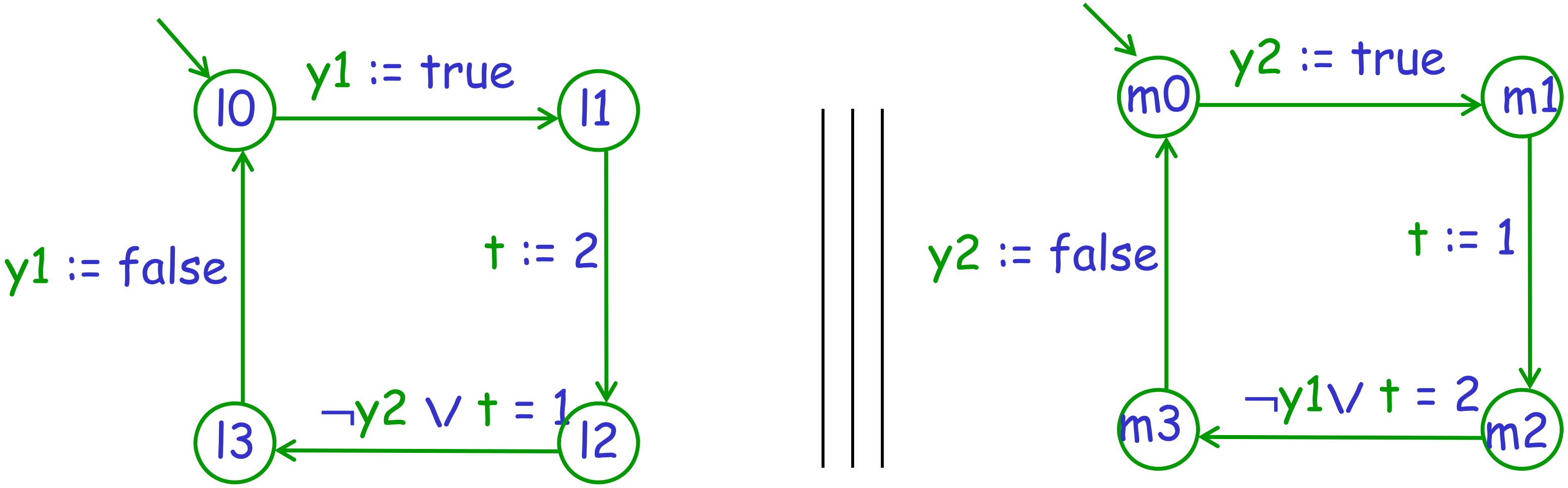
Lecture 6

Liveness and 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

Specifying progress:

Idea: Specify control states that should occur infinitely often,

- Typically, what can happen in infinite loops.

In Promela:

- **`progress`** label should be visited infinitely often
- **`accept`** label should not be visited infinitely often

Peterson-Fischer Mutual Exclusion

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l0: y1 = true;
l1: t = turn2;
l2: (y2 == false ||  t == turn1) ;
    mutex++ ;
    assert (mutex <= 1) ;
l3:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l0 }
}
```

```
proctype P2() {
m0: y2 = true;
m1: t = turn1;
m2: (y1 == false ||  t == turn2) ;
    mutex++ ;
    assert (mutex <= 1) ;
m3:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m0 }
}



init {
  atomic { run P1() ; run P2() }
}
```

Attempt at specifying progress

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l0: y1 = true;
l1: t = turn2;
l2: (y2 == false ||  t == turn1) ;
    mutex++ ;
    assert (mutex <= 1) ;
progress3:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l0 }
}
```

```
proctype P2() {
m0: y2 = true;
m1: t = turn1;
m2: (y1 == false ||  t == turn2) ;
    mutex++ ;
    assert (mutex <= 1) ;
m3:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m0 }
}




init {
  atomic { run P1() ; run P2() }
}
```

What about fairness?

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
l0: do :: skip
       :: y1 = true -> break
    od ;
l1: t = turn2;
l2: (y2 == false ||  t == turn1) ;
    mutex++ ;
    assert (mutex <= 1) ;
progress3:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l0 }
}
```

```
proctype P2() {
m0: y2 = true;
m1: t = turn1;
m2: (y1 == false ||  t == turn2) ;
    mutex++ ;
    assert (mutex <= 1) ;
m3:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m0 }
}




init {
  atomic { run P1() ; run P2() }
}
```

What about fairness?

