AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Test of Distributed Systems
Lecture 3

Temporal Logic:
LTL
Reasoning
Model-checking

Today's lecture

- Time
- Linear temporal logic (LTL)
- Abstraction of time: Temporal operators
- Semantics: LTL and computation sequences
- Specifying and verifying with LTL
- LTL Patterns
- Logic

Modeling discrete time

A file that has been requested to be printed is eventually printed:

- Explicit time:

  $\forall f \forall t_1.\ \text{reqprint}(f,t_1) \longrightarrow \exists t_2.\ t_2 \geq t_1 \wedge \text{print}(f,t_2)$

- Implicit time:

  $\forall f.\ \Box(\text{reqprint}(f) \longrightarrow \Diamond \text{print}(f))$

Syntax of LTL

Always: □

"for any time t in the future"

Eventually: ◊

"for some time t in the future"

LTL is predicate logic plus formulas prefixed by □ or ◊, e.g.,

□(c≤1), □◊(flag), □□◊□◊◊(exc = w)

SPIN Syntax for LTL

- not: !,
  and: &&,
  or: ||,
  implies: ->,
  equivalent: <->
- always: [],
  eventually <>,
  until: U
- *Precedence? Use brackets for clarity!*

Semantics of Always

- Let A be an LTL formula and c=(s_0,s_1,s_2,…) be a computation.
  Then
  []A is true in state s_i
  if and only if
  A is true ***for all*** s_j in c such that j≥i.

Semantics of Eventually

- Let A be an LTL formula and c=(s_0,s_1,s_2,…) be a computation.
  Then
  <>A is true in state s_i
  if and only if
  A is true ***for some*** s_j in c such that j≥i.

Semantics of Until

- Let A be an LTL formula and $c=(s_0,s_1,s_2,\ldots)$ be a computation.
  Then
  B U A is true in state s_i
  if and only if
  A is true ***for some*** s_k in c such that $k \geq i$
  and
  B is true in s_j ***for all*** s_j in c such that $i \leq j < k$

Semantics of Next

- Let A be an LTL formula and $c=(s_0,s_1,s_2,\ldots)$ be a computation.
  Then
  X A is true in state s_i
  if and only if
  A is true ***for*** s_{i+1}
- Do ***not*** use next in specifications (→ stutter invariance)
- It is only useful in rare occasions

Semantics of LTL

- Let A be an LTL formula and c=(s_0,s_1,s_2,…) be a computation.
  Then
  A is true
  if and only if
  A is true ***for*** s_0
- A **Promela program** describes a set of computations C: *A is true for C if and only if it is true for all computations contained in C*

Safety and Liveness

- A safety property states a condition that should never happen: "nothing bad ever happens". It is of the form []A.

- A liveness property states a condition that should eventually happen: "something good eventually happens".
  It is of the form <>A.

Mutual exclusion

- Is a safety property: (jspin: safety mode!)
  [] “at most one process in its critical section”

```
1.  #define excC []!(csP && csQ)
2.
3.  bool wantP = false, wantQ = false;
4.  bool csP = false, csQ = false;
5.
6.  active proctype P() {
7.    do
8.    :: wantP = true;
9.       !wantQ;
10.      csP = true;
11.      csP = false;
12.      wantP = false
13.   od
14. }
15.
16. active proctype Q() {
17.   do
18.   :: wantQ = true;
19.      !wantP;
20.      csQ = true;
21.      csQ = false;
22.      wantQ = false
23.   od
24. }
25.
26. ltl exc { excC }
```

Array Index Bounds

- Let a be an array, LEN its length, i be a variable to index the array.
- The following should hold: “In every state the index variable i should be in the range $0 \leq i < LEN$”
- ltl index { [] $0 \leq i < LEN$ }

Absence of Deadlock

- Is a liveness property: (jspin: acceptance mode!)
  "Eventually some (waiting) process enters its critical section"

```
1.   #define dlkC []<>(!wantP || !wantQ)      or      #define dlkC !<>[](wantP && wantQ)

2.   bool wantP = false, wantQ = false;

3.   active proctype P() {
4.     do
5.     :: wantP = true;
6.        !wantQ;
7.        wantP = false
8.     od
9.   }

10.  active proctype Q() {
11.    do
12.    :: wantQ = true;
13.       !wantP;
14.       wantQ = false
15.    od
16.  }

17.  ltl dlk { dlkC }
```

Absence of Starvation

- Is a liveness property: (jspin: acceptance mode!)

