \equiv \varphi \vee \circ \Diamond \varphi$

$\varphi \, U \, \psi \equiv \psi \vee (\varphi \wedge \circ \varphi \, U \, \psi)$

$\varphi \, W \, \psi \equiv \psi \vee (\varphi \wedge \circ \varphi \, W \, \psi)$

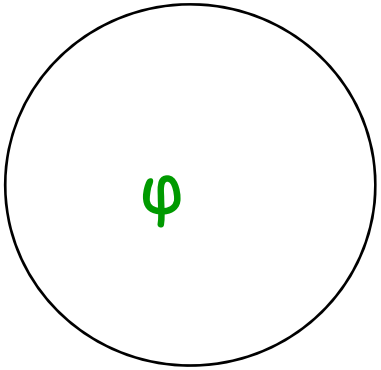
Translating formulas to Automata

$\varphi \equiv \Diamond(p \wedge \Box \neg q)$



Translating formulas to Automata

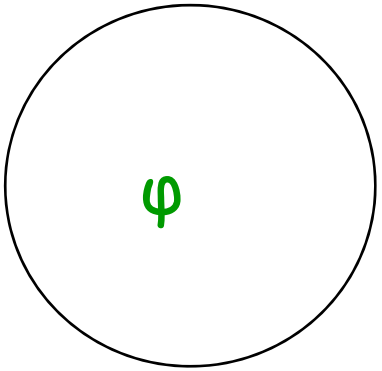
$\varphi \equiv \Diamond(p \wedge \Box \neg q) \equiv (p \wedge \Box \neg q) \vee \circ \Diamond(p \wedge \Box \neg q)$



Translating formulas to Automata

$$\varphi \equiv \Diamond(p \wedge \Box\neg q) \equiv (p \wedge \Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$

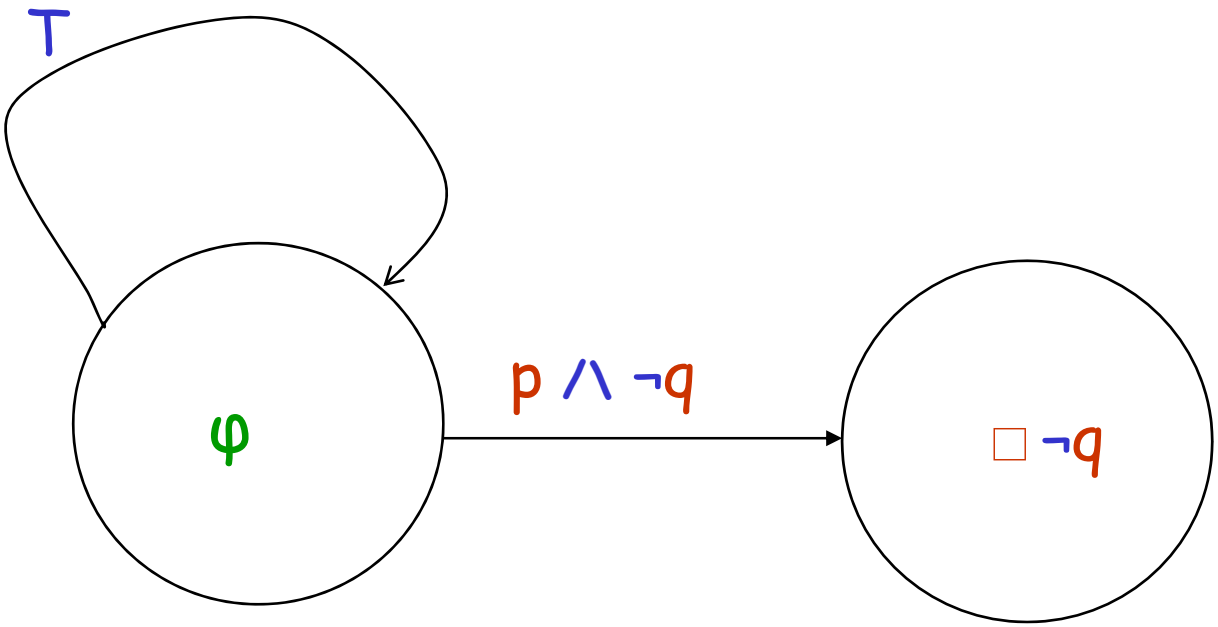
$$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$



Translating formulas to Automata

$$\varphi \equiv \Diamond(p \wedge \Box\neg q) \equiv (p \wedge \Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$

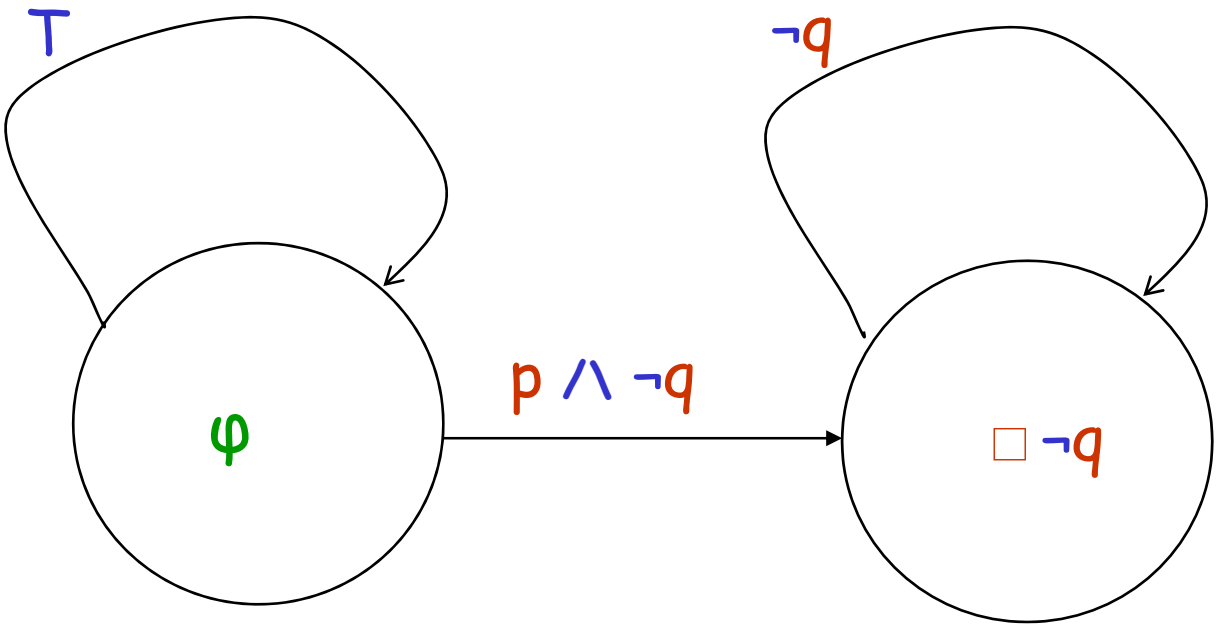
$$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$



Translating formulas to Automata

$$\varphi \equiv \Diamond(p \wedge \Box\neg q) \equiv (p \wedge \Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$

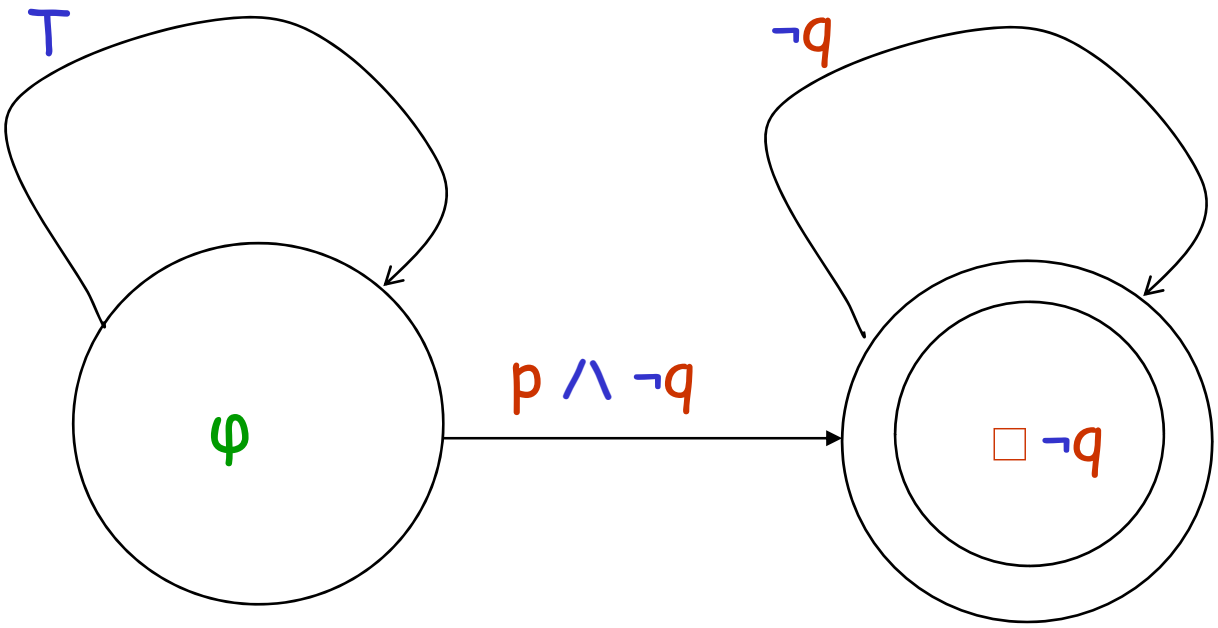
$$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$



Adding accepting states

$$\varphi \equiv \Diamond(p \wedge \Box\neg q) \equiv (p \wedge \Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$

$$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$$



Searching for accepting computations

Safety properties:

- $T \times A_{\neg\varphi}$ has self-loops on all accepting states
- Find a sequence of transitions from an initial state to a final states
- This is the reachability problem
- Can be solved by search from initial states
  - Visit all reachable states,
  - If accepting state is encountered, sequence of transitions = error trace

Liveness properties:

- Infinite computation of $T \times A_{\neg\varphi}$ must visit some accepting state infinitely many times
- Find a path to an accepting state, wich a loop to itself
- This is the repeated reachability problem
- Can be solved by doble search from initial states
  - Visit all reachable accepting states,
  - Search for loops from accepting states
  - If accepting loop (”lasso”, ”bad loop”) is encountered = error trace

General technique for finding loops

- Accepting runs == strongly connected components reachable from initial state
  - containing at least one accepting state
  - Having at least one internal transition
- Finding all SCCs of a graph can be done by Tarjan's algorithm (uses DFS)
- For finding some accepting cycle: more efficiently by nested depth-first search (Alur/Courcoubetis/Yannakakis, Holzmann)

Searching for reachable accepting loops

[Alur Courcoubetis Dill]

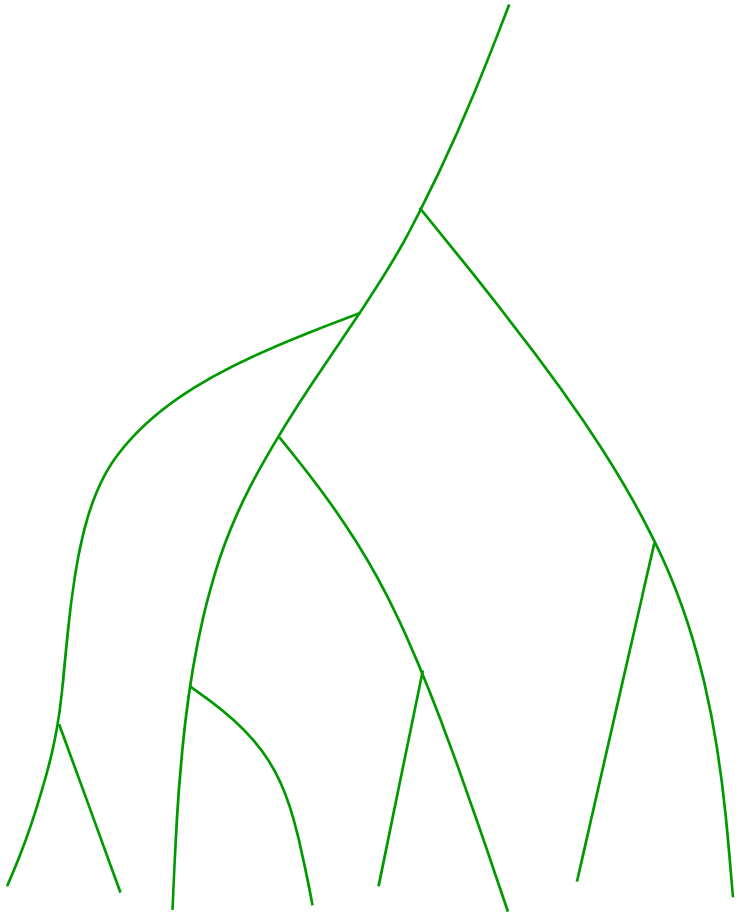
1. Perform depth-first search of the reachable states
2. List reachable accepting states in post-order
   $q1, \ldots, qn$
3. Search from each qi (starting with $q1$) to find a loop back to itself.
4. When post-order used, when searching states from qj, one need not reconsider states that were reached from states qi with $i < j$

