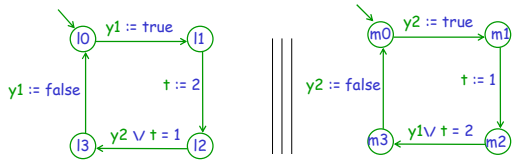
Lecture 1

linear time temporal logic

Peterson-Fischer: Possible Specifications

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



Mutual Exclusion: the two processes never simultaneously reach l3,m3

Absence of Starvation: If the left process is at l1, it will later reach l3

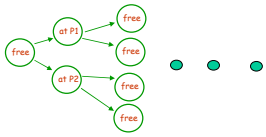
Bounded Overtaking: If the left process is at l1, the other process will reach m3 at most once (twice?) before the left process reaches l3

Two Classes of Formalizations

Linear Time Properties:
Specify properties that hold for all computations

Free in every 2nd state ;
at P1 always followed by free

Branching Time Properties:
Specify properties that hold for the computation tree (unfolding)



always possible to reach free ;
always possible to reach at P1 ;
free is always eventually reached

Transition Systems (formal definition)

A Transition System is a tuple $(S, S_0, \rightarrow, V, L)$ where

- S is a set of states
- S_0 is a set of initial states
- $\rightarrow$ (a subset of $S \times S$) is a transition relation
- V is a set of variables
- $L : S \rightarrow (V \rightarrow Val)$ gives a valuation of variables in each state

Write $s \rightarrow s'$ for $(s, s') \in \rightarrow$

A Computation is a finite or infinite sequence of states
$s_0 \; s_1 \; s_2 \; s_3 \; s_4 \; s_5 \; s_6 \; s_7 \; s_8 \; s_9$

such that
- s_0 is an initial state
- $(s_i, s_{i+1}) \in \rightarrow$ for all (relevant) i

Linear Time Properties

- Properties of Computations
- Specify "what happens when the system executes"
- Technical convenience
  - Consider only infinite computations
  - For finite computations, repeat the last state:

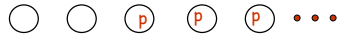
    $s_0 \; s_1 \; s_2 \; s_3 \; s_4 \; s_5 \; s_6 \; s_7$

    becomes

    $s_0 \; s_1 \; s_2 \; s_3 \; s_4 \; s_5 \; s_6 \; s_7 \; s_7 \; s_7 \; s_7 \ldots$
- In essence
  Linear time property φ = set of computations
- Transtion system T satisfies linear time property φ
  $T \models \varphi$
  if all computations of T are in (satisfy) φ

Linear Time Properties: examples

- Start from properties about states p , q
  e.g., $y0 < 1$ $x = y$ at m3
- p is invariant
- p is stable

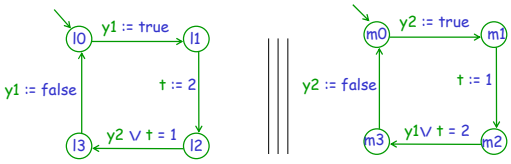


- p precedes q

- p leads to q

Peterson-Fischer: Possible Specifications

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



Mutual Exclusion: ¬(at l3 /\ at m3) is invariant

Absence of Starvation: at l1 leads to at l3

Bounded Overtaking: ?????

Linear Time Temporal Logic: formulas

- State formulas (denoted by p , q): properties about states
  e.g., y0 < 1 x = y at m3
- Formulas built using temporal operators
  - ○ φ "in the next state φ"
  - □ φ "always (in all future states) φ"
  - ◇ φ "sometimes (in some future states) φ"
  - φ U ψ "φ until ψ"
  - φ W ψ "φ unless ψ"
- And boolean connectives
  ¬ \/ /\ ->

Linear Time Temporal Logic: interpretation

- Formulas interpreted over computations
- A formula is either true or false in each state of a computation
- Example:

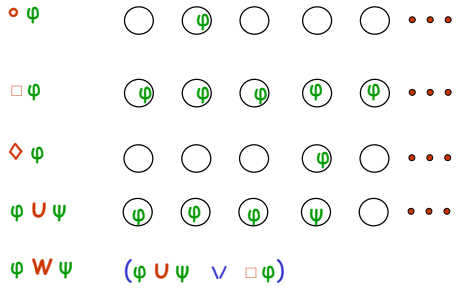
- The following formulas hold at the particular state

  ¬p /\ q
  ○ (p /\ q)

- The following formula does not hold at the particular state

  □ p

Meaning of operators



Operators: Formal semantics

Let ß be computation s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 ....