```
#define true    1
#define false   0
#define turn1   false
#define turn2   true

bool   y1, y2, t;
byte   mutex = 0;

proctype P1() {
progress0: do :: skip
              :: y1 = true -> break
    od ;
l1: t = turn2;
l2: (y2 == false ||  t == turn1) ;
    mutex++ ;
    assert (mutex <= 1) ;
progress3:  /* critical section */
    mutex -- ;
    atomic{ y1 = false ; goto l0 }
}
```

```
proctype P2() {
m0: y2 = true;
m1: t = turn1;
m2: (y1 == false ||  t == turn2) ;
    mutex++ ;
    assert (mutex <= 1) ;
m3:  /* critical section */
    mutex-- ;
    atomic{ y2 = false ; goto m0 }
}



init {
  atomic { run P1() ; run P2() }
}
```

AB Protocol (version 5, limiting checks)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: data < 10 -> s_r ! msg(data,
    sbit)
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
  progress:   assert(recd == expected) ;
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> skip
    od
}
```

AB Protocol (version 6, w progress)

```
mtype    = { msg, ack };
chan s_r = [2] of {mtype , byte, bit};
chan r_s = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    do
    :: data < 10 -> s_r ! msg(data,
    sbit)
    :: (1) -> progress1: skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  data++
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd, expected = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                  rbit = 1 - rbit ;
  progress:   assert(recd == expected) ;
                  expected++
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> progress2: skip
    od
}
```

AB Protocol (version 7, w progress)

```
# define MAXMSG 4

mtype       = { msg, ack };
chan s_r    = [2] of {mtype , byte,
    bit};
chan r_s    = [2] of {mtype , bit };
chan source = [0] of {byte};
chan sink   = [0] of {byte};

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    source?data;
    do
    :: s_r ! msg(data, sbit)
    :: (1) -> progress1: skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                  sbit = 1 - sbit ;
                  source?data
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                sink!recd;
                progress: rbit = 1 - rbit
           :: else
           fi
    :: r_s ! ack (rbit)
    :: (1) -> progress2: skip
    od
}
```

Linear Time Temporal Logic

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

  $\circ\ \varphi$ "in the next state φ"

  $\square\ \varphi$ "always (in all future states) φ"

  $\lozenge\ \varphi$ "sometimes (in some future states) φ"

  $\varphi\ \mathsf{U}\ \psi$ "φ until ψ"

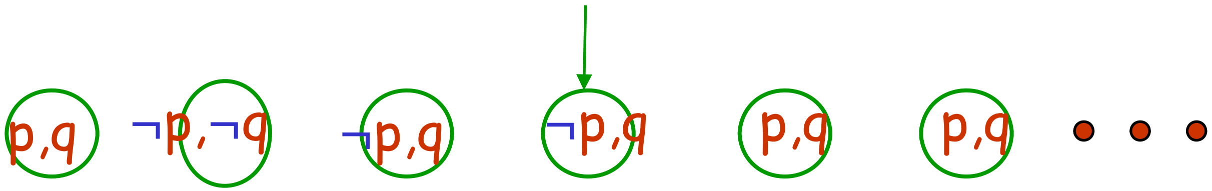
  $\varphi\ \mathsf{W}\ \psi$ "φ unless ψ"