Why post-order is good

Post-order is order of
popping states in DFS

If $q1, \ldots, qn$ in post-order
path from qi to qj with
$i < j$ must pass ancestor
of qi

Hence if state reached
from qi has path to qj
then qi is in cycle

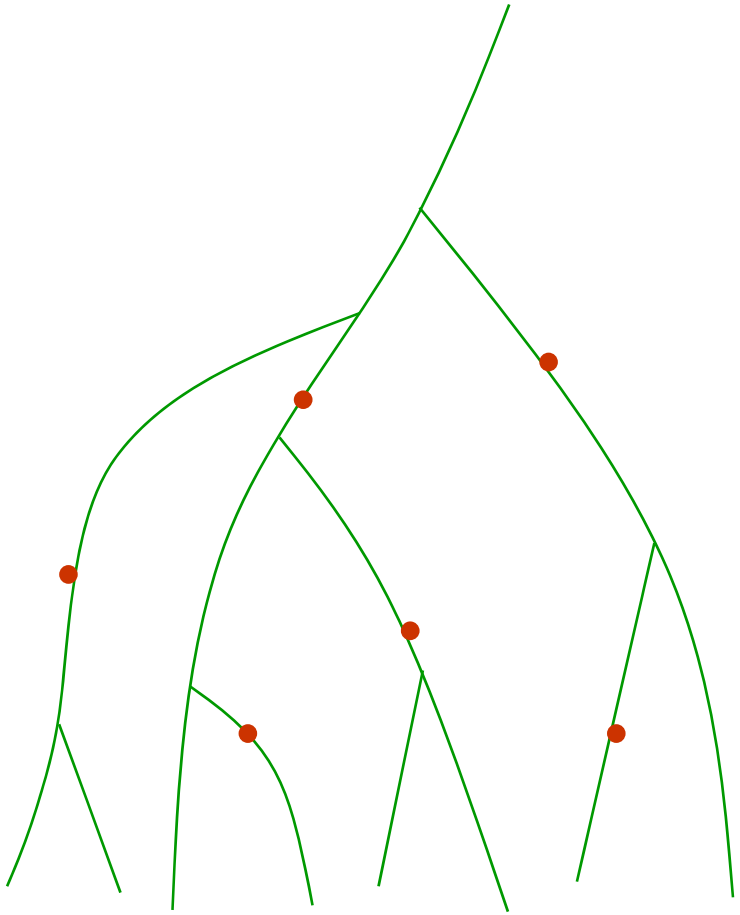


Why post-order is good

Post-order is order of
popping states in DFS

If $q1, \ldots, qn$ in post-order
path from qi to qj with
$i < j$ must pass ancestor
of qi

Hence if state reached
from qi has path to qj
then qi is in cycle

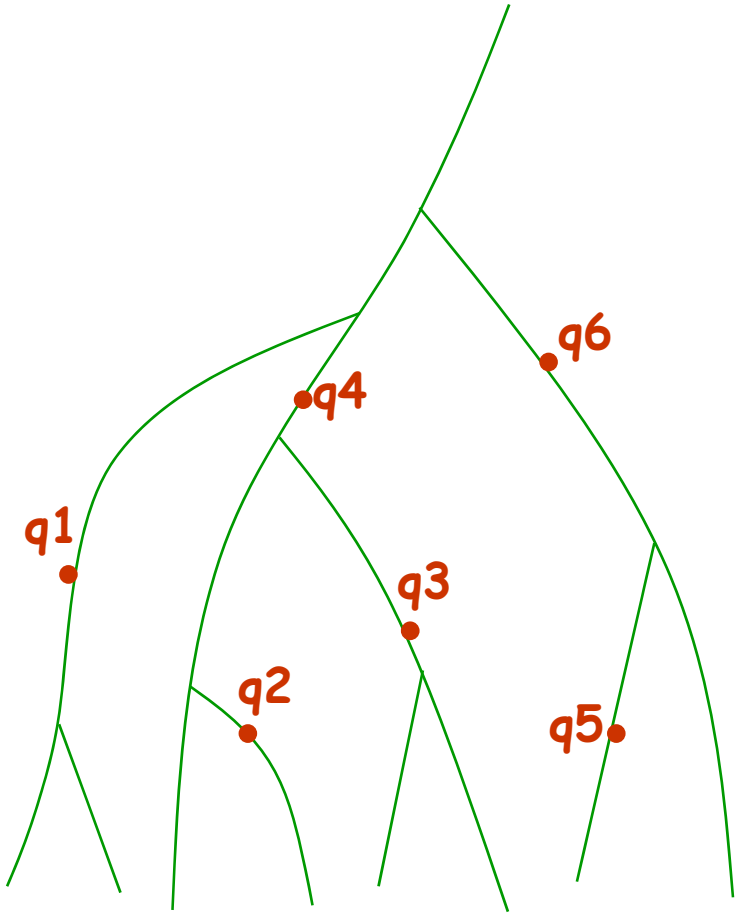


Why post-order is good

Post-order is order of popping states in DFS

If $q1, \ldots, qn$ in post-order path from qi to qj with $i < j$ must pass ancestor of qi

Hence if state reached from qi has path to qj then qi is in cycle

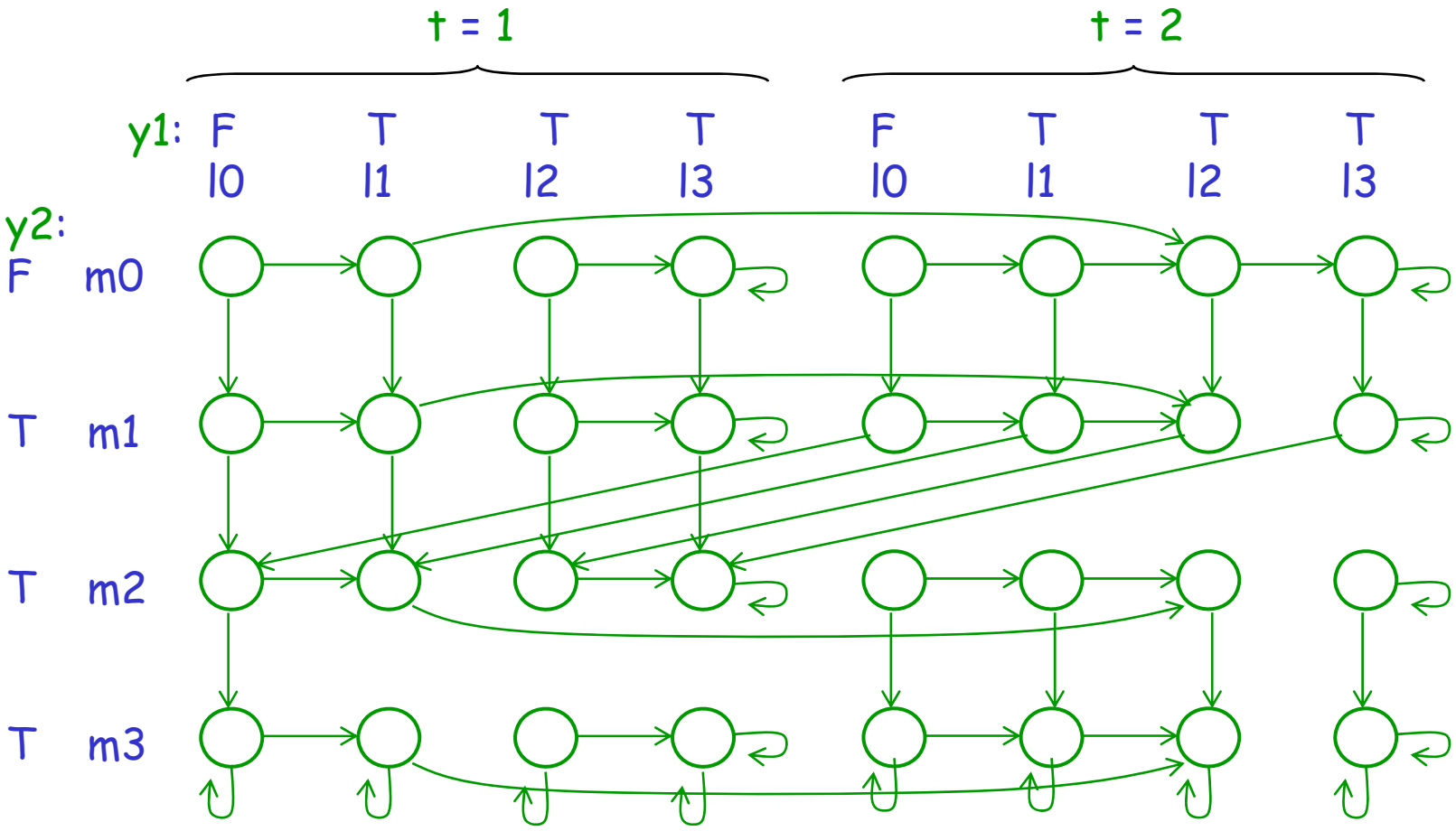


Pseudocode f. Nested Depth-First Search

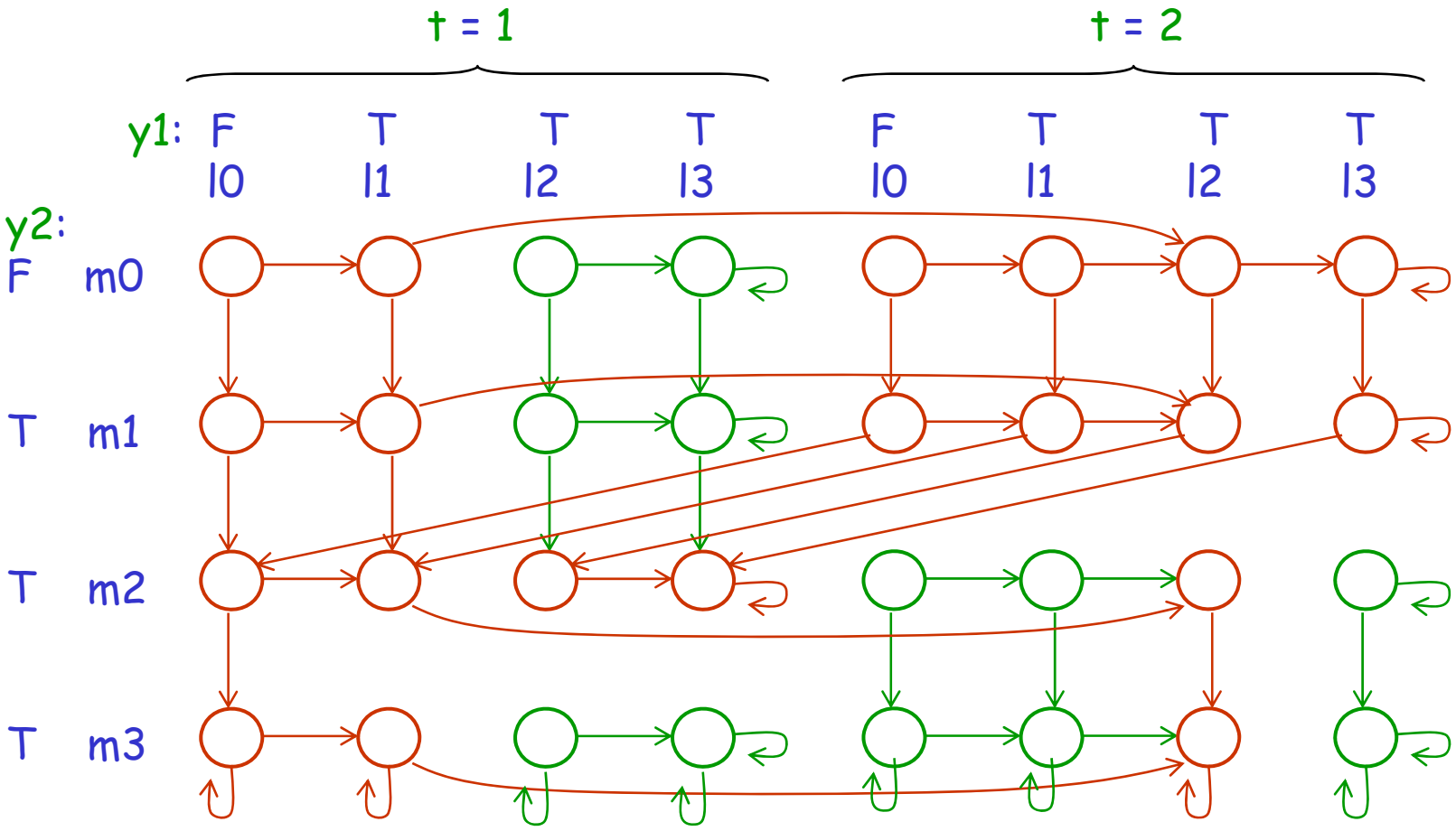
```
Procedure DFS1(s):
    Stack.push(s);Visited.insert(s);
    ∀ successors s' of s do
        if s' not in Visited then DFS1(s');
    if accepting (s) then DFS2(s);
    Stack.pop();

Procedure DFS2(s):
    flagged(s)   <-   true;
    ∀ successors s' of s do
        if s' in Stack then return "exists cycle"
        if not flagged(s') then DFS2(s');
```