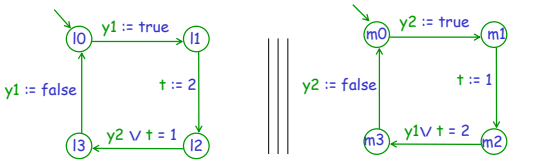
- (ß,i) |= φ denotes that φ is true in state s_i
- (ß,i) |= p (for p state formula) iff p holds in state s_i
- Boolean connectives work as usual
- (ß,i) |= ○ φ iff (ß,i+1) |= φ
- (ß,i) |= □ φ iff ∀j ≥ i (ß,j) |= φ
- (ß,i) |= ◇ φ iff ∃j ≥ i (ß,j) |= φ
- (ß,i) |= φ U ψ iff ∃j ≥ i (ß,j) |= ψ and
  ∀k : i ≤ k <j :(ß,k) |= φ
- (ß,i) |= φW ψ iff (ß,i) |= φ U ψ or (ß,i) |= □ φ

Linear Temporal logic: examples

- □ p — p is invariant
- □ (p -> □p) — p is stable
- ¬ q W (p /\ ¬q) — p precedes q
- □(p -> ◇ q) — p leads to q
- □◇ p — infinitely often p
- ◇□ p — from some point on p

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)

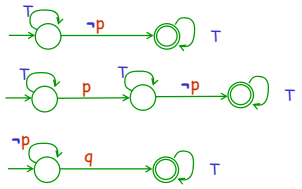
Bounded Overtaking: □ (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

Specifying linear properties by automata

- A linear time property is a set of computations.
- Can be seen as a language of (infinite) words,
  - Alphabet: possible assignments of truth-values to occurring state formulas
- A language can be specified by an automaton
- It turns out that it is better to represent the complement, i.e., the set of computations that violate the property

Automata Specifications: examples

- □ p
- □ (p -> □ p)
- ¬ q W (p /\ ¬q)
- □(p -> ◇ q)
- □◇ p
- ◇□ p



?

Superposing Automaton on Transition system

Superposition of Automaton (Σ, Q, Q_0 , -> , Φ)
onto Transition System (S, S_0 , -> , V , L) has

S x Q as the set of states

$\langle s,q \rangle \rightarrow \langle s',q' \rangle$ if $s \xrightarrow{s'} s'$ and $q \rightarrow q'$

Accepting condition adapted from Φ

Safety vs. Liveness properties.

- Safety property is of the form
  "nothing bad will ever happen"

A computation that violates the property will do so after a finite number of transitions

Enough to specify set of finite violating prefixes

- Liveness property is of the form
  "something good will eventually happen"

A computation that violates the property can never so after a finite number of transitions

We must specify set of infinite violating computations

- Any regular property is conjunction of safety and liveness properties.

Automata over infinite words

Automaton over infinite words: (Σ, Q, Q_0 , -> , Φ)
where,

- Σ is an alphabet (typically set of assignments of truth values to state formulas)
- Q is a set of states
- Q_0 is a set of initial states
- -> (a subset of S x Σ x S) is a transition relation
- Φ is an acceptance condition

A run over infinite word $a_1 a_2 a_3 a_4 a_5 a_6 a_7$ is

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \xrightarrow{a_4} q_4 \xrightarrow{a_5} q_5 \xrightarrow{a_6} q_6 \xrightarrow{a_7} q_7 \xrightarrow{a_8} q_8$$

which satisfies the acceptance condition Φ

Classes of acceptance conditions

Buchi automata
- One set of accepting states F
- Acceptance condition □◇ F

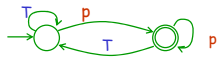
Generalized Buchi automata
- Collection of sets of accepting states $F_1, \dots, F_n$
- Acceptance condition $\square\Diamond F_1 \wedge \dots \wedge \square\Diamond F_n$