  "Eventually each (waiting) process enters its critical section"

```
1.  #define stvC <>csP && <>csQ
2.
3.  bool wantP = false, wantQ = false;
4.  bool csP = false, csQ = false;
5.
6.  active proctype P() {
7.    do
8.    :: wantP = true;
9.       csP = true;
10.      csP = false;
11.      wantP = false
12.   od
13. }
14.
```

```
15. active proctype Q() {
16.   do
17.   :: wantQ = true;
18.      csQ = true;
19.      csQ = false;
20.      wantQ = false
21.   od
22. }
23.
24. ltl stv { stvC }
```

Does not hold in absence of weak fairness!

Weak Fairness

- A computation is **weakly fair** if and only if the following condition holds:

  if a statement is ***always*** executable, then it is ***eventually*** executed as part of the computation.

Absence of Starvation

- Is a liveness property: (jspin: acceptance mode!)

  "Eventually each (waiting) process enters its critical section"

```
1.   #define stvC <>csP && <>csQ
2.
3.   bool wantP = false, wantQ = false;
4.   bool csP = false, csQ = false;
5.
6.   active proctype P() {
7.     do
8.     :: wantP = true;
9.        csP = true;
10.       csP = false;
11.       wantP = false
12.    od
13.  }
14.
15.  active proctype Q() {
16.    do
17.    :: wantQ = true;
18.       csQ = true;
19.       csQ = false;
20.       wantQ = false
21.    od
22.  }
23.
24.  ltl stv { stvC }
```

Does hold in presence of weak fairness!

Absence of Starvation

- Is a liveness property: (jspin: acceptance mode!)

  "Eventually each (waiting) process enters its critical section"

```
1.   #define stvC <>csP && <>csQ
2.
3.   bool wantP = false, wantQ = false;
4.   bool csP = false, csQ = false;
5.
6.   active proctype P() {
7.     do
8.     :: atomic { !wantQ; wantP = true }
9.       csP = true;
10.      csP = false;
11.      wantP = false
12.    od
13.  }
14.
15.  active proctype Q() {
16.    do
17.    :: atomic { !wantP; wantQ = true }
18.      csQ = true;
19.      csQ = false;
20.      wantQ = false
21.    od
22.  }
23.
24.  ltl stv { stvC }
```

Does not hold in the presence of weak fairness!
Due to the guards the atomic statements are not **always** enabled!

Warning on Safety Properties

```
bool wantP = false, wantQ = false;
bool p = false, q = false;

active proctype P() {
sp:
  do
  :: p = true;
     p = false;
     goto sp;
     wantP = true;
     wantP = false
  od
}
```

```
active proctype Q() {
sq:
  do
  :: q = true;
     q = false;
     goto sq;
     wantQ = true;
     wantQ = false
  od
}
```

ltl mutex { []!(wantP && wantQ)}

Additional liveness properties needed!

Warning on Termination

```
int n = 0;
bool flag = false;

active proctype P() {
  do
  :: flag -> break
  :: else -> n = 1-n
  od
}

active proctype Q() {
 flag = true
}
```

Not fair: ✗, fair: ✓

```
int n = 0;
bool flag = false;

active proctype P() {
  do
  :: flag -> break
  :: else -> n = 1-n
  od
}

active proctype Q() {
 do :: flag = !flag od
}
```

Not fair: ✗, fair: ✗

LTL specification patterns

- Invariance: []p
- Response: [](p -> <>q)
- Strong precedence: (!p) U q
- Weak precedence: <>p => (!p U q)
- True before: <>q -> (p U q)
- Latching: <>[]p
- Infinitely often: []<>p
- *http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml*

Overtaking

```
#define ptry P@try
#define qcs Q@cs
#define pcs P@cs

bool wantP, wantQ;
byte last = 1;

active proctype P() {
  do
  :: wantP = true;
     last = 1;
try: (wantQ == false) || (last == 2);
 cs: wantP = false
  od
}
```

```
active proctype Q() {
  do
  :: wantQ = true;
     last = 2;
try: (wantP == false) || (last == 1);
  cs: wantQ = false
  od
}

ltl otk {
  [](ptry -> (!qcs U (qcs U (!qcs U pcs))))
}
```