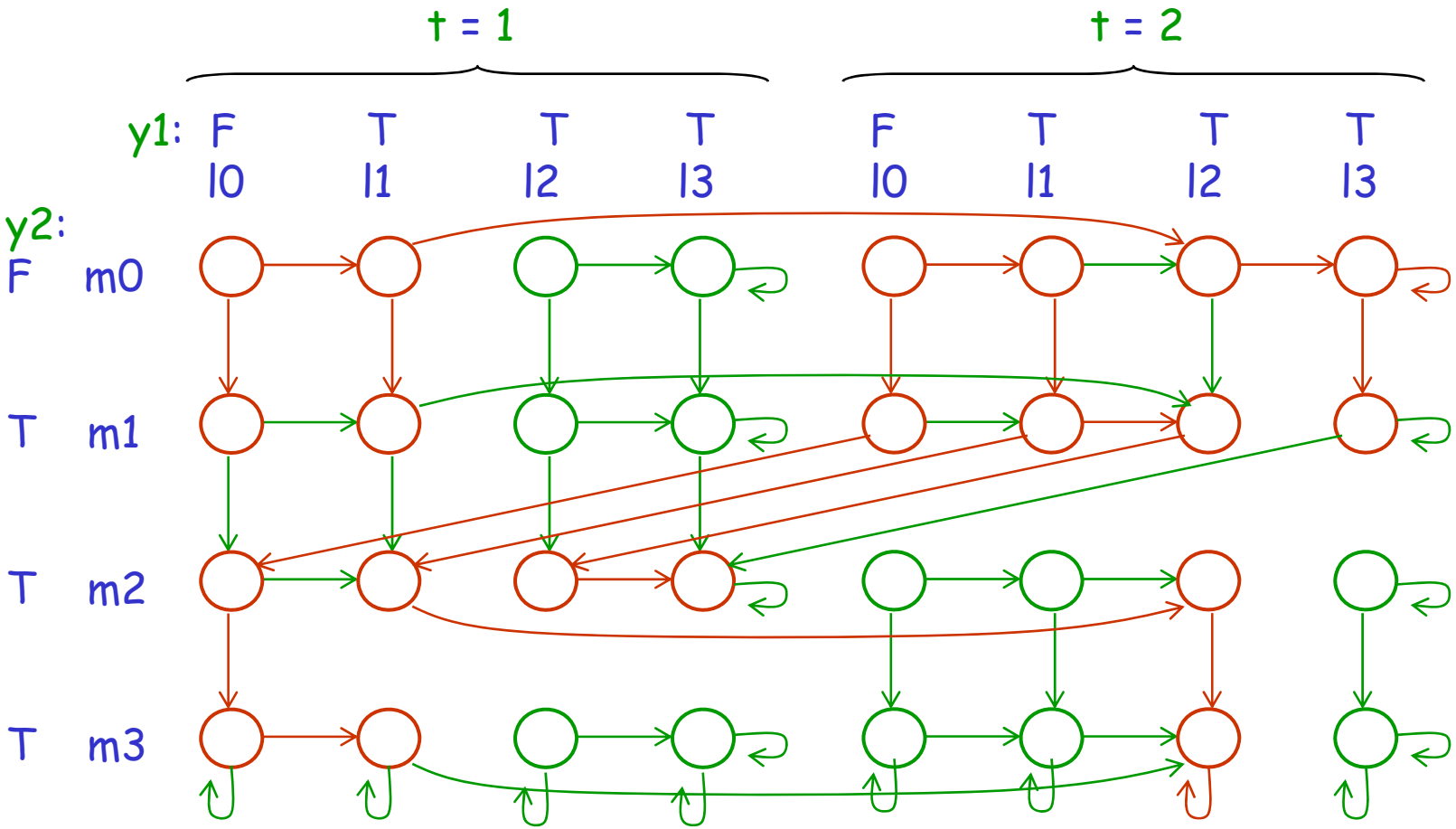
State-space exploration of PF



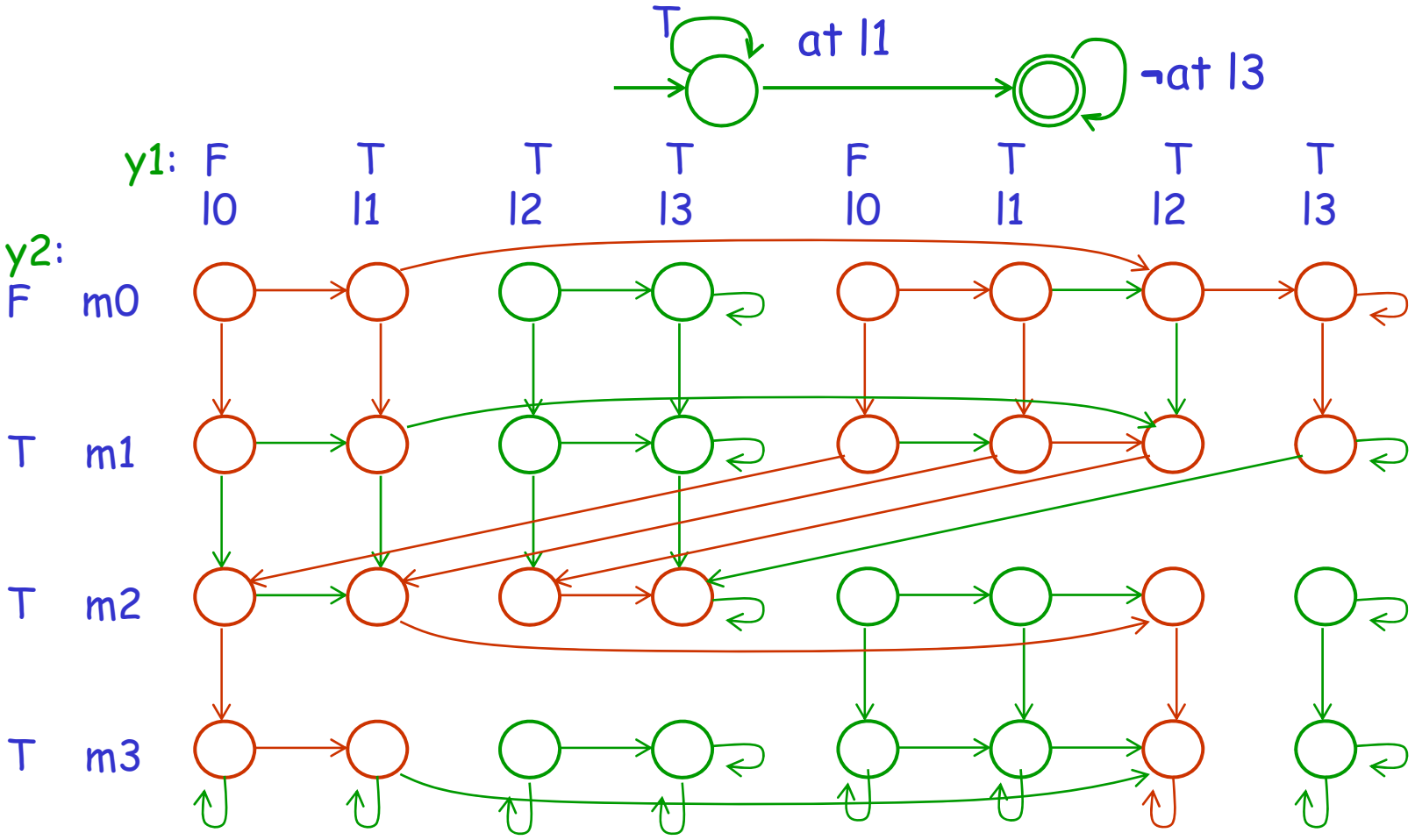
Reachable states



DFS tree

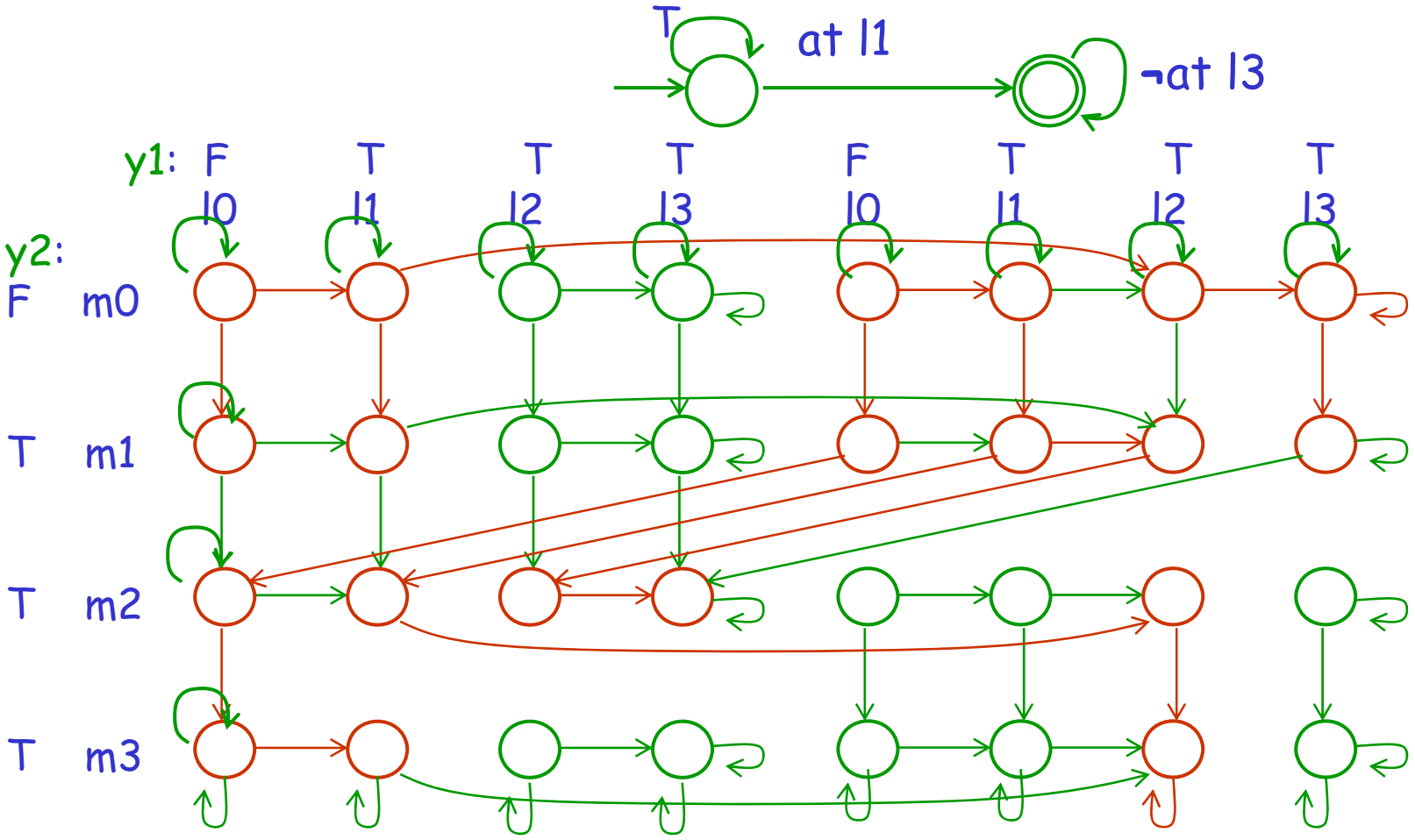


DFS tree with automaton



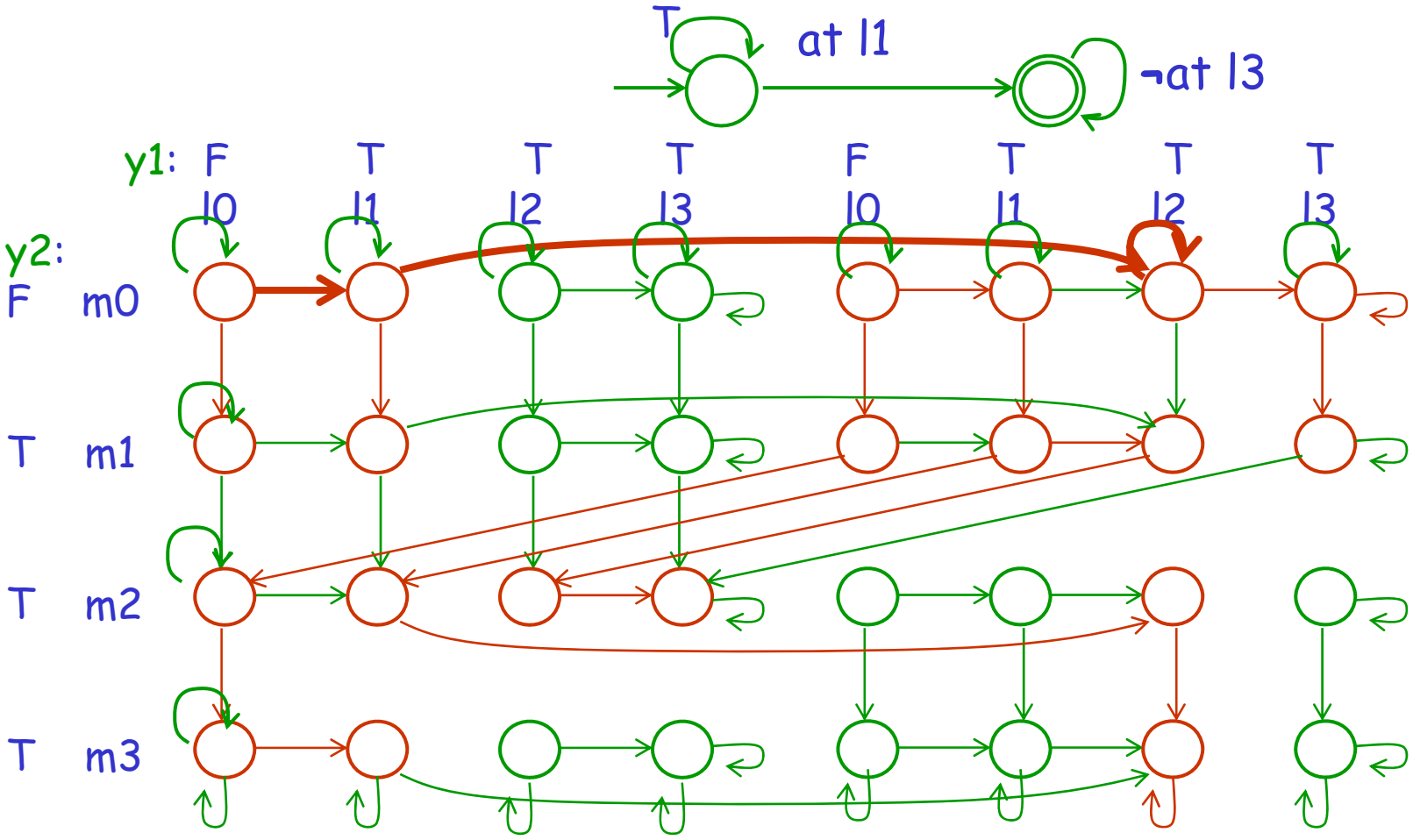
The graph has no loops that do not visit l3

Add self-loops at l0 and m0



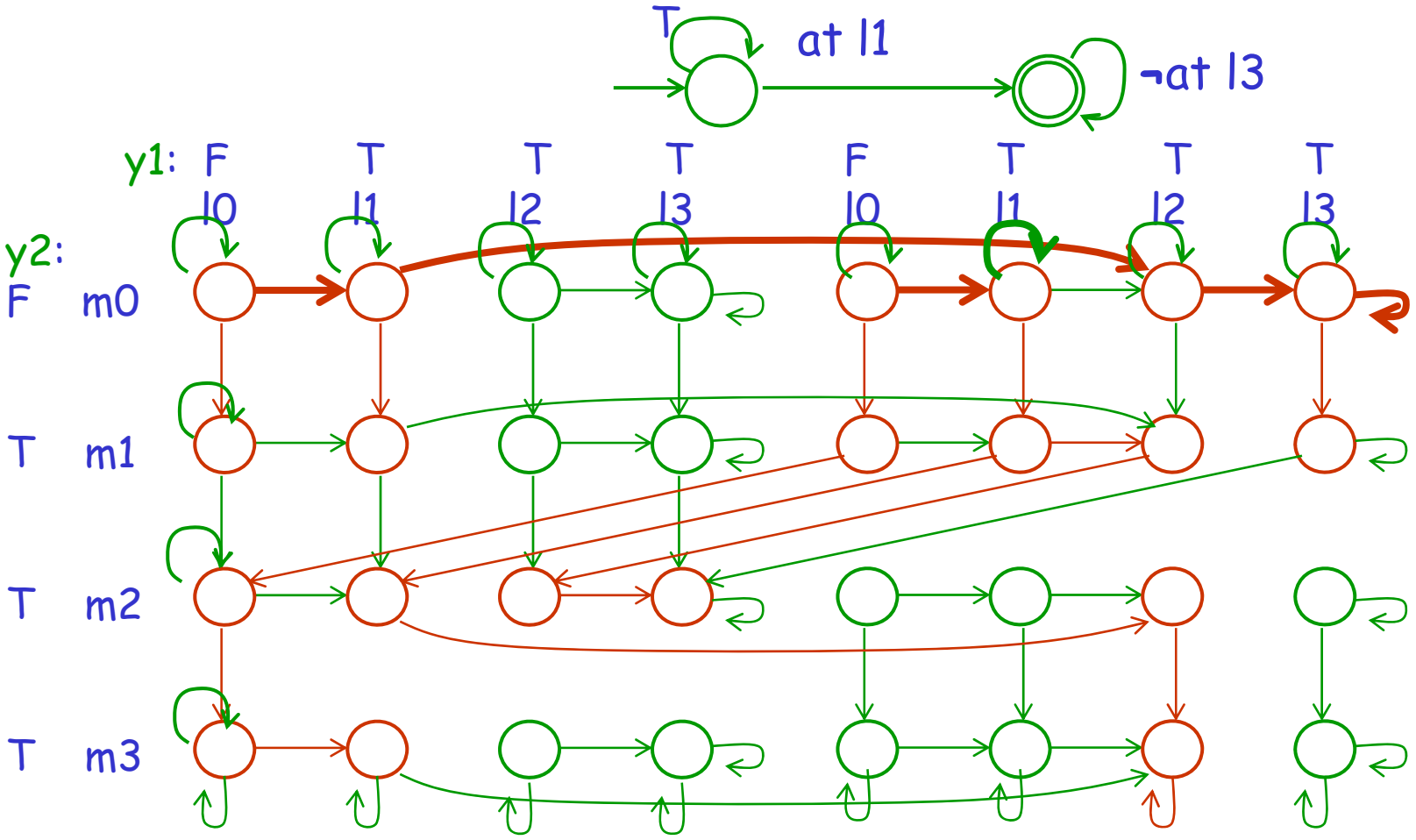
The graph has no loops that do not visit l3

Add self-loops at l0 and m0



The graph has a bad loop that does not visit l3

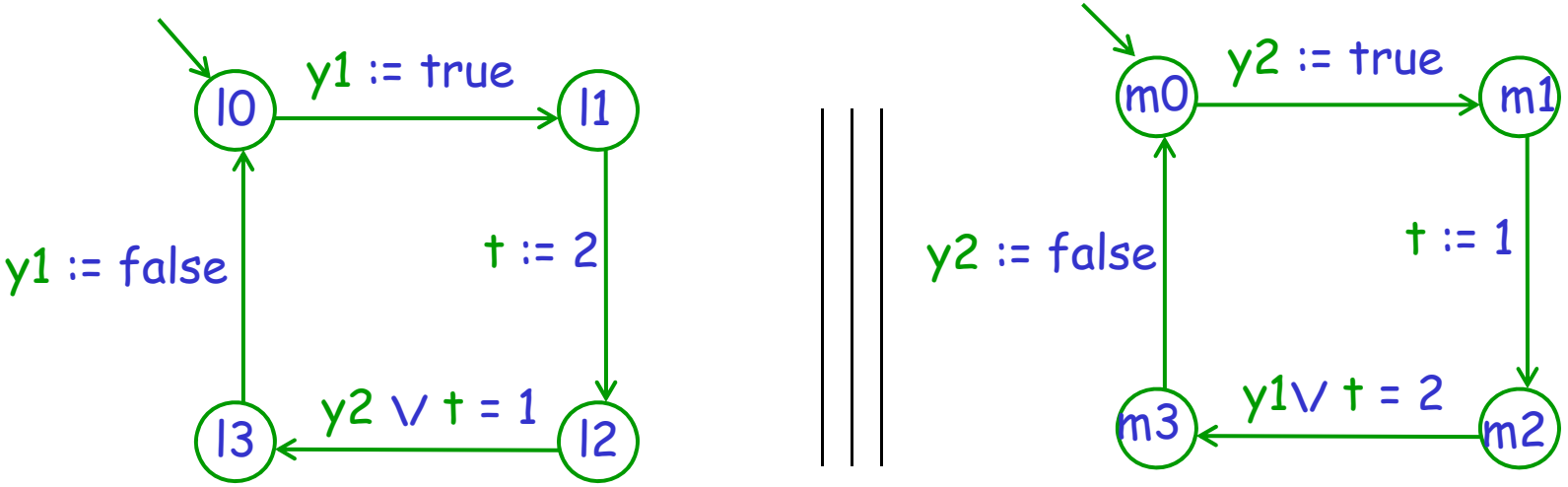