Rabin automata
- Collection of pairs of accepting states $(F_1, G_1), \dots, (F_n, G_n)$
- Acceptance condition $\bigvee_i (\square\Diamond F_i \wedge \neg\square\Diamond G_i)$

Streett automata
- Collection of pairs of accepting states $(F_1, G_1), \dots, (F_n, G_n)$
- Acceptance condition $\bigwedge_i (\square\Diamond F_i \rightarrow \square\Diamond G_i)$

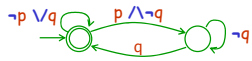
Examples of Buchi Automata

- □◇ p

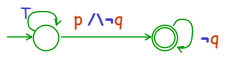


- ◇□ p

- □(p -> ◇ q)



- ◇(p ∧ □¬q)

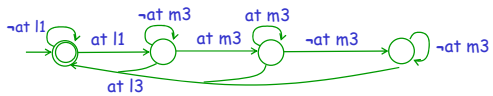


Translation Results

- Any property expressible in linear temporal logic is accepted by some Buchi automaton
- There are properties accepted by some Buchi automaton that can not be expressed in linear temporal logic
  - e.g., "p is true in every even state"

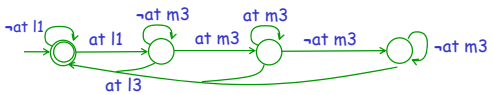
Convenience of expression

- Some properties are easier to understand as automata
- Bounded Overtaking:
  - (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

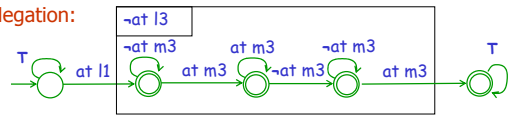


Convenience of expression

- Some properties are easier to understand as automata
- Bounded Overtaking:
  - (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

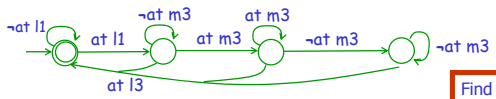


Negation:

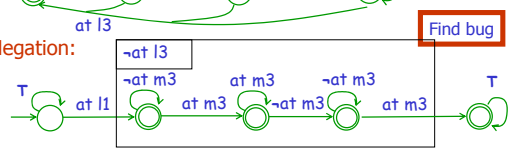


Convenience of expression

- Some properties are easier to understand as automata
- Bounded Overtaking:
  - (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))



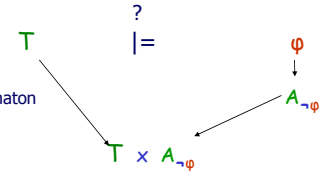
Negation:



Model checking Linear Temporal logic

Automata-Theoretic Approach [Vardi Wolper 1986]

- Question
- Construct
  Buchi Automaton
- Combine



- Find
  accepting computation (error trace)

Searching for accepting computations

Safety properties:

- T × $A_{\neg\varphi}$ 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 × $A_{\neg\varphi}$ 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

Searching for reachable accepting loops

[Alur Courcoubetis Dill]

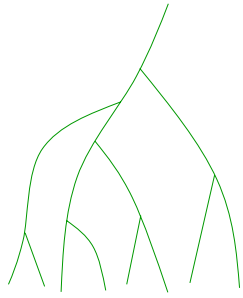
1. Perform depth-first search of the reachable states
2. List reachable accepting states in post-order
   q1, …, qn
3. Search from each qi (starting with q1) to find a loop back to itself.
4. When post-order used, when searching states from qj, one need not reconsider states that were reached from states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

