Lecture 6

Liveness and Linear Time Temporal Logic
Peterson-Fischer: Possible Specifications

Variables: \( y_1, y_2 \): boolean, \( t: \{1,2\} \)
Initially \( y_1 = y_2 = false, \ t = 1 \)

Mutual Exclusion: the two processes never simultaneously reach \( l_3, m_3 \)

Absence of Starvation: If the left process is at \( l_1 \), it will later reach \( l_3 \)

Bounded Overtaking: If the left process is at \( l_1 \), the other process will reach \( m_3 \) at most once (twice?) before the left process reaches \( l_3 \)
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

```c
#include <stdio.h>

#define true    1
#define false   0

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

proctype P1() {
  10: y1 = true;
  11: t = turn2;
  12: (y2 == false || t == turn1) ;
      mutex++ ;
      assert (mutex <= 1) ;
  13: /* critical section */
      mutex -- ;
      atomic{ y1 = false ; goto 10 } 
}

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

```c
#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() {
    do :: skip
    :: y1 = true -> break
    od ;
    t = turn2;
    l2: (y2 == false || t == turn1) ;
        mutex++ ;
        assert (mutex <= 1) ;

    l3: /* critical section */
        mutex-- ;
        atomic{ y2 = false ; goto m0 }
}

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?

```plaintext
#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 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(seqno);
          if :: seqno == sbit ->
            sbit = 1 - sbit;
            data++
            :: else
            fi
    od
}

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
}
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 Receiver() {  
    byte recd, expected = 0;  
    bit rbit = 1, seqno;  
    do  
        :: data < 10 -> s_r ! msg(data, sbit)  
        :: (1) -> progress1: skip  
        :: r_s ? msg (recd, seqno) ->  
        :: (1) -> progress1: skip  
        :: r_s ? ack(seqno);  
        if  
            :: seqno == sbit ->  
                sbit = 1 - sbit ;  
                data++  
            :: else  
                sbit = 1 - sbit ;  
                data++  
            :: elsefi  
    od  
}

active proctype Sender() {  
    byte data = 0;  
    bit sbit, seqno = 0;  
    do  
        :: s_r ? msg (recd, seqno) ->  
        if  
            :: seqno != rbit ->  
                rbit = 1 - rbit ;  
                seqno = 0;  
        :: elsefi  
    od  
}

progress: assert(recd == expected) ;  
expected++  
progress2: skip  

AB Protocol (version 7, w progress)

```c
#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 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
}