Add self-loops at l0 and m0



This is the first to be visited

Peterson-Fischer: Possible Specifications

Variables: y1, y2: boolean, t: {1,2}
Initially y1 = y2 = false, t = 1



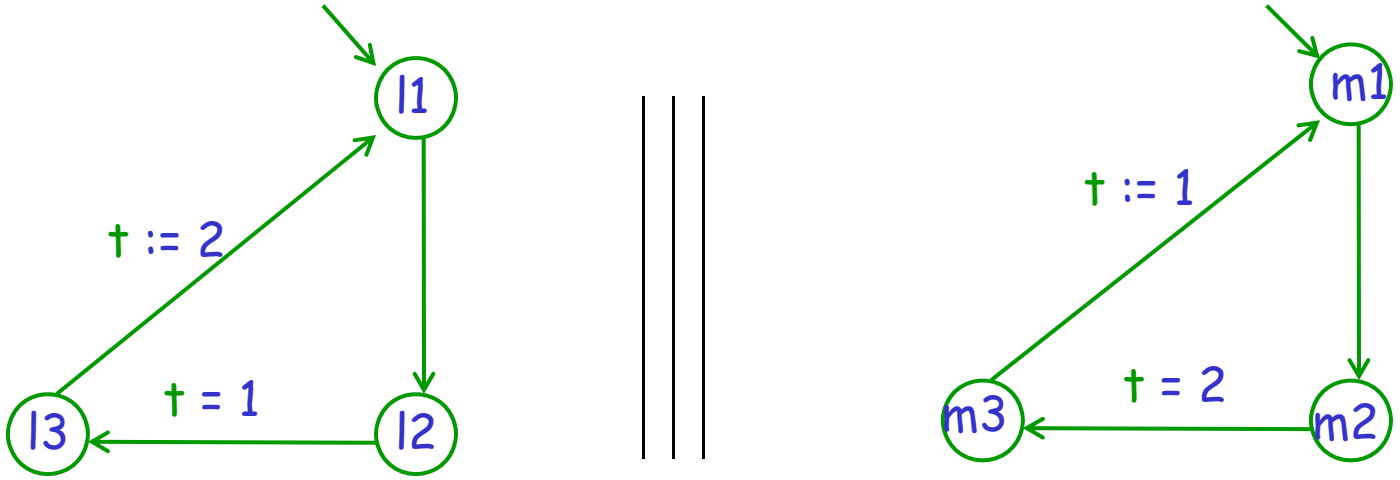
Mutual Exclusion: □ ¬(at l3 /\ at m3)

Absence of Starvation: □ (at l1 -> ◊ at l3)

Bounded Overtaking: □ (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

Simpler mutex

Variables: t: {1,2}
Initially t = 1

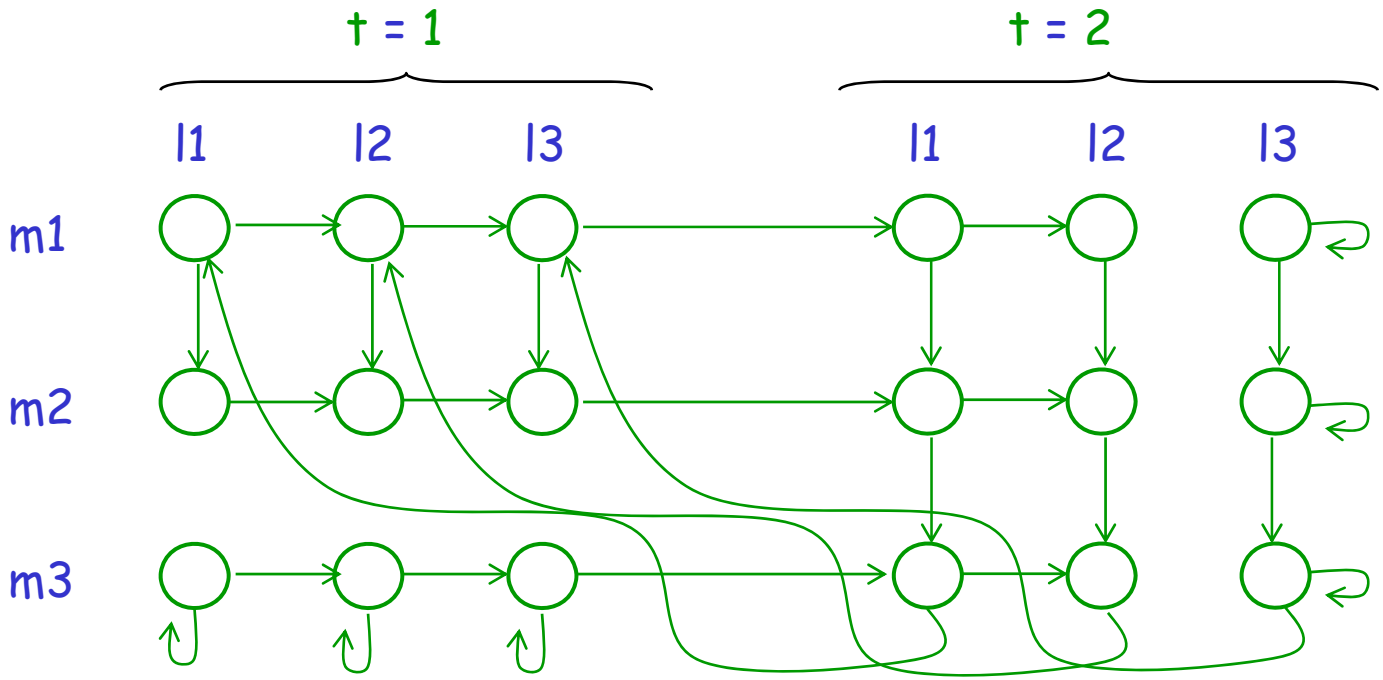


Mutual Exclusion: □ ¬(at l2 /\ at m3)