Why post-order is good

Post-order is order of popping states in DFS

If q1, …, 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

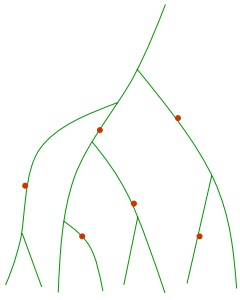


Why post-order is good

Post-order is order of popping states in DFS

If q1, …, 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

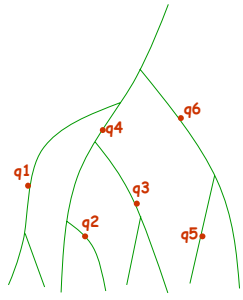


Why post-order is good

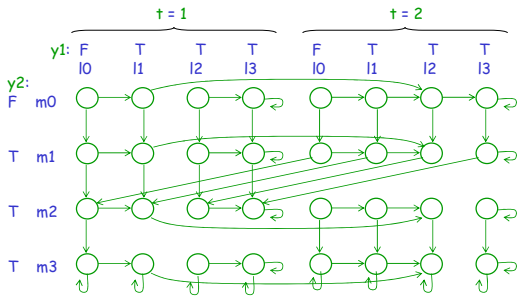
Post-order is order of popping states in DFS

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

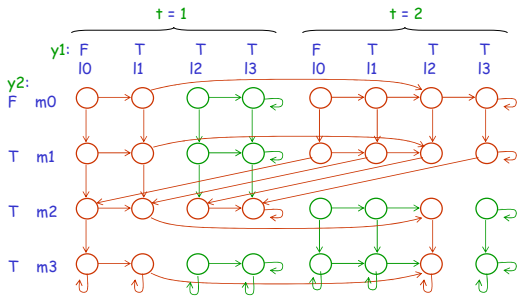
Hence if state reached from qi has path to qj then qi is in cycle



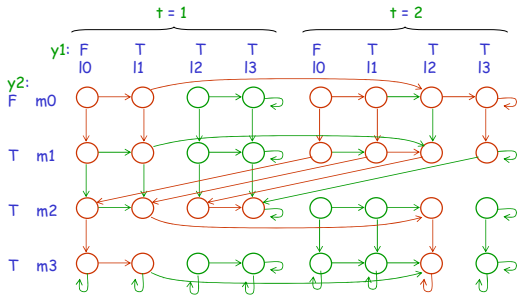
State-space exploration of PF



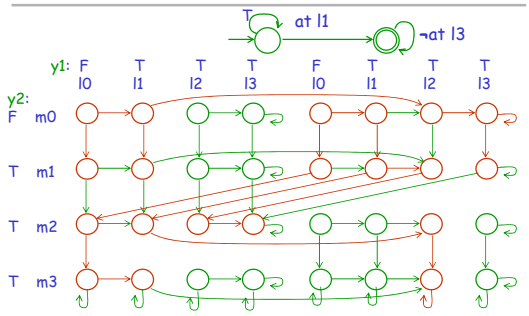
Reachable states



DFS tree

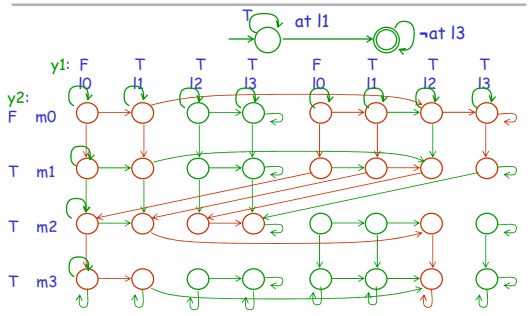


DFS tree with automaton



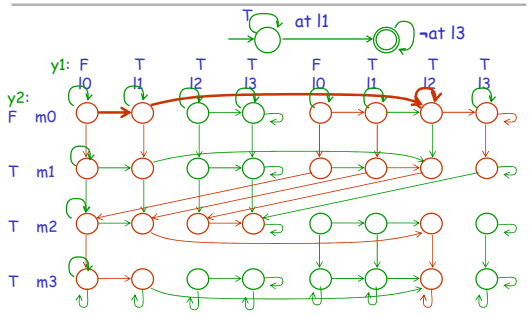
The graph has no loops that do not visit l3