- 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

$\neg p \wedge q$

$\circ (p \wedge q)$

- The following formula does not hold at the particular state

$\Box\, p$

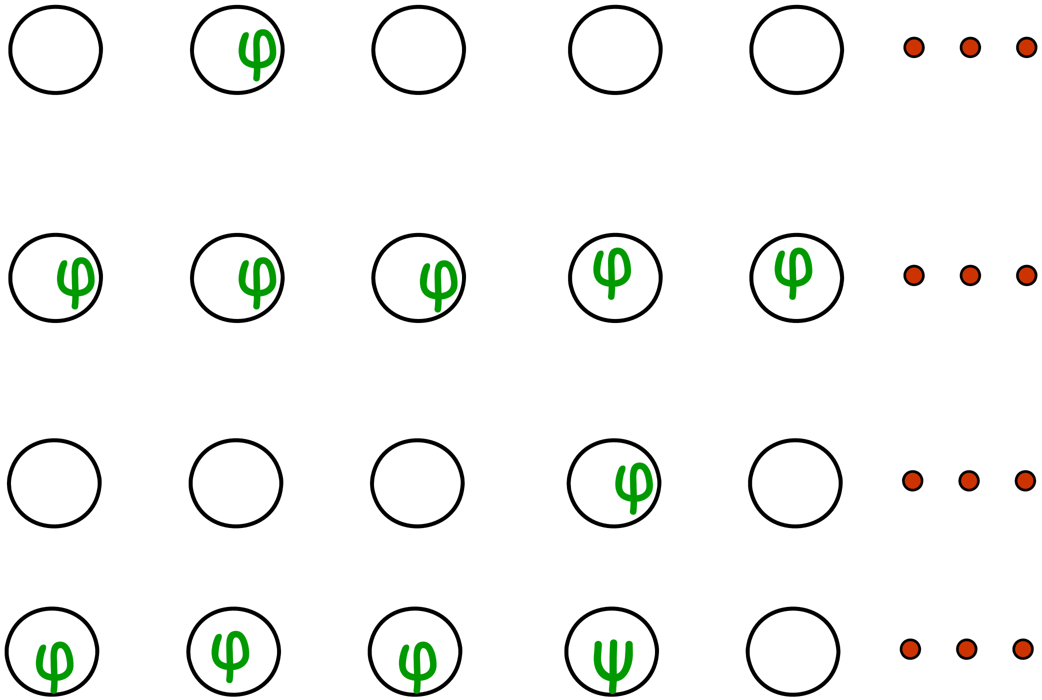
Meaning of operators

$\circ \varphi$

$\Box \varphi$

$\Diamond \varphi$

$\varphi \, U \, \psi$



$\varphi \, W \, \psi$ $\qquad (\varphi \, U \, \psi \;\lor\; \Box \varphi)$

Operators: Formal semantics

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

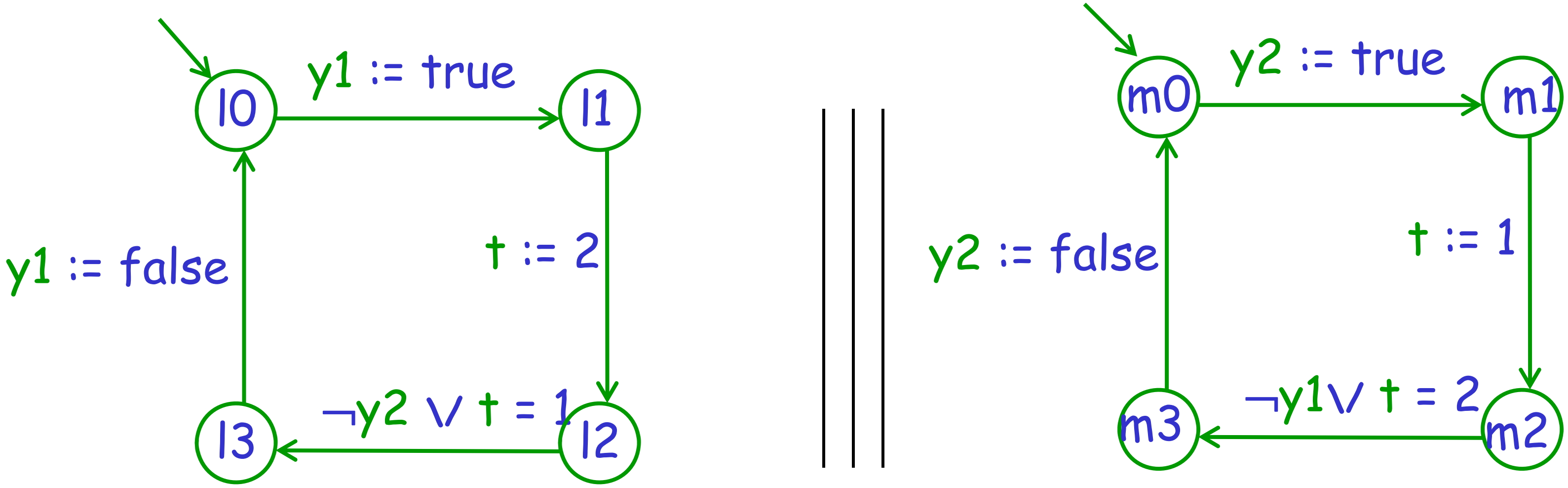
- (ß,i) |= φ denotes that φ is true in state s_i
- (ß,i) |= p (for p state formula) iff p holds in state s_i
- Boolean connectives work as usual
- (ß,i) |= $\circ\ \varphi$ iff (ß,i+1) |= φ
- (ß,i) |= $\Box\ \varphi$ iff $\forall j \geq i$ (ß,j) |= φ
- (ß,i) |= $\Diamond\ \varphi$ iff $\exists j \geq i$ (ß,j) |= φ
- (ß,i) |= φ U ψ iff $\exists j \geq i$ (ß,j) |= ψ and
  $\forall k : i \leq k < j$:(ß,k) |= φ
- (ß,i) |= φW ψ iff (ß,i) |= φ U ψ or (ß,i) |= $\Box\ \varphi$