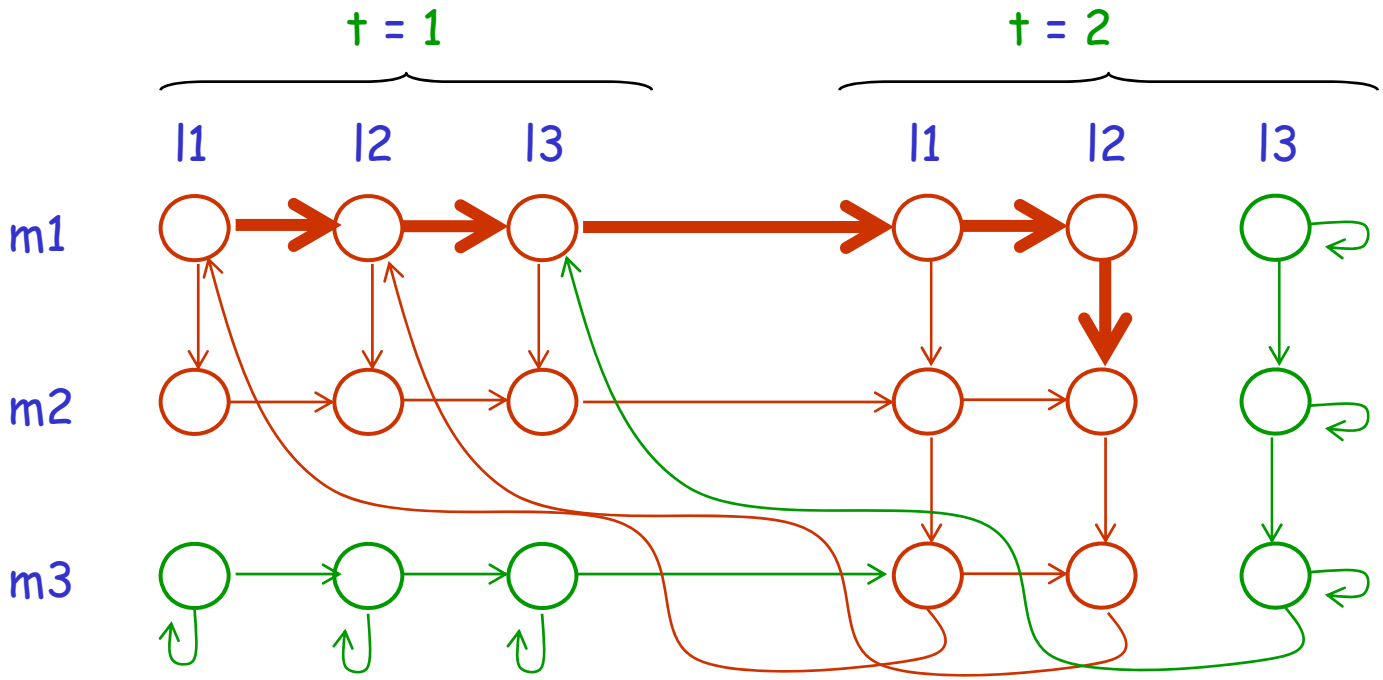
Absence of Starvation: □ (at l2 -> ◊ at l3)

Bounded Overtaking: □ (at l2 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