Add self-loops at l0 and m0



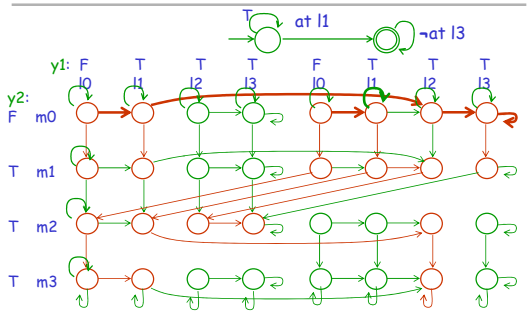
The graph has no loops that do not visit l3

Add self-loops at l0 and m0



The graph has a bad loop that does not visit l3

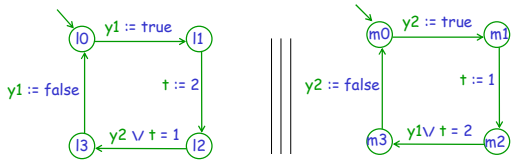
Add self-loops at l0 and m0



This is the first to be visited

Peterson-Fischer: Possible Specifications

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

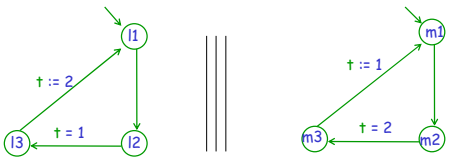


Mutual Exclusion: □ ¬(at l3 /\ at m3)

Absence of Starvation: □ (at l1 -> ◇ at l3)
Bounded Overtaking: □ (at l1 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

Simpler mutex

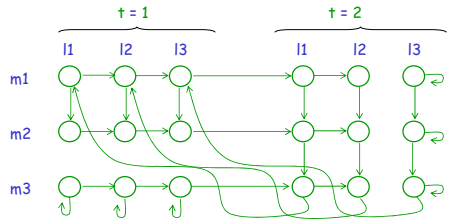
Variables: t: {1,2}
Initially t = 1



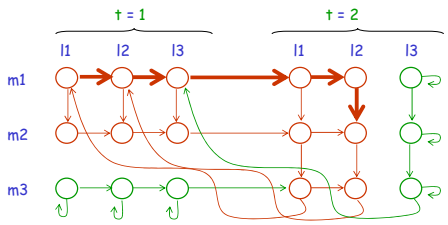
Mutual Exclusion: □ ¬(at l2 /\ at m3)

Absence of Starvation: □ (at l2 -> ◇ at l3)
Bounded Overtaking: □ (at l2 -> (¬at m3 U (at m3 U(¬at m3 U at l3)))