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
}
```

Linear Time Temporal Logic
Linear Time Temporal Logic: formulas

- **State formulas** (denoted by $p$, $q$): properties about states
  
  e.g., $y_0 < 1 \quad x = y \quad a \uparrow m3$

- **Formulas built using temporal operators**
  
  - $\circ \varphi$ “in the next state $\varphi$”
  
  - $\square \varphi$ “always (in all future states) $\varphi$”
  
  - $\diamond \varphi$ “sometimes (in some future states) $\varphi$”
  
  - $\varphi \mathcal{U} \psi$ “$\varphi$ until $\psi$”
  
  - $\varphi \mathcal{W} \psi$ “$\varphi$ unless $\psi$”

- **And boolean connectives**
  
  $\neg \quad \lor \quad \land \quad \rightarrow$
Linear Time Temporal Logic: interpretation

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

  ![Diagram of states](image)

  - The following formulas hold at the particular state

    - $\lnot p \land q$
    - $(p \land q)$

  - The following formula does not hold at the particular state

    - $\Box p$
Meaning of operators

- $\circ \varphi$
- $\Box \varphi$
- $\Diamond \varphi$
- $\varphi \cup \psi$
- $\varphi \wedge \psi$ (if $\varphi \cup \psi \lor \Box \varphi$)
Operators: Formal semantics

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

• $(\beta, i) \models \varphi$ denotes that $\varphi$ is true in state $s_i$

• $(\beta, i) \models p$ (for $p$ state formula) iff $p$ holds in state $s_i$

• Boolean connectives work as usual

• $(\beta, i) \models \Diamond \varphi$ iff $\exists j \geq i (\beta, j) \models \varphi$

• $(\beta, i) \models \Box \varphi$ iff $\forall j \geq i (\beta, j) \models \varphi$

• $(\beta, i) \models \varphi \cup \psi$ iff $\exists j \geq i (\beta, j) \models \psi$ and $\forall k : i \leq k < j : (\beta, k) \models \varphi$

• $(\beta, i) \models \varphi \mathcal{W} \psi$ iff $(\beta, i) \models \varphi \cup \psi$ or $(\beta, i) \models \Box \varphi$
Linear Temporal logic: examples

• $\Box p$  $p$ is invariant
• $\Box ( p \rightarrow \Box p )$  $p$ is stable
• $\neg q \ W ( p \land \neg q )$  $p$ 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: $y_1, y_2$: boolean, $t$: {1,2}
Initially $y_1 = y_2 = false, t = 1$

Mutual Exclusion: $\square \neg (at l3 \land at m3)$

Absence of Starvation: $\square (at l1 \rightarrow \Diamond at l3 )$

Bounded Overtaking: $\square (at l1 \rightarrow (\neg at m3 U (at m3 U(\neg at m3 U at l3 )))$
Example: GCD Computation

Action System
Variables: $y_1, y_2$: integer
Initially $y_1 = x_1, y_2 = x_2$

P:

\[ y_1 > y_2 \rightarrow y_1 := y_1 - y_2 \]
\[ y_2 > y_1 \rightarrow y_2 := y_2 - y_1 \]

Partial Correctness:
\[ x_1 > 0 \land x_2 > 0 \rightarrow □[P@stop \rightarrow y_1 = y_2 = \text{gcd}(x_1, x_2)] \]

Total Correctness:
\[ x_1 > 0 \land x_2 > 0 \rightarrow [P@loop \rightarrow \Diamond P@stop] \]
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]?white

ch[0]?black
dist

~dist

term
Example: Termination Detection

Abbreviation:

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

Safety Property:

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

Liveness Property:

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

• □ p

• □ ( p -> □ p )

• ¬ q W (p \(\land\) ¬q)

• □ ( p -> ◊ q)

• □ ◊ p

• ◊ □ 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

\[\text{S} \times \text{Q}\] as the set of states

\(<s,q> \rightarrow <s',q'>\] if \(s \rightarrow s'\) and \(q \rightarrow q'\)

Accepting condition adapted from \(\Phi\)
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,
- $\Sigma$ 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
- $\Phi$ is an acceptance condition

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

$$a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$$

$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow q_5 \rightarrow q_6 \rightarrow q_7 \rightarrow q_8$

which satisfies the acceptance condition $\Phi$
Classes of acceptance conditions

Büchi automata

One set of accepting states $F$
Acceptance condition $\Box \Diamond F$

Generalized Büchi automata

Collection of sets of accepting states $F_1, \ldots, F_n$
Acceptance condition $\Box \Diamond F_1 \land \ldots \land \Box \Diamond F_n$

Rabin automata

Collection of pairs of accepting states $(F_1, G_1), \ldots, (F_n, G_n)$
Acceptance condition $\forall_i (\Box \Diamond F_i \land \neg \Box \Diamond G_i)$

Streett automata

Collection of pairs of accepting states $(F_1, G_1), \ldots, (F_n, G_n)$
Acceptance condition $\land_i (\Box \Diamond F_i \rightarrow \Box \Diamond G_i)$
Examples of Büchi Automata

- □ ◊ p

- ◊ □ p

- □ ( p → ◊ q)

- ◊ ( p ∧ □ ¬q)
More Examples of Büchi Automata

• $(\square \Diamond p \land \square \Diamond q)$

• $(\square \Diamond p \rightarrow \square \Diamond q)$
Translation Results

• Any property expressible in linear temporal logic is accepted by some Buchi automaton

• There are properties accepted by some Buchi automaton that cannot 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:**
  \[ (\text{at } l1 \rightarrow (\neg \text{at } m3 \cup (\text{at } m3 \cup (\neg \text{at } m3 \cup \text{at } l3 )))) \]
Convenience of expression

- Some properties are easier to understand as automata
- **Bounded Overtaking:**
  \[(\text{at } l_1 \rightarrow (\neg \text{at } m_3 \cup (\text{at } m_3 \cup (\neg \text{at } m_3 \cup \text{at } l_3 )))\]

**Negation:**
Convenience of expression

- Some properties are easier to understand as automata
- Bounded Overtaking:
  \[ (\neg \text{at } l_1 \rightarrow (\neg \text{at } m_3 \cup (\text{at } m_3 \cup (\neg \text{at } m_3 \cup \text{at } l_3 ))) \]
AB Protocol (version 9, with 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 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 (seqno) ->
                :: r_s ! ack (seqno) ->
                    Rlabel: skip
                    :: (1) -> skip
                :: r_s ? ack (seqno);
                    if
                        :: seqno == sbit ->
                            sbit = 1 - sbit;
                            source?data
                        :: else
                            fi
                    od
    od
}
```

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
```
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;

}