Linear Temporal logic: examples

$\Box\ p$	p is invariant
$\Box\ (p \rightarrow \Box p)$	p is stable
$\neg\ q\ \mathbf{W}\ (p \wedge \neg q)$	p precedes q
$\Box (p \rightarrow \Diamond\ q)$	p leads to q
$\Box \Diamond\ p$	infinitely often p
$\Diamond \Box\ p$	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)))

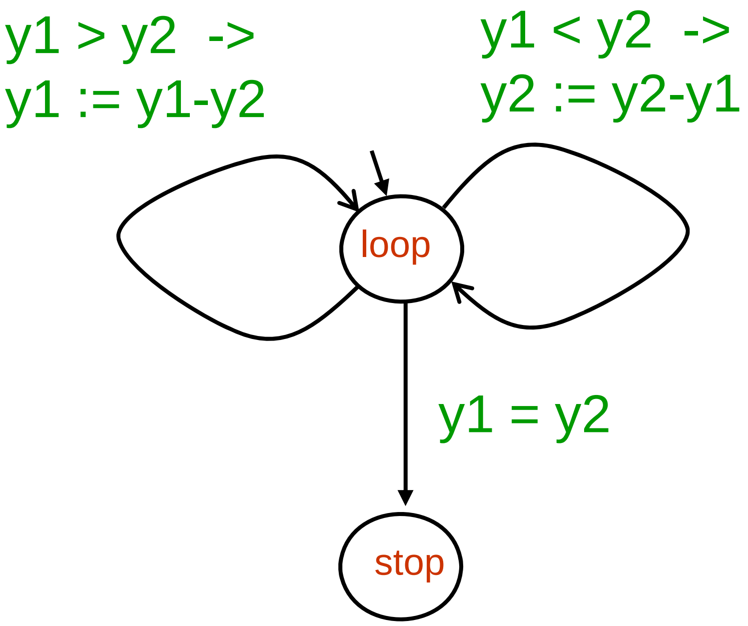
Example: GCD Computation

Action System

Variables: y1,y2: integer

Initially   y1=x1, y2=x2

P:



y1 > y2 ->
y1 := y1-y2

y1 < y2 ->
y2 := y2-y1

y1 = y2

Partial Correctness:

x1> 0 /\ x2> 0 ->

□[P@stop -> y1=y2=gcd(x1,x2)]

Total Correctness:

x1> 0 /\ x2> 0 ->

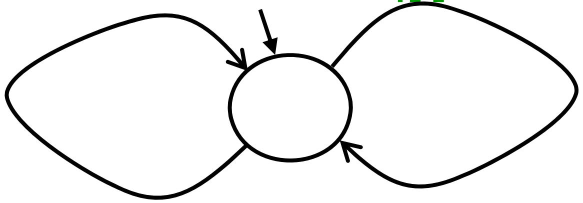
[P@loop -> ◇ P@stop]

Example: Termination Detection

Variables: ch[N]: FIFO Channel of {black,white}
q[N]: boolean
dist = false: boolean

Processes: Q[0] || ... || Q[N-1] || P[0] || ... || P[N-1]

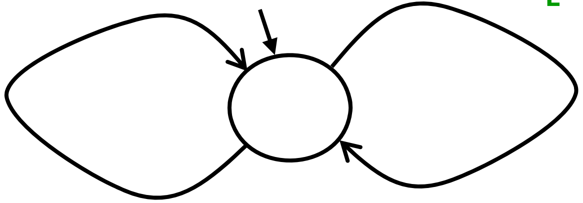
Q[i]:



q[i] := true

q[i-1] -> q[i] := false ;
if i=0 then dist := true

P[i]:



ch[i]?white -> if q[i] then ch[i+1]!white
else ch[i+1]!black

ch[i]?black -> ch[i+1]!black

P[0]:

ch[1]!white ; dist := false

ch[0]?black

ch[0]?white

¬dist

dist

Example: Termination Detection

Abbreviation:

$\text{Terminated} \equiv \forall i : 0 \le i < N :: q[i]$

Safety Property:

$\Box[\text{P[0]@term} \rightarrow \text{Terminated}]$

Liveness Property:

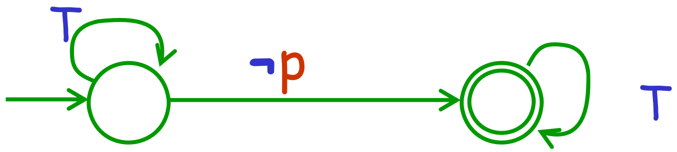
$\text{Terminated} \rightarrow \Diamond\ \text{P[0]@term}$

Specifying linear properties by automata

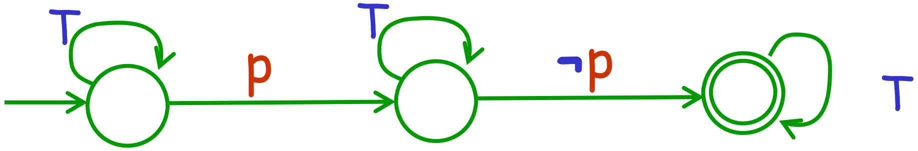
- A linear time property is a set of computations.
- Can be seen as a language of (infinite) words,
  - Alphabet: possible assignments of truth-values to occurring state formulas
- A language can be specified by an automaton
- It turns out that it is better to represent the complement, i.e., the set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

Automata Specifications: examples

- $\Box\ p$



- $\Box\ (\ p \rightarrow \Box p\ )$



- $\neg\ q\ W\ (p\ \wedge\ \neg q)$

- $\Box(\ p \rightarrow \Diamond\ q)$

?

- $\Box \Diamond\ p$
- $\Diamond \Box\ p$

Superposing Automaton on Transition system

Superposition of Automaton $(\Sigma, Q, Q_0 , \rightarrow , \Phi)$
onto Transition System $(S, S_0 , \rightarrow , V , L)$ has

$S \times Q$ as the set of states

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

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 (finite) prefixes

Can be done by (standard) finite automaton

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

Automata over infinite words

Automaton over infinite words: $(\Sigma, Q, Q_0, \rightarrow, \Phi)$ 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
- $\rightarrow$ (a subset of $S \times \Sigma \times S$) 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

Büchi automata

One set of accepting states F

Acceptance condition $\Box\Diamond F$

Generalized Buchi automata

Collection of sets of accepting states $F_1, \ldots, F_n$

Acceptance condition $\Box\Diamond F_1 \wedge \ldots \wedge \Box\Diamond F_n$

Rabin automata

Collection of pairs of accepting states $(F_1, G_1), \ldots, (F_n, G_n)$

Acceptance condition $\bigvee_i (\Box\Diamond F_i \wedge \neg\Box\Diamond G_i)$

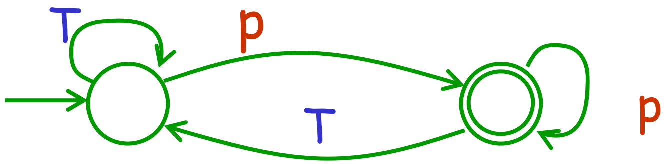
Streett automata

Collection of pairs of accepting states $(F_1, G_1), \ldots, (F_n, G_n)$

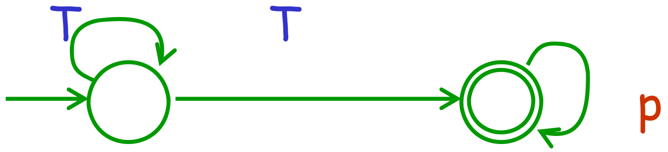
Acceptance condition $\bigwedge_i (\Box\Diamond F_i \rightarrow \Box\Diamond G_i)$