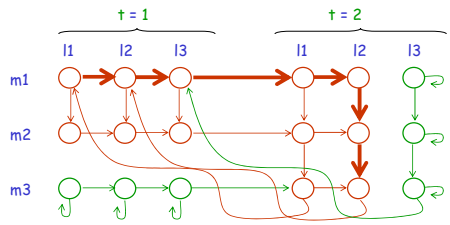
Simpler state space



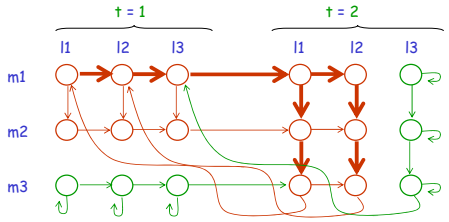
DFS tree



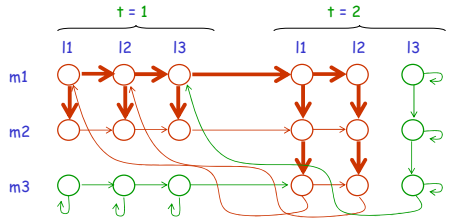
DFS tree



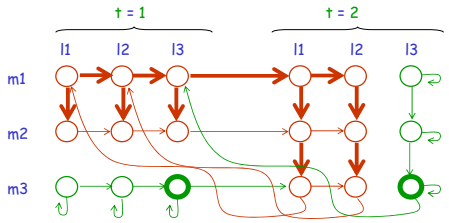
DFS tree



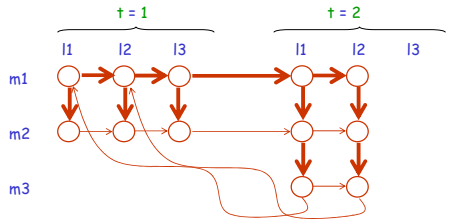
DFS tree



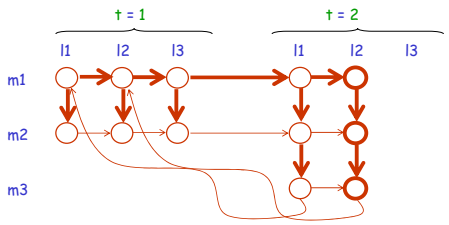
Bad states are unreachable



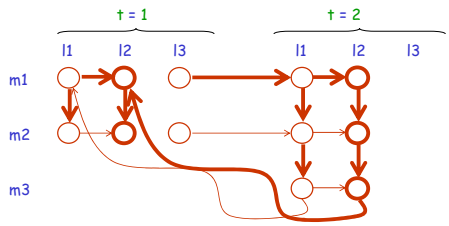
Reachable states



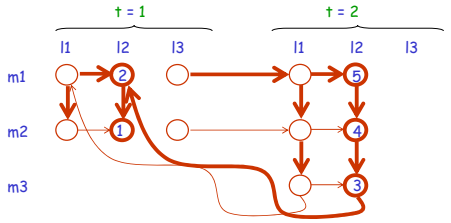
Accepting states for absence of starvation



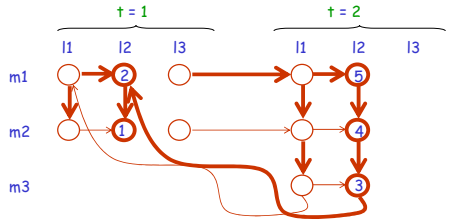
Accepting states for absence of starvation



Post-order



No loops can be found



Simpler mutex with self-loops

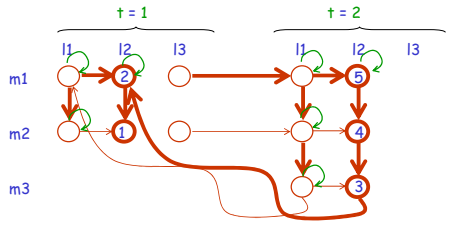
Variables: t: {1,2}
Initially t = 1

Mutual Exclusion: □ ¬(at l2 /\ at m3)

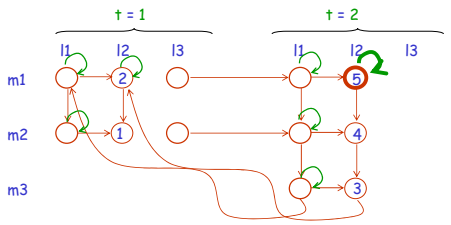
Absence of Starvation: □ (at l2 -> ◇ at l3)

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

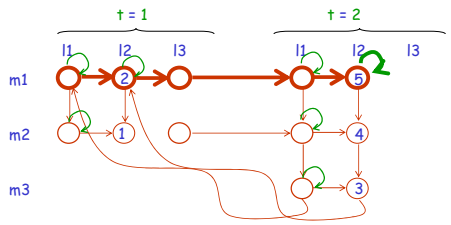
State space with self-loops



Idle loop found

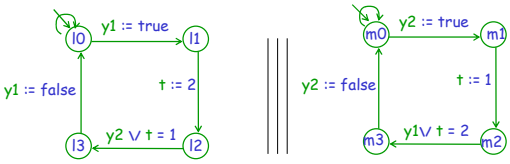


Corresponding bad reachable loop



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 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*