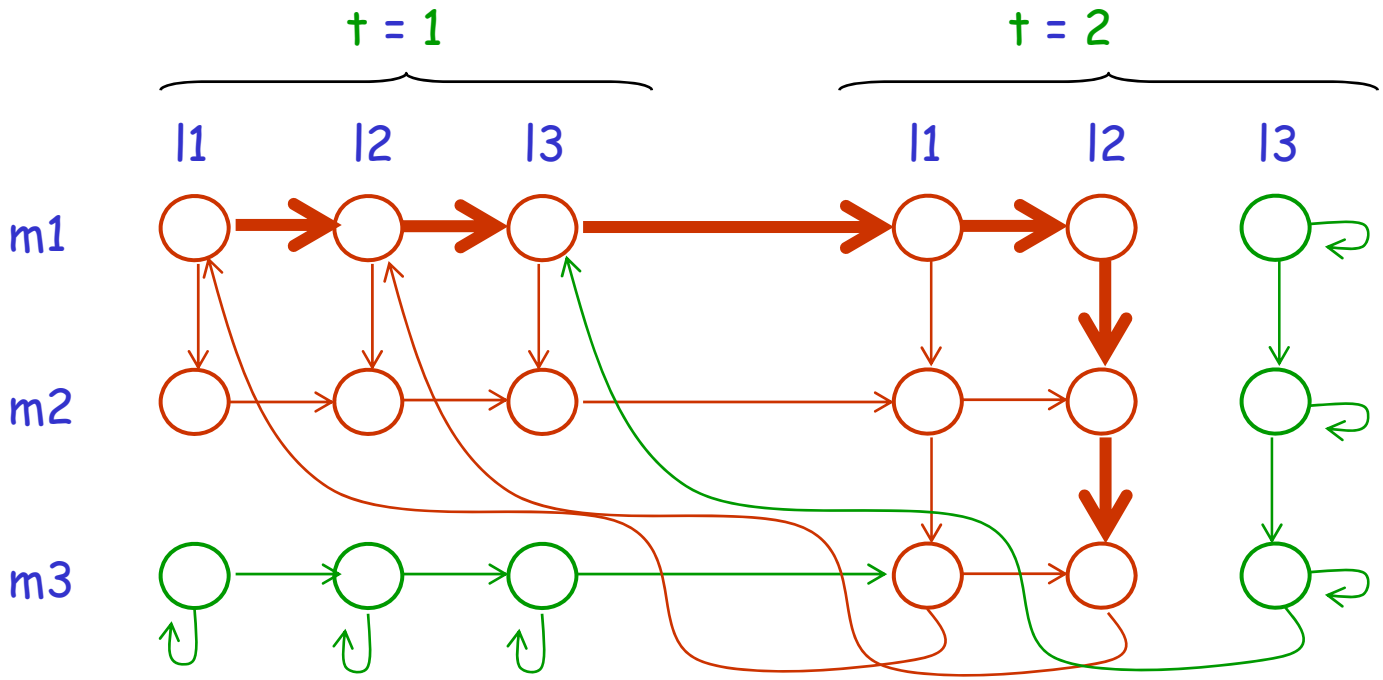
Simpler state space



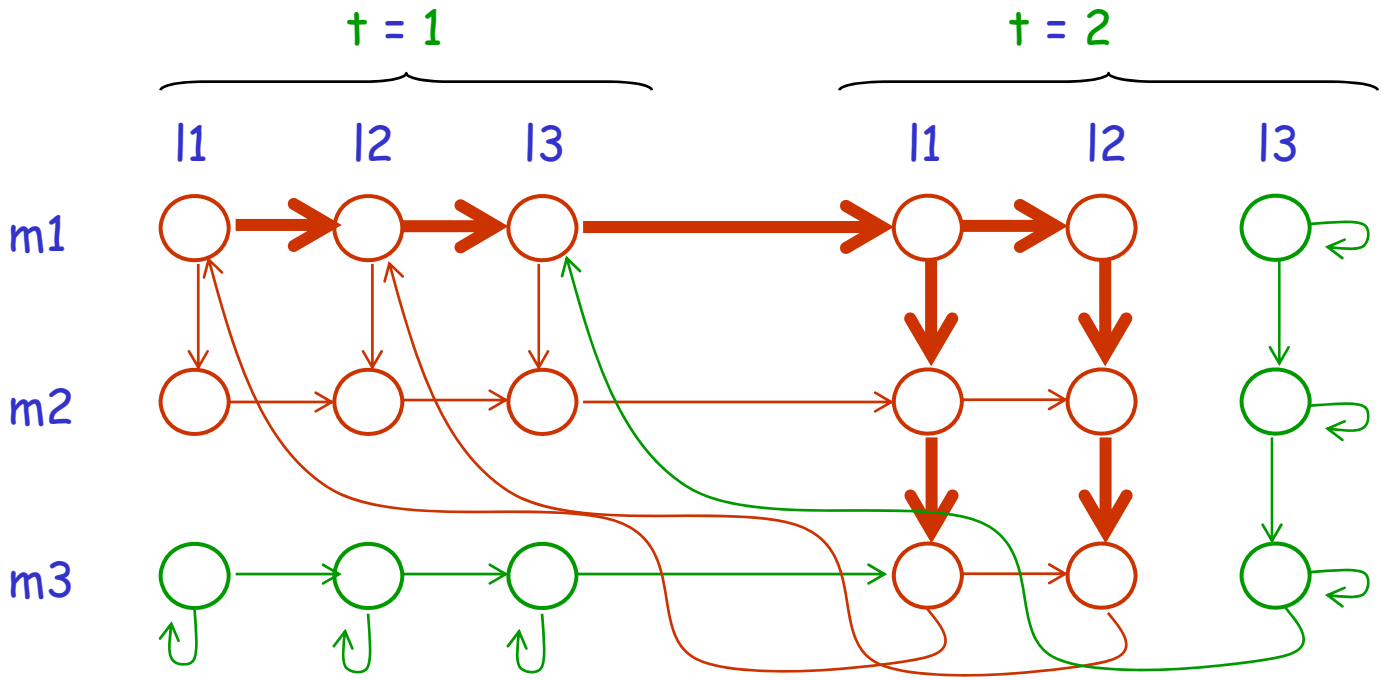
DFS tree



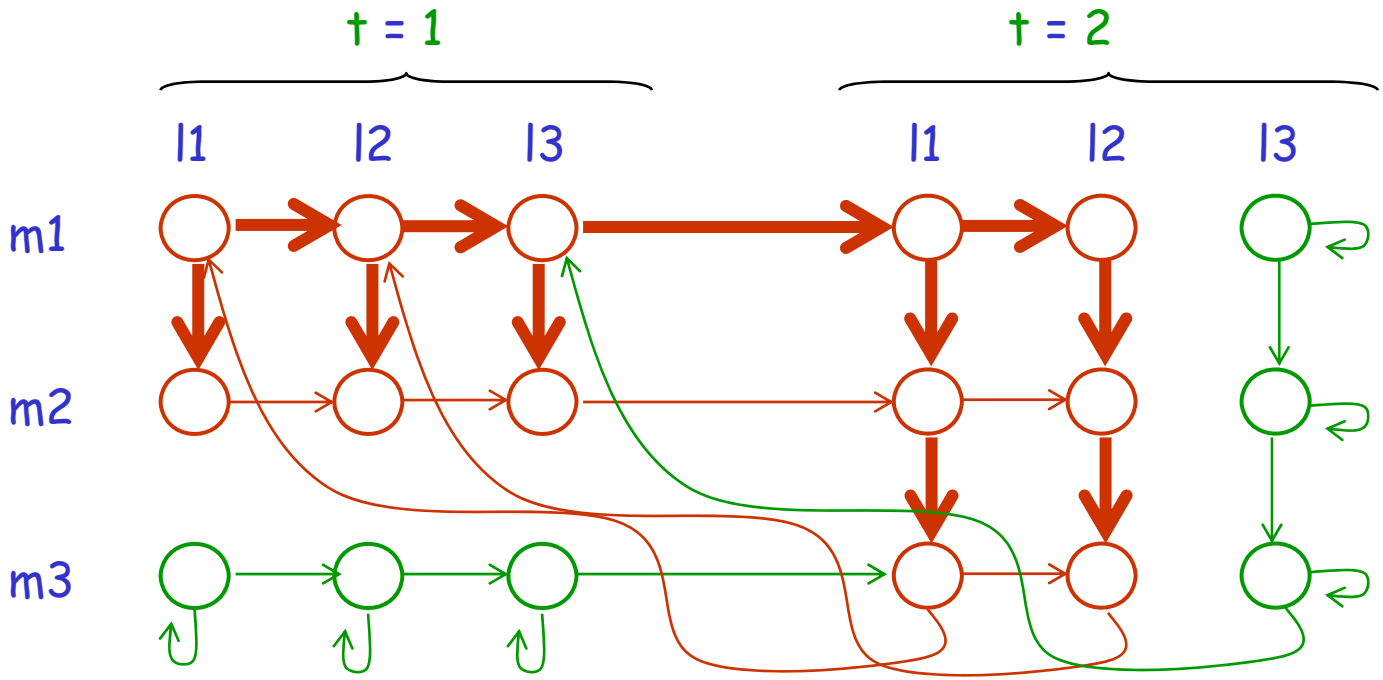
DFS tree



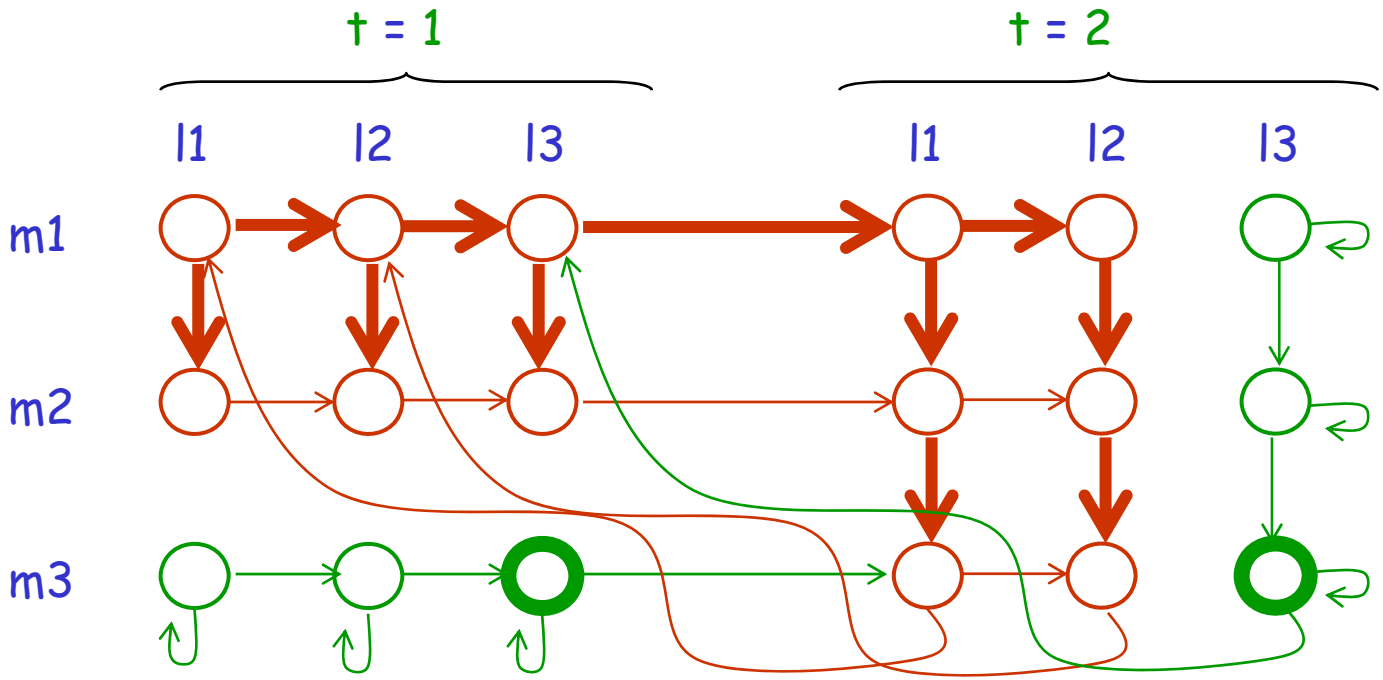
DFS tree



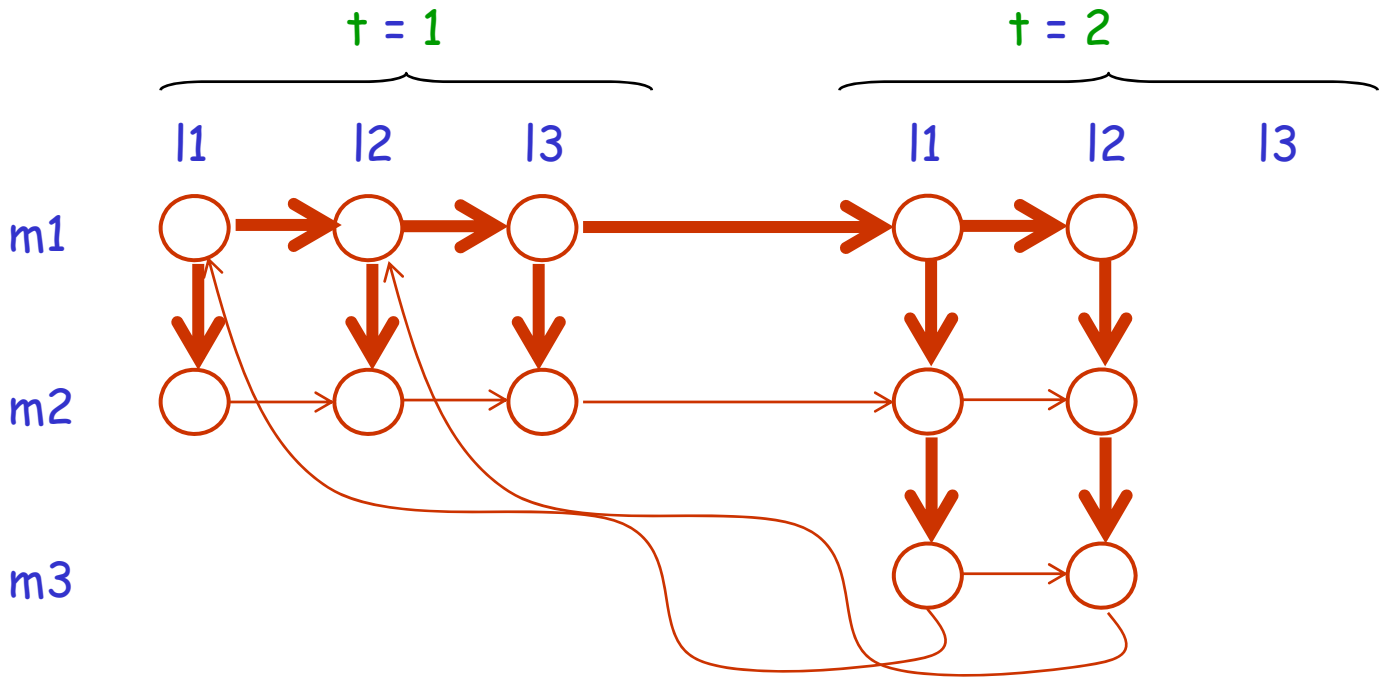
DFS tree



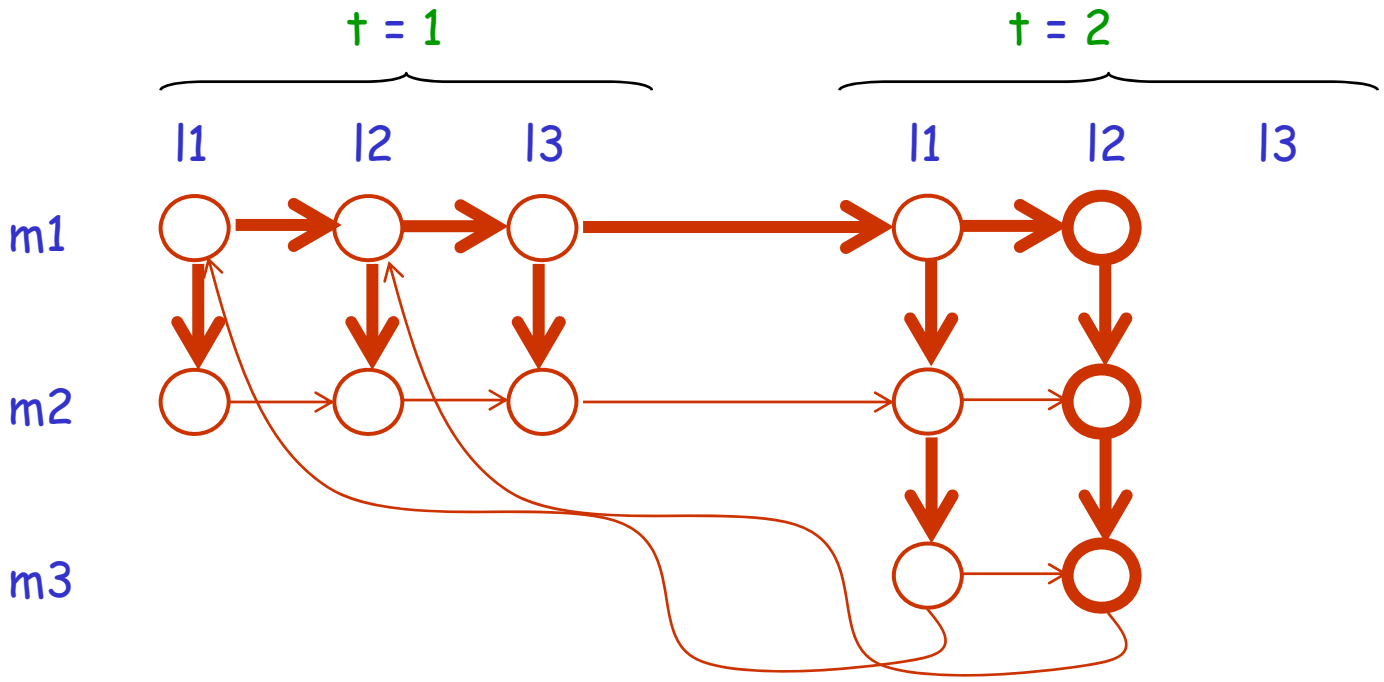
Bad states are unreachable



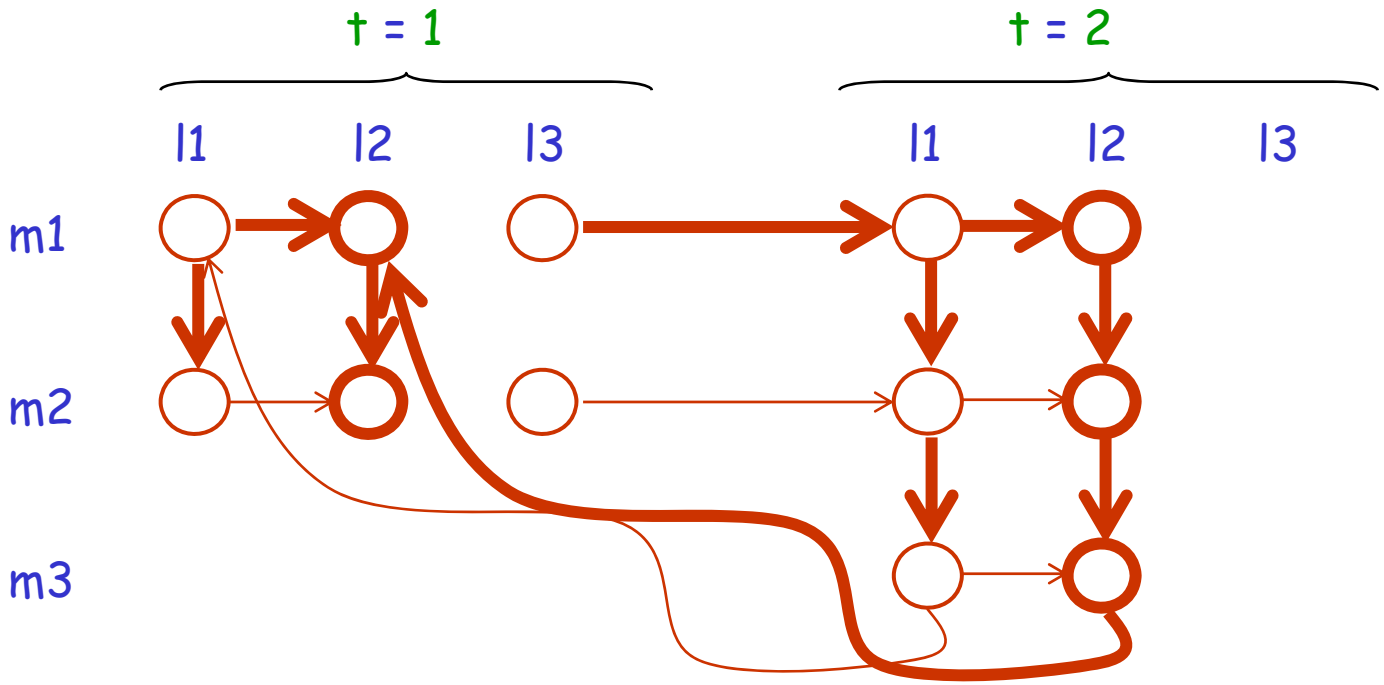
Reachable states



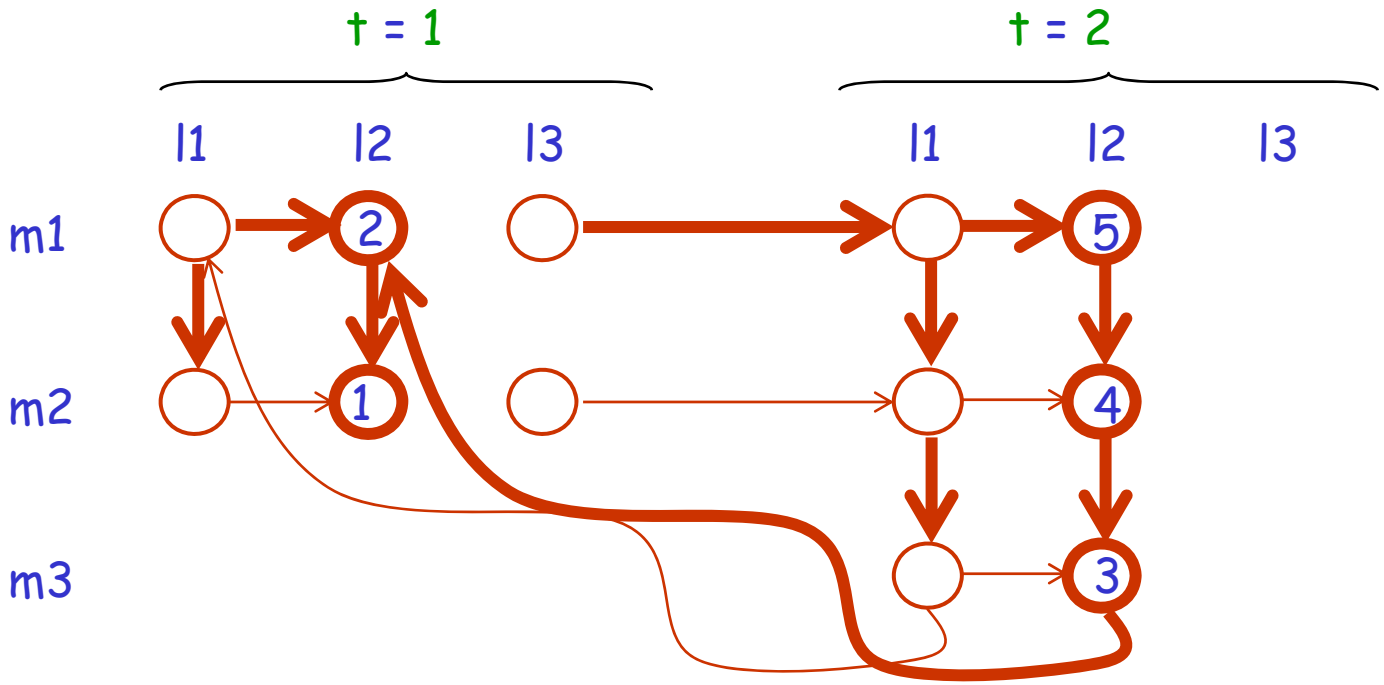
