Chapter 14

Logical Properties

Satisfied?  Not satisfied?
Logical Properties

**Concepts**: modeling properties that refer to states

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

**fluent linear temporal logic FLTL**

**Practice**: **assert** – Java proposition on the state of variables
Background

◆ Temporal Logic due to Pneuli (1977) is a popular means to describe process properties in logic.
◆ Use propositions on selected variable states at particular points in program executions.
◆ Realized as the `assert` construct in Java.

States in an LTS model based on actions or events? HOW?
◆ Introduce **fluent**s to describe abstract “states”.
◆ Express both safety and liveness properties as fluent propositions.

Fluents

const False = 0
const True = 1

SWITCH = (on->{off, power_cut}->SWITCH).

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

fluent DARK = <{off, power_cut},{on}>
   initially True
**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\}\>** thus 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}\}.

![Diagram showing s_i and e_i transitions](image-url)
Fluent Linear Temporal Logic (FLTL) Expressions

- FLTL expression can be constructed using Boolean operators and quantifiers:
  \( \&\&, ||, !, \rightarrow, \leftarrow, \forall, \exists \)

- E.g., If the light is on, power is also on:
  
  ```
  fluent \text{LIGHT} = \langle \text{on}, \text{off} \rangle \\
  fluent \text{POWER} = \langle \text{power\_on}, \text{power\_off} \rangle \\
  \text{LIGHT} \rightarrow \text{POWER}
  ```

- All lights are on:
  
  ```
  fluent \text{LIGHT}[i:1..2] = \langle \text{on}[i], \text{off}[i] \rangle \\
  \forall[i:1..2] \ \text{LIGHT}[i]
  ```

- At least one light is on:
  
  ```
  fluent \text{LIGHT}[i:1..2] = \langle \text{on}[i], \text{off}[i] \rangle \\
  \exists[i:1..2] \ \text{LIGHT}[i]
  ```
Fluent Linear Temporal Logic (FLTL) Expressions

- There are five temporal 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.
Temporal propositions

const False = 0
const True = 1

SWITCH = (power_on -> OFF),
OFF   = (on -> ON | power_off -> SWITCH),
ON    = (off-> OFF | power_off -> SWITCH).

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

assert OK = [](LIGHT -> POWER)
Safety Properties: Mutual Exclusion

\[
\begin{align*}
\text{const } N &= 2 \\
\text{range } \text{Int} &= 0..N \\
\text{SEMAPHORE}(I=0) &= \text{SEMA}[I], \\
\text{SEMA}[v:\text{Int}] &= (\text{up} \to \text{SEMA}[v+1] \\
&\quad \text{when}(v>0) \downarrow \text{SEMA}[v-1] \\
\end{align*}
\]

\[
\text{LOOP} = (\text{mutex}.\downarrow \to \text{enter} \to \text{exit} \to \text{mutex}.\uparrow \to \text{LOOP}).
\]

\[
||\text{SEMADEMO} = (p[1..N]:\text{LOOP} \\
&\quad || \{p[1..N]\}::\text{mutex}:\text{SEMAPHORE}(2)).
\]

\[
\text{fluent CRITICAL}[i:1..N] = <p[i].\text{enter}, p[i].\text{exit}> 
\]

\[\text{Two processes} \text{ are not in their critical sections simultaneously?}\]
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 safety property process with an ERROR state.
Safety Properties: Mutual Exclusion

Trace to property violation in MUTEX:

- p.1.mutex.down
- p.1.enter \quad \text{CRITICAL.1}
- p.2.mutex.down \quad \text{CRITICAL.1}
- p.2.enter \quad \text{CRITICAL.1} \land \text{CRITICAL.2}

General expression of the mutual exclusion property for N processes:

\text{assert \ MUTEX}_N(N=2) = \neg ![\text{exists } [i:1..N-1]
\quad (\text{CRITICAL}[i] \land \text{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 $\langle \rangle F$ – 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 = $\langle \rangle red[1].enter$

- 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.
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.
Fluent Linear Temporal Logic (FLTL)

- There are five operators in FLTL
  - Always $\Box$
  - Eventually $\diamond$
  - Until $U$
  - Weak until $W$
  - Next time $X$

- Amongst the five operators, always $\Box$ and eventually $\diamond$ are the two most commonly used ones.

- Until, Weak until and Next time allows complex relation between abstract states.
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 fluent linear temporal logic FLTL.

- 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 (Case Study)

The main basic
Concepts
Models
Practice

Advanced topics …
9. Dynamic systems
10. Passing
11. Concurrent Software Architectures
12. Timed Systems
13. Program Verification
14. Logical Properties

2015 Concurrency: logical properties