Examples of Büchi Automata

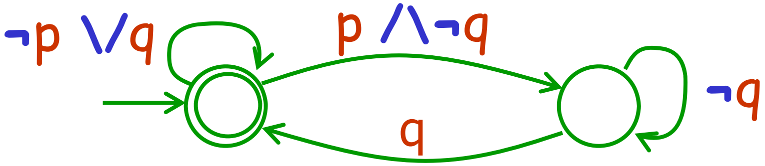
- $\Box \Diamond p$



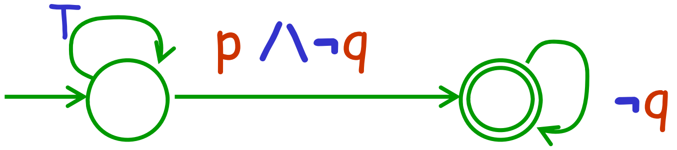
- $\Diamond \Box p$



- $\Box (p \rightarrow \Diamond q)$

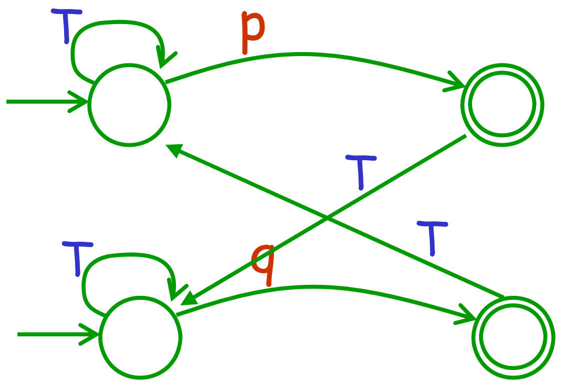


- $\Diamond (p \wedge \Box \neg q)$

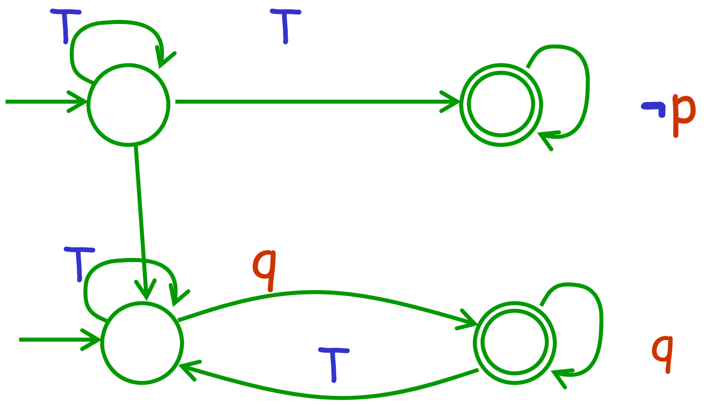


More Examples of Büchi Automata

- $(\Box\Diamond p \land \Box\Diamond q)$

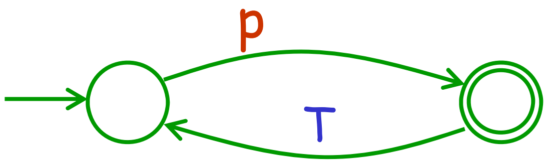


- $(\Box\Diamond p \rightarrow \Box\Diamond q)$



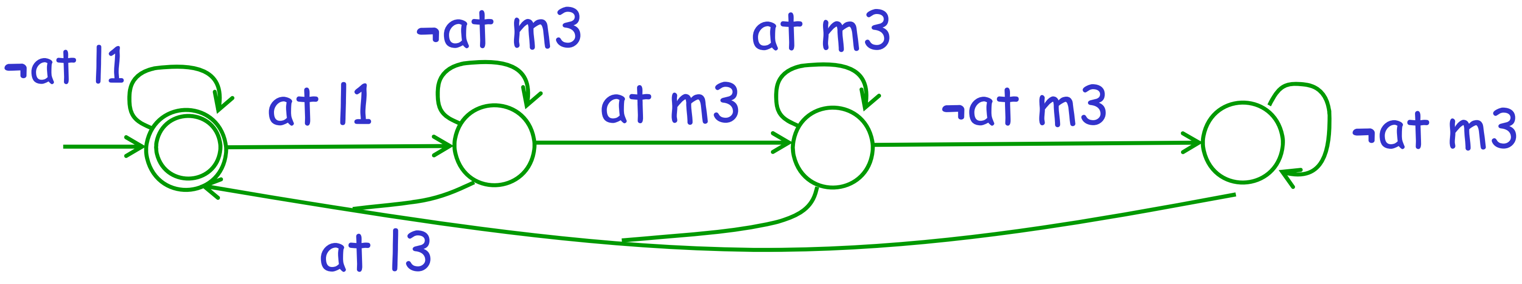
Translation Results

- 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”



