Design and Analysis of Distributed Interacting Systems

Lecture 5 – Linear Temporal Logic (cont.)

Prof. Dr. Joel Greenyer

May 2, 2013
(Last Time:) LTL Semantics (Informally)

- LTL Formulae are interpreted on the runs of a Kripke structure

\[
\begin{align*}
\models \mathbf{X} p & \quad \text{means: } p \text{ is element of the label of the state in the next state,} \\
\models \mathbf{G} p & \quad \text{“satisfies” an initial state of a Kripke structure,} \\
\models \mathbf{F} p & \quad \text{“satisfies” an initial state of a Kripke structure} \\
\models p \mathbf{U} q & \quad \text{“satisfies” an initial state of a Kripke structure}
\end{align*}
\]
(Last Time:) Typical properties in LTL

- $p$ is always eventually followed by $q$
  - $G (p \Rightarrow F q)$

- $p$ is always directly followed by $q$
  - $G (p \Rightarrow X q)$

- $p$ will eventually be true forever
  - $F G p$

- $p$ will always be true
  - $G p$

- $p$ will be true infinitely often
  - $G F p$  ($p$ will always eventually be true)
Equivalence of LTL properties

• Def. 4: LTL formulae $\varphi$ and $\psi$ are said to be equivalent, written $\varphi \equiv \psi$, iff for all Kripke Structures $