Chapter 14

Logical Properties

Logical Properties

Concepts: modeling properties that refer to states

Models: **fluent** – characterization of abstract state based on action sets
fluent linear temporal logic

Introduction

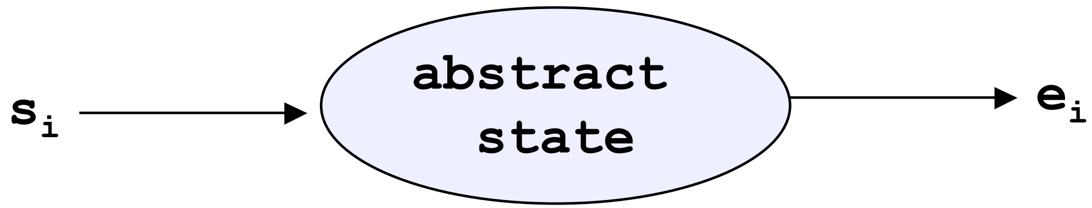
- Temporal logic due to Pneuli (1977) is a popular means to describe behavioural properties in logic.
- Use propositions to describe selected variable states at particular points in program executions.
- Realized as the **assert** construct in Java.
- Challenge: How to refer to states in an LTS model based on actions or events?
- Artifact: Introduce **fluents** to describe abstract state of LTS models.
- Express both safety and liveness properties in fluent propositions.

Pnueli, A. (1997). The Temporal Logic of Programs. Proc. of the 18th IEEE Symposium on the Foundations of Computer Science, Oct/Nov 1997, pp. 46-57.

Fluents

fluent FL = <{s_1,…,s_n},{e_1,…,e_n}> initially *B* defines a fluent FL that is initially true if the expression *B* is true and initially false if the expression *B* is false. FL becomes true when any of the initiating (or starting) actions {s_1,…,s_n} occur and false when any of the terminating (or ending) actions {e_1,…,e_n} occur. If the term **initially** *B* is omitted then FL is initially false. The same action may not be used as both an initiating and terminating action.

- A fluent **<{s_1,…,s_n},{e_1,…,e_n}>** describes an abstract state that is entered by executing any of the actions in {s_1,…,s_n}, and exited by executing any of the actions in {e_1,…,e_n}.



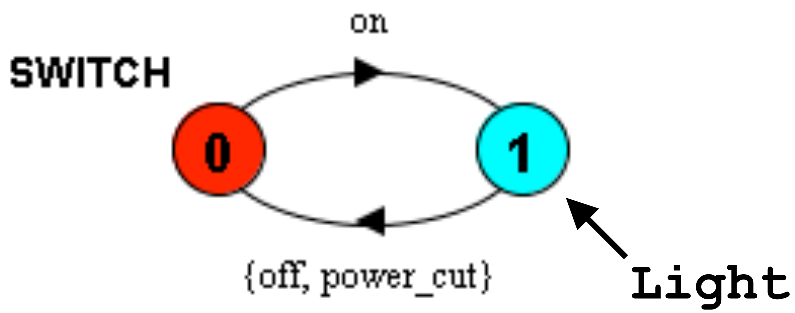
Fluents

```
const False = 0
const True = 1

fluent LIGHT = <{on},{off, power_cut}>
initially false
```

- SWITCH is initially not at state Light.

```
fluent DARK = <{off, power_cut},on>
initially true
```



Fluent Family

```
const False = 0
const True = 1

fluent LIGHT[i:1..2] = <on[i],{off[i], power_cut}>
```

is equivalent to defining two fluents:

```
const False = 0
const True = 1

fluent LIGHT1 = <on[1],{off[1], power_cut}>
fluent LIGHT2 = <on[2],{off[2], power_cut}>
```

Fluent Expressions

- Fluents can be composed using normal logical operators: &&, ||, !, ->, <->, forall, exists
- If the light is on, power is also on:

```
fluent LIGHT = <on,off>
fluent POWER = <power_on, power_off>
LIGHT -> POWER
```

- All lights are on:

```
forall[i:1..2] LIGHT[i]
```

- At least one light is on:

```
exists[i:1..2] LIGHT[i]
```

Action Fluent

- An action fluent **a** defines the state after an action **a** has been executed but before the execution of other actions including the silent action tau.
- Suppose α is the alphabet of a system:

```
fluent a = <a,α∪{τ}-a>
```

- bell rings once the room is lit:

```
bellring && LIGHT
```

- Note that action bell_ring must not belong to the terminating action set of fluent LIGHT

Safety Properties: Mutual Exclusion

```
const N = 2
range Int = 0..N
SEMAPHORE(I=0) = SEMA[I],
SEMA[v:Int]    = (up->SEMA[v+1]
                 |when(v>0) down->SEMA[v-1]
                 ).

LOOP = (mutex.down->enter->exit->mutex.up->LOOP).
||SEMADEMO = (p[1..N]:LOOP
           || {p[1..N]}::mutex:SEMAPHORE(2)).

fluent CRITICAL[i:1..N] = <p[i].enter, p[i].exit>
```

fault (arrow pointing to SEMAPHORE(2))

- Two processes are in their critical sections simultaneously:

```
CRITICAL[1] && CRITICAL[2]
```

Safety Properties: Mutual Exclusion

The linear temporal logic formula **[]***F* – always *F* – is true if and only if the formula F is true at the current instant and at all future instants.