Accepting states for absence of starvation



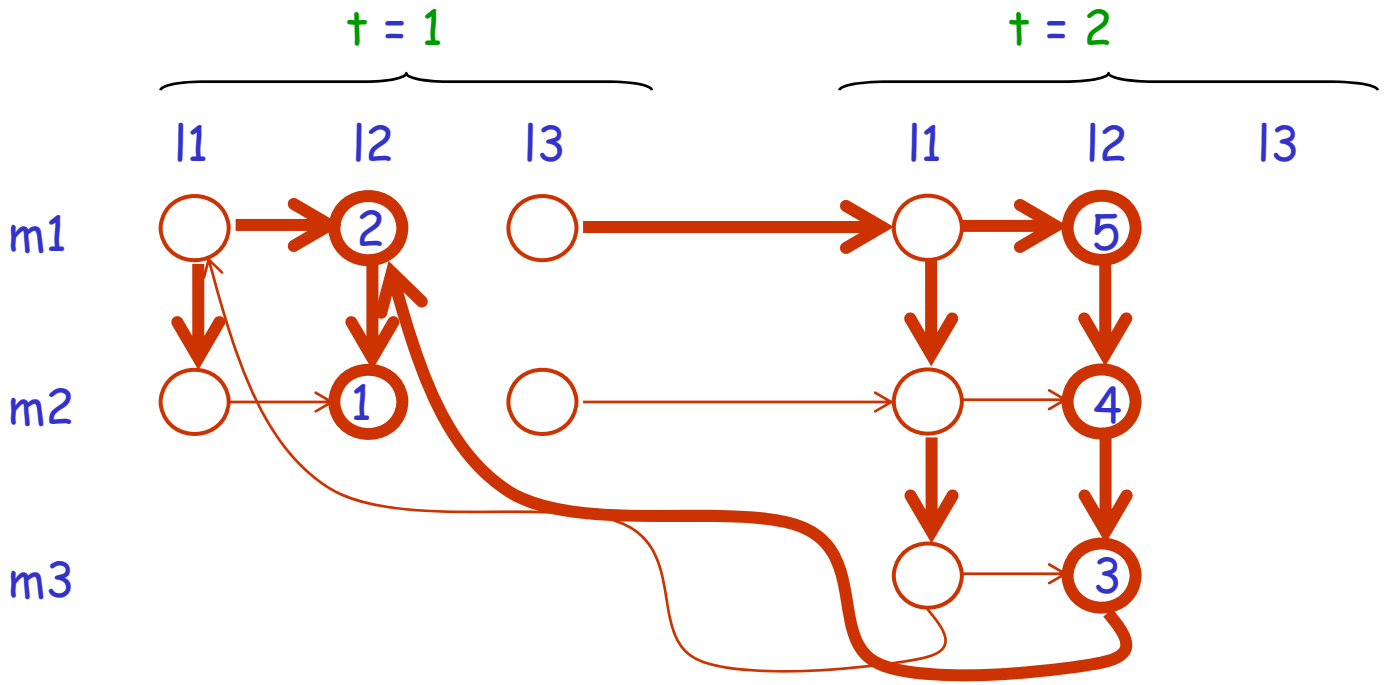
Accepting states for absence of starvation



Post-order

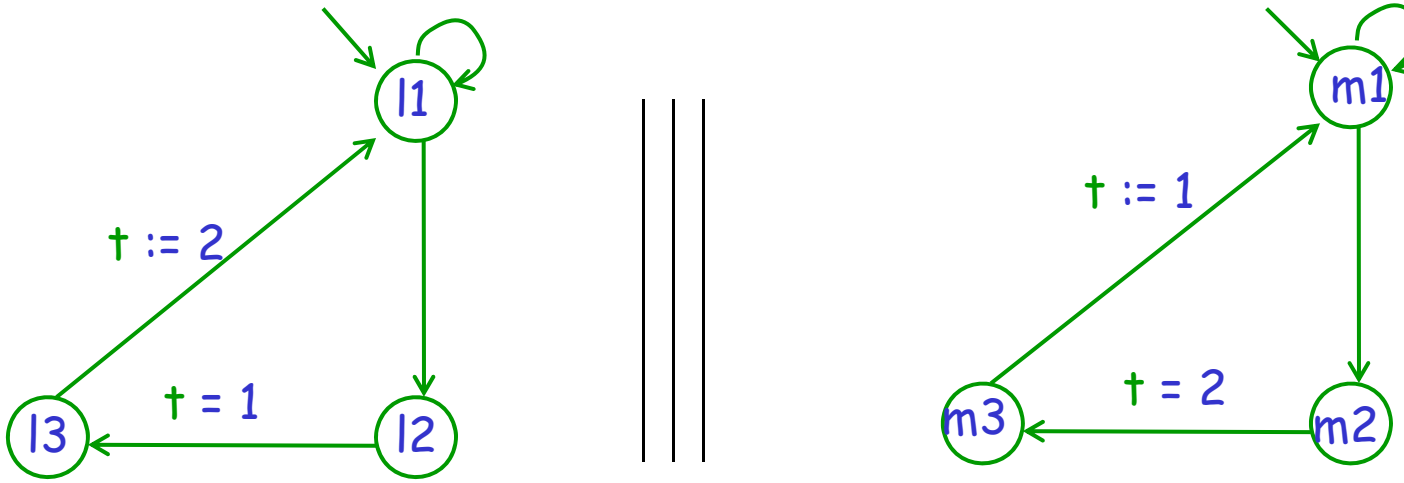


No loops can be found



Simpler mutex with self-loops

Variables: t: {1,2}
Initially t = 1

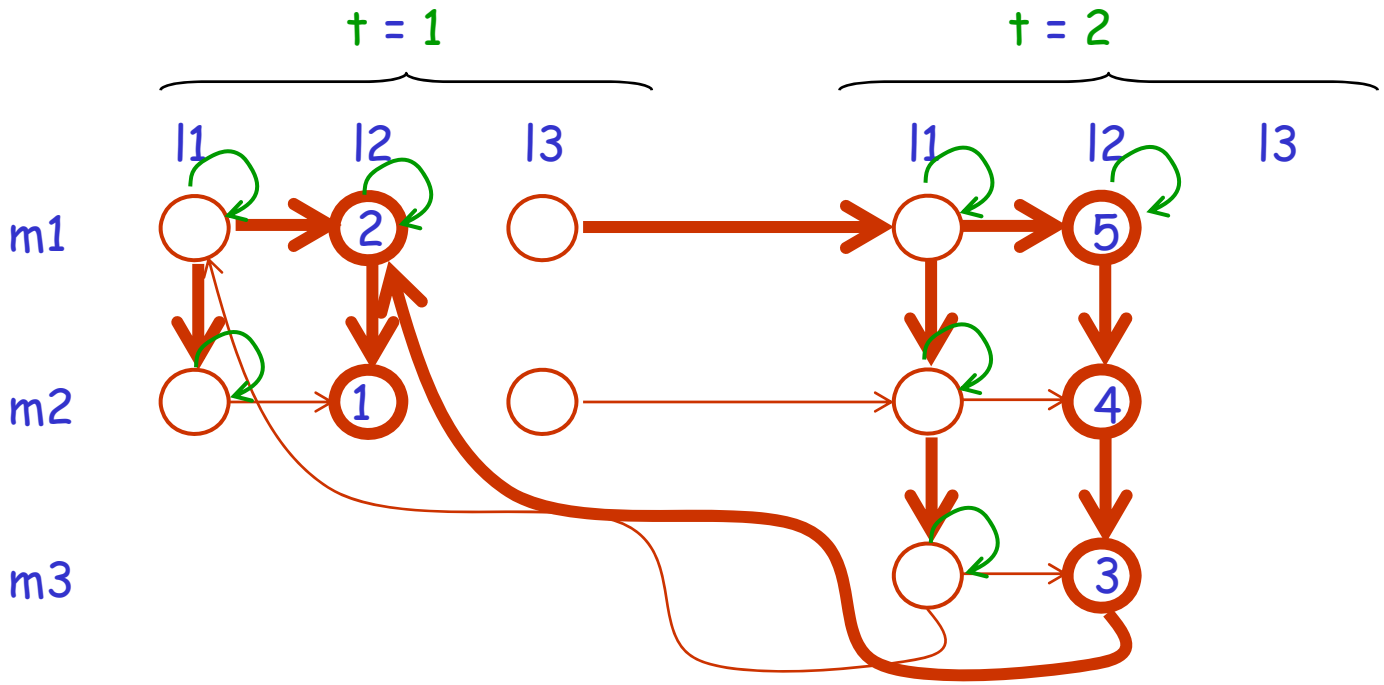


Mutual Exclusion: □ ¬(at l2 /\ at m3)

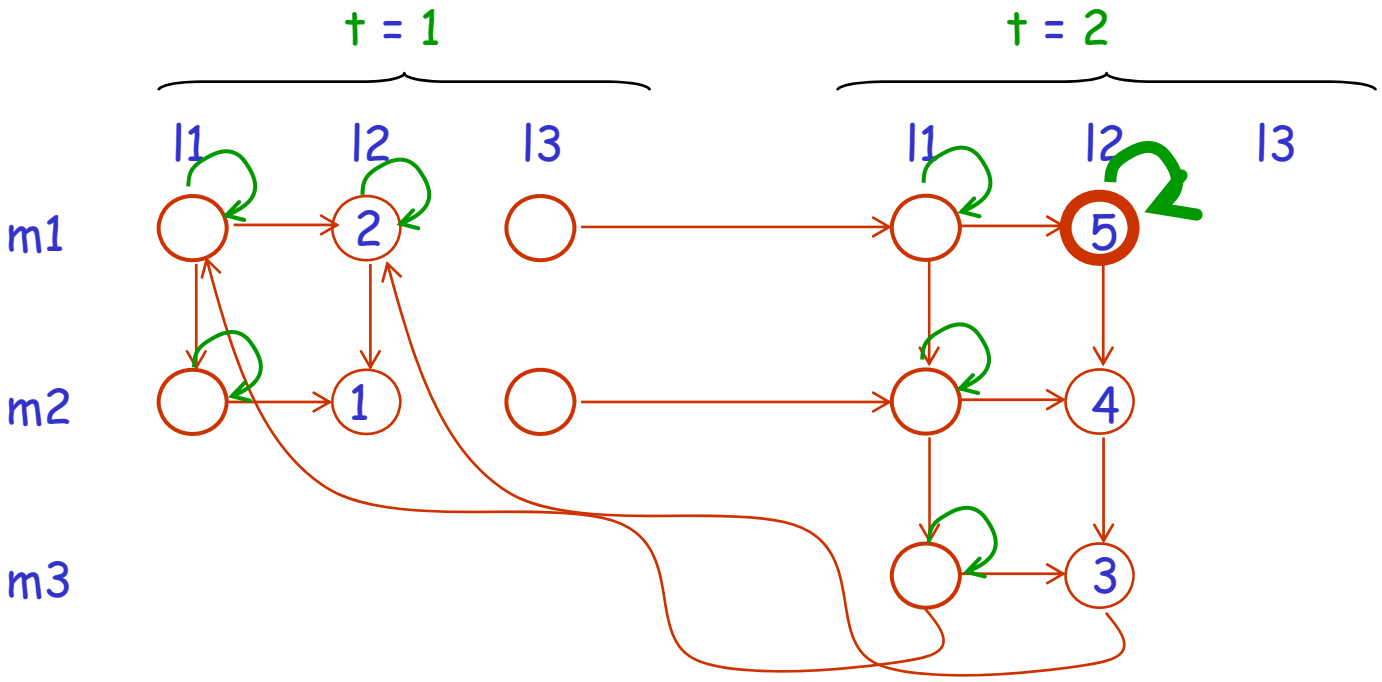
Absence of Starvation: □ (at l2 -> ◊ at l3)