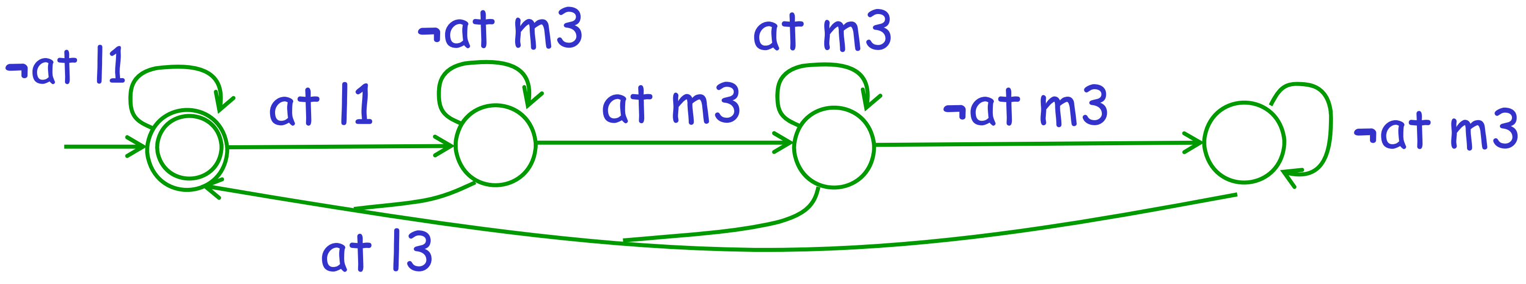
Convenience of expression

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

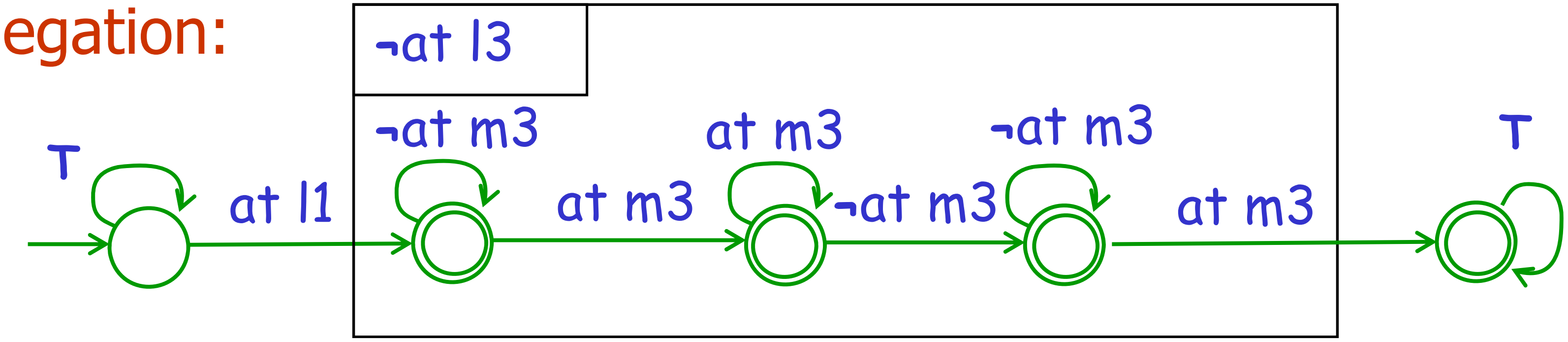


Convenience of expression

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

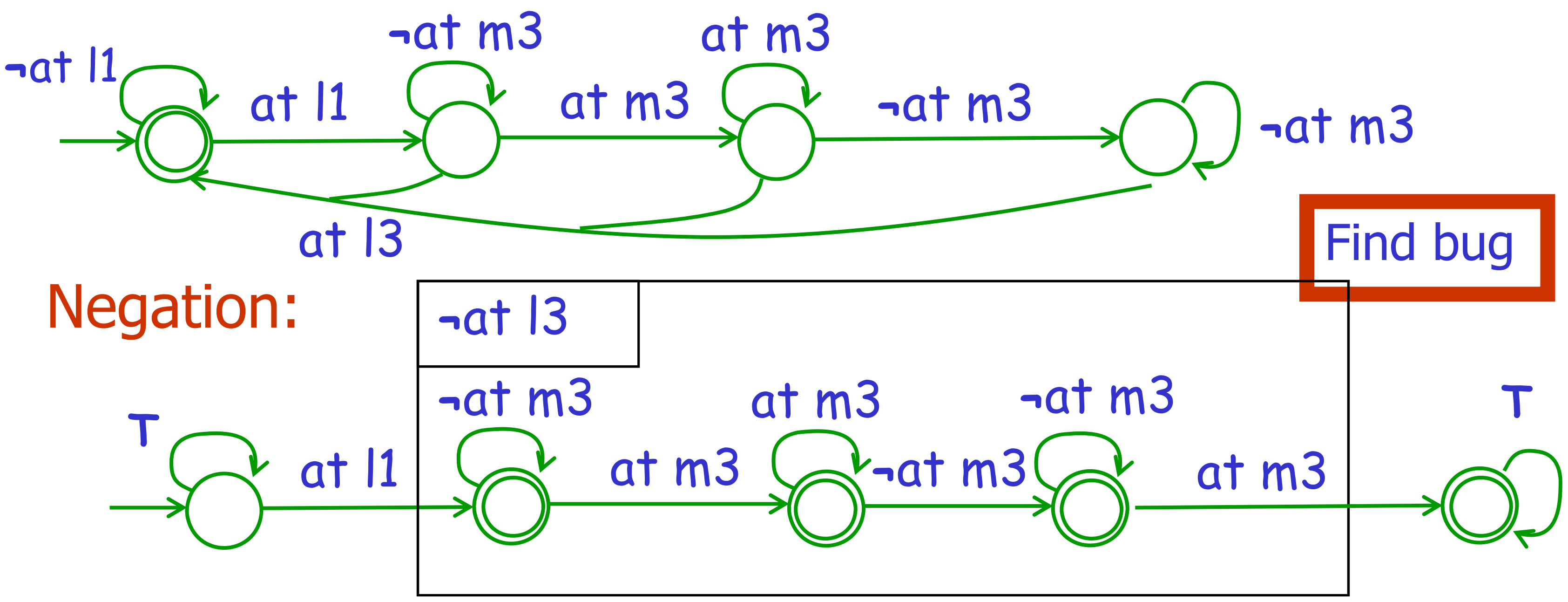


Negation:



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)))



Find bug

Negation:

AB Protocol (version 9, w acceptance)

```
#define p Sender@Slabel
#define q Receiver@Rlabel
#define r Receiver@Rsuccess

mtype       = { msg, ack };
chan s_r    = [2] of {mtype , byte,
   bit};
chan r_s    = [2] of {mtype , bit };

active proctype Sender() {
    byte data = 0;
    bit sbit, seqno = 0;
    source?data;
    do
    :: s_r ! msg(data, sbit) ->
    Slabel: skip
    :: (1) -> skip
    :: r_s ? ack(seqno);
           if
           :: seqno == sbit ->
                 sbit = 1 - sbit ;
                 source?data
           :: else
           fi
    od
}
```

```
active proctype Receiver() {
    byte recd = 0;
    bit rbit = 1, seqno;
    do
    :: s_r ? msg (recd, seqno) ->
           if
           :: seqno != rbit ->
                Rsuccess: sink!recd;
                rbit = 1 - rbit
           :: else
           fi
    :: r_s ! ack (rbit) -> Rlabel: skip
    :: (1) -> skip
    od
}
```

AB Protocol (version 9, the never claim)

```
#define p Sender@Slabel
#define q Receiver@Rlabel
#define r Receiver@Rsuccess

/*
     * Formula As Typed: ([]  <>  p &&
    []  <>  q) -> []  <>  r
     * The Never Claim Below
    Corresponds
     * To The Negated Formula !(([]  <>
    p && []  <>  q) -> []  <>  r)
     * (formalizing violations of the
    original)
     */

never {    /* !(([]  <>  p && []  <>
    q) -> []  <>  r) */
```

```
T0_init:
    if
    :: (! ((r)) && (p) && (q)) -> goto
    accept_S485
    :: (! ((r)) && (p)) -> goto T2_S485
    :: (! ((r))) -> goto T0_S485
    :: (1) -> goto T0_init
    fi;
accept_S485:
    if
    :: (! ((r))) -> goto T0_S485
    fi;
T2_S485:
    if
    :: (! ((r)) && (q)) -> goto
    accept_S485
    :: (! ((r))) -> goto T2_S485
    fi;
T0_S485:
    if
    :: (! ((r)) && (p) && (q)) -> goto
    accept_S485
    :: (! ((r)) && (p)) -> goto T2_S485
    :: (! ((r))) -> goto T0_S485
    fi;
}
```