- No two processes can be at critical sections simultaneously:

```
assert MUTEX = []!(CRITICAL[1] && CRITICAL[2])
```

- LTSA compiles the assert statement into a property process:

```
property MUTEX =(p[i:1..N].enter -> p[i].exit -> MUTEX
).
```

Safety Properties: Mutual Exclusion

```
Trace to property violation in MUTEX:
      p.1.mutex.down
      p.1.enter            CRITICAL.1
      p.2.mutex.down       CRITICAL.1
      p.2.enter            CRITICAL.1 && CRITICAL.2
```

- General expression of the mutual exclusion property for N processes:

```
assert MUTEX_N(N=2) = []!(exists [i:1..N-1]
(CRITICAL[i] && CRITICAL[i+1..N] ))
```

Safety Properties: Oneway in Single-Lane Bridge

```
const N = 2 // number of each type of car
range ID= 1..N // car identities

fluent RED[i:ID]  = <red[i].enter, red[i].exit>
fluent BLUE[i:ID] = <blue[i].enter, blue[i].exit>

assert ONEWAY  = []!(exists[i:ID] RED[i]
                  && exists[j:ID] BLUE[j])
```

- Abbreviating exists[i:R] FL[i] as FL[R]

```
assert ONEWAY  = []!(RED[ID] && BLUE[ID])
```

Single Lane Bridge - safety property ONEWAY

The fluent proposition is more concise as compared with the property process ONEWAY. This is usually the case where a safety property can be expressed as a relationship between abstract states of a system.

```
property ONEWAY =(red[ID].enter   -> RED[1]
                 |blue.[ID].enter -> BLUE[1]
                 ),
RED[i:ID] = (red[ID].enter -> RED[i+1]
            |when(i==1)red[ID].exit  -> ONEWAY
            |when(i>1) red[ID].exit  -> RED[i-1]
            ),         //i is a count of red cars on the bridge
BLUE[i:ID]= (blue[ID].enter-> BLUE[i+1]
            |when(i==1)blue[ID].exit -> ONEWAY
            |when( i>1)blue[ID].exit -> BLUE[i-1]
            ).         //i is a count of blue cars on the bridge
```

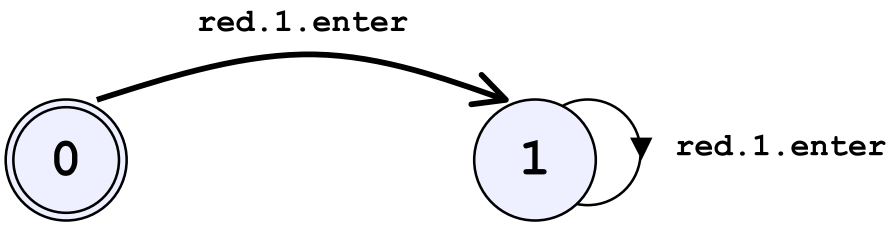
Liveness Properties

The linear temporal logic formula **<>*F*** – eventually *F* – is true if and only if the formula F is true at the current instant or at some future instant.

- First red car must eventually enter the bridge:

```
assert FIRSTRED  = <>red[1].enter
```

- To check the liveness property, LTSA transforms the negation of the assert statement in terms of a **Büchi** automaton.
- A **Büchi** automaton recognizes an infinite trace if that trace passes through an acceptance state infinitely often.



Büchi Automaton

Liveness Properties: Progress Properties

- Compose the Büchi automaton and the original system.
- Search for acceptance state in strong connected components.
- Failure of the search implies no trace can satisfy the Buchi automaton.
- It validates that the assert property holds.

- Red and blue cars enter the bridge infinitely often.

```
assert REDCROSS  = forall [i:ID] []<>red[i].enter
assert BLUECROSS = forall [i:ID] []<>blue[i].enter
assert CROSS     = (REDCROSS && BLUECROSS)
```

Liveness Properties: Response Properties

- If a red car enters the bridge, it should eventually exit.
- It does not stop in the middle or fall over the side!

```
assert REDEXIT = forall [i:ID]
          [](red[i].enter -> <>red[i].exit)
```

- Such kind of properties is sometimes termed "response" properties, which follows the form:

```
[](request-> <>reply)
```

- This form of liveness property cannot be specified using the progress properties discussed earlier.

Liveness Properties:

```
assert MUTEX_N(N=2) = []!(exists [i:1..N-1]
(CRITICAL[i] && CRITICAL[i+1..N] )) //safety

assert EXIT_N(N=2) = forall[i:1..N]
      [](p[i].enter -> <>p[i].exit  //liveness

assert MUTEX_LIVE(N=2) = MUTEX_N(N) && EXIT_N(N)
```

- EXIT_N asserts that a process enterng the critical section will eventually exit.

Fluent Linear Temporal Logic (FLTL)

- There are five operators in FLTL
  - Always []
  - Eventually <>
  - Until U
  - Weak until W
  - Next time X

- Amongst the five operators, always [] and eventually <> are the two most commonly used ones.

- Until, Weak until and Next time allows complex relation between abstract states.

FLTL: Until U

The linear temporal logic formula ***p*** **U** ***q*** – *p* until *q* – is true if and only if *q* is true at the current instant or *p* is true until some future instant where *q* is true.

- We should not enter a room before knocking

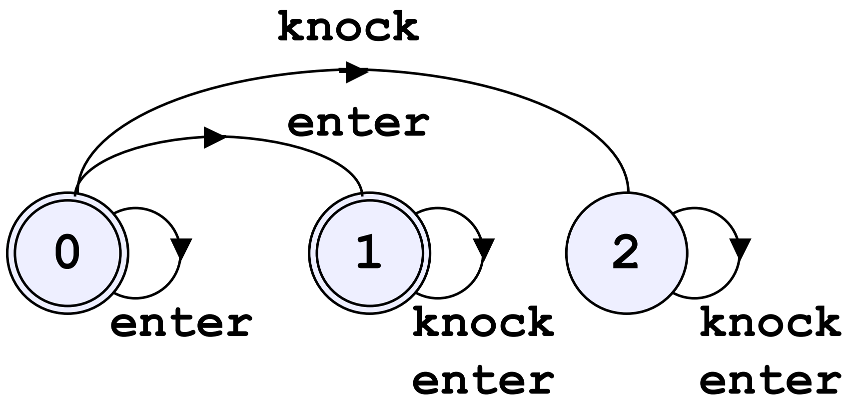
```
assert POLITE = (!enter U knock)
```

- The proposition also mandates that a knock action should eventually happen

- The proposition is not purely a safety property

FLTL: Until U

Büchi automaton of POLITE



- Note that if a knock never occurs, the property is violated because the automaton remains in the acceptance state 0.

FLTL: Weak Until W

The linear temporal logic formula ***p* W *q*** – *p* weak until *q* – is true if and only if *p* is true indefinitely or if *p* U *q*.

```
assert POLITE = (!enter W knock)
```

- We should not enter a room before knocking.
- It does not mandate that knock will eventually happens.
- The proposition is a safety property.

Definitions

- always [], eventually <> and weak until W can be simulated by until U:
  - `<>p ≡ true U p`
  - `[]p ≡ !<>!p`
  - `p W q ≡ []p || (p U q)`

- FLTL allows boolean expressions of constants and parameters.
  - `assert true = rigid(1)`
  - `assert false = rigid(0)`

  - rigid(0) and rigid(1) are fluent propositions that do not change truth value with the passage of time as measured by the occurrence of events.

FLTL: Next time X

The linear temporal logic formula **X** ***p*** – next *p* – is true if and only if *p* is true at the next instant.

- By next instant, we mean when the next action occurs – this includes silent actions.

```
assert SEQ = (a && X b && X X c)
```

- The proposition requires that the system executes a in the initial instance, which is immediately followed by b and then c.

Summary

- A fluent is defined by a set of initiating actions and a set of terminating actions.
- At a particular instant, a fluent is true if and only if it was initially true or an initiating action has previously occurred and, in both cases, no terminating action has yet occurred.
- In general, we don't differentiate safety and liveness properties in linear temporal logic.
- We verify an LTS model against a given set of fluent propositions.
- LTSA evaluates the set of fluents that hold each time an action has taken place in the model.

Course Outline

2. Processes and Threads
3. Concurrent Execution
4. Shared Objects & Interference
5. Monitors & Condition Synchronization
6. Deadlock
7. Safety and Liveness Properties
8. Model-based Design

The main basic

Concepts

Models

Practice

Advanced topics …

9. Dynamic systems
10. Message Passing
11. Concurrent Software Architectures
12. Timed Systems
13. Program Verification
14. **Logical Properties**