Bounded Overtaking: □ (at l2 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

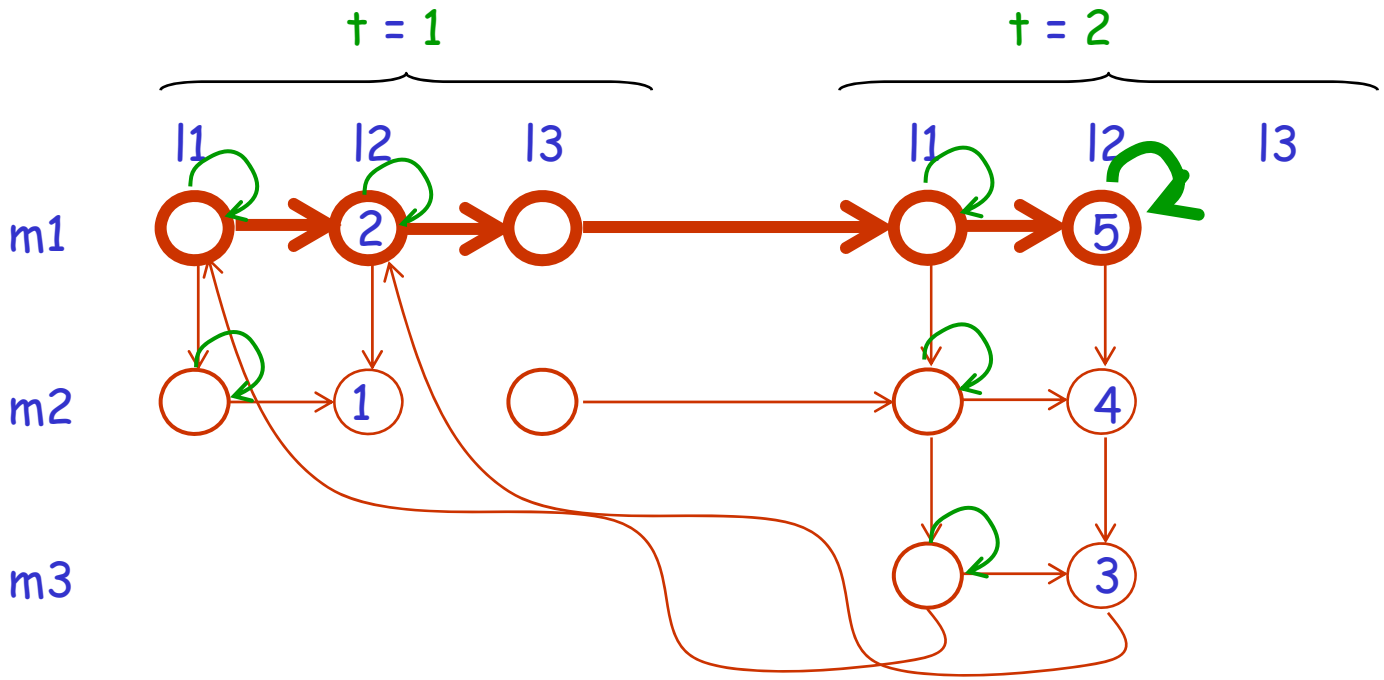
State space with self-loops



Idle loop found

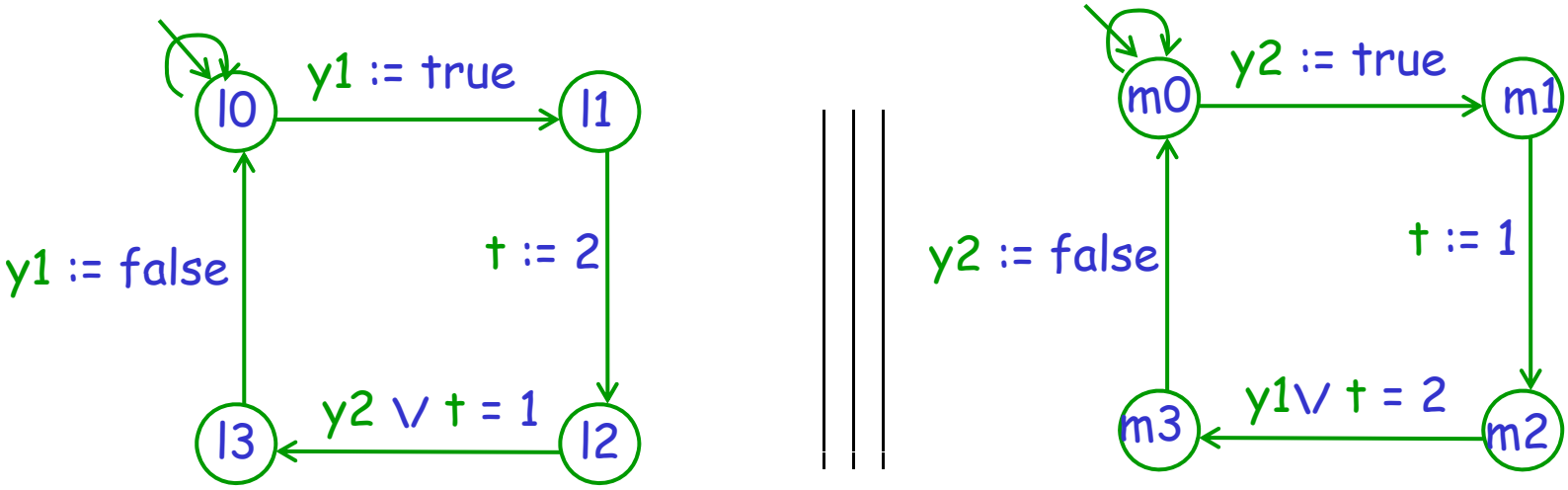


Corresponding bad reachable loop



Peterson-Fischer with idle loops

Variables: y1, y2: boolean, t: {1,2}
Initially y1 = y2 = false, t = 1



We need mechanism to avoid computations that only perform idle steps

Fairness

Fairness

- Assumption that some part of a transition system eventually progresses, without quantitative restrictions
- Can be viewed as an abstraction of many possible concrete transition scheduling policies.
- There are many different notions of fairness.
- The most common is weak fairness
- Another not uncommon is strong fairness

Fairness (definition)

- action *A* : any set of transitions (e.g., of one process)
- A computation

  s0 s1 s2 s3 s4 s5 s6 s7 . . .

  is
- weakly fair wrp. to *A* if
  - whenever *A* is enabled in all sj with j > i
    then some transition *A* is taken from some sk with k > i
  - □(□ enabled *A* -> ◇ taken *A*)
  - ◇□ enabled *A* -> □◇ taken *A*
- strongly fair wrp. to *A* if
  - whenever *A* is enabled in infinitely many sj with j > i
    then some transition *A* is taken from some sk with k > i
  - □(□ ◇enabled *A* -> ◇ taken *A*)
  - □◇ enabled *A* -> □◇ taken *A*

Fairness (definition)

- action *A* : any set of transitions (e.g., of one process)
- A computation

  s0 s1 s2 s3 s4 s5 s6 s7 . . .

  is
- Weakly fair wrp. to *A* if
  - whenever *A* is enabled in all s_j with $j > i$
  - then some transition *A* is taken from some s_k with $k > i$
  - □(□ enabled *A* -> ◇ taken *A*)
  - ◇□ enabled *A* -> □◇ taken *A*
- Strongly fair wrp. to *A* if
  - whenever *A* is enabled in infinitely many s_j with $j > i$
  - then some transition *A* is taken from some s_k with $k > i$
  - □(□ ◇enabled *A* -> ◇ taken *A*)
  - □◇ enabled *A* -> □◇ taken *A*

Checking fairness in exploration algorithm

To verify φ under fairness assumption

- Naive algorithm searches for bad loops that satisfy

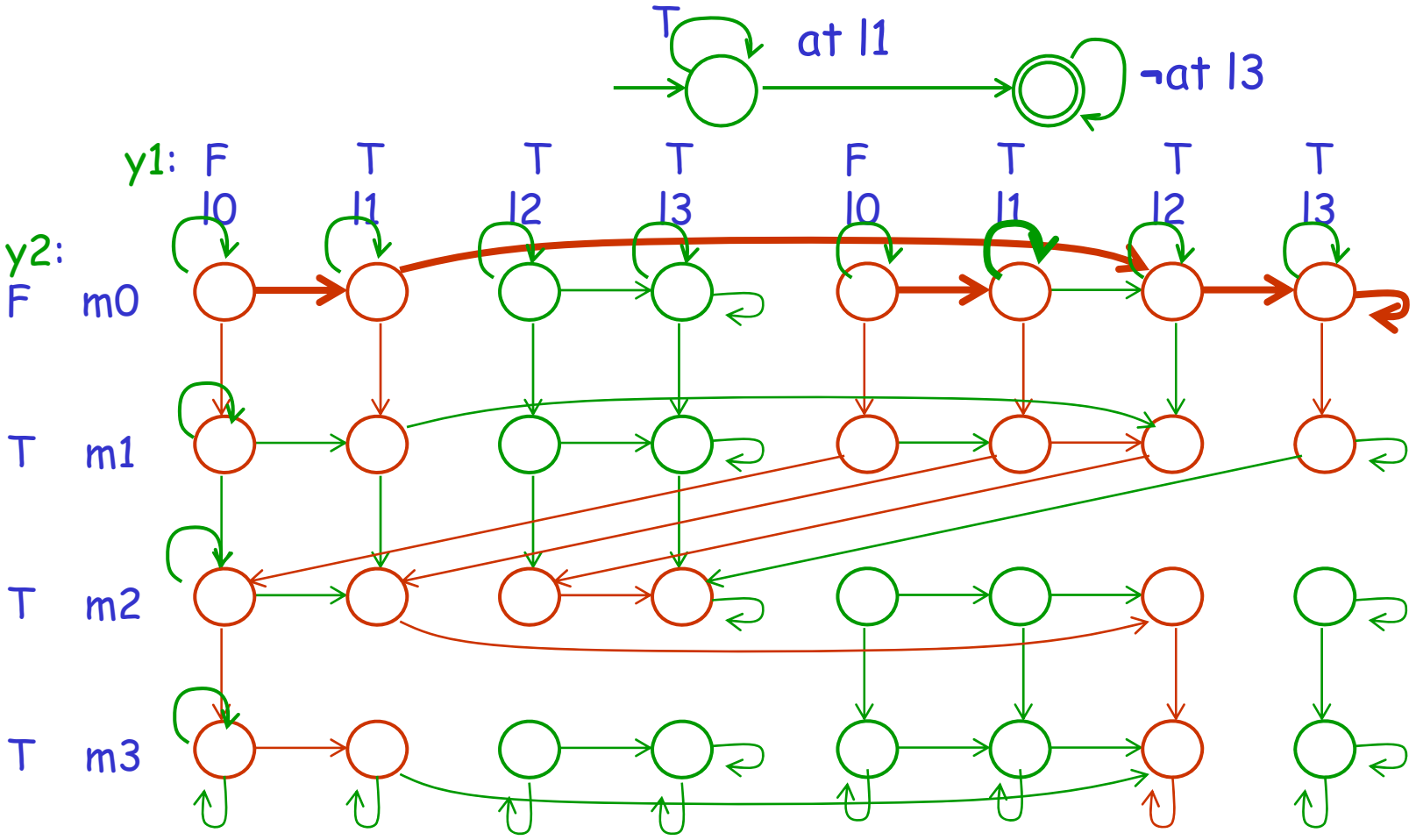
  fairness assumption /\¬φ

More efficient solution for weak fairness:

- search for bad loops that satisfy ¬φ

  in which each action *A* with weak fairness is once either disabled or taken

Add self-loops at l0 and m0



The loop does not have left process disabled or taken