Checking fairness in exploration algorithm

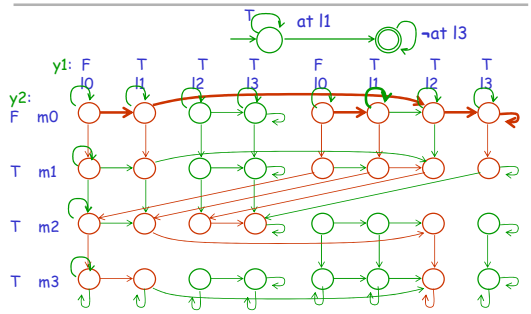
To verify φ under fairness assumption

- 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



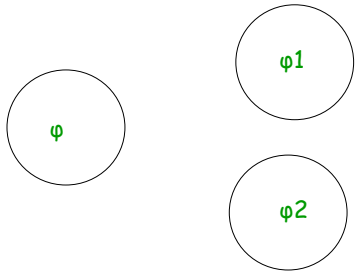
The loop does not have left process disabled or taken

Translating formulas to Automata

◊(p ∧ □¬q)

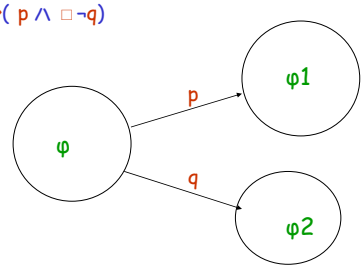
Translating formulas to Automata

◊(p ∧ □¬q)



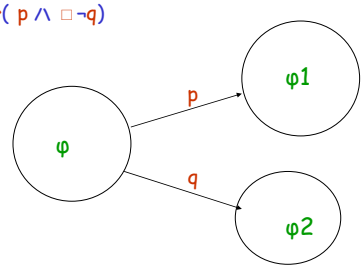
Translating formulas to Automata

◊(p ∧ □¬q)



Translating formulas to Automata

◊(p ∧ □¬q)



φ ≡ p ∧ ∘φ1 ∨ q ∧ ∘φ2

Decomposing temporal operators

Pushing negations

¬ □ φ ≡ ◊ ¬φ φ ∧ ∘ □ φ

¬◊ φ ≡ □ ¬φ ∨ ∘ ◊ φ

¬ (φ U ψ) ≡ ¬ψ W (¬φ ∧ ¬ψ)

¬ (φ W ψ) ≡ ¬ψ U (¬φ ∧ ¬ψ)

Putting on Normal form

□ φ ≡ φ ∧ ∘ □ φ

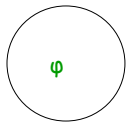
◊ φ ≡ φ ∨ ∘ ◊ φ

φ U ψ ≡ ψ ∨ (φ ∧ ∘ φ U ψ)

φ W ψ ≡ ψ ∨ (φ ∧ ∘ φ W ψ)

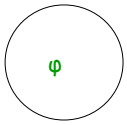
Translating formulas to Automata

φ ≡ ◊(p ∧ □¬q)



Translating formulas to Automata

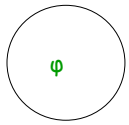
$\varphi \equiv \Diamond(p \wedge \Box\neg q) \equiv (p \wedge \Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$



Translating formulas to Automata

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

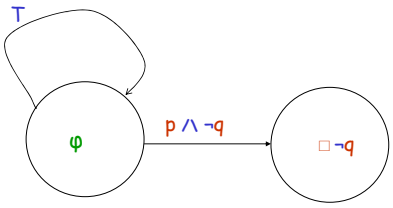
$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$



Translating formulas to Automata

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

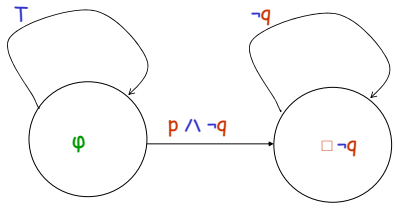
$\equiv (p \wedge \neg q \wedge \circ\Box\neg q) \vee \circ \Diamond(p \wedge \Box\neg q)$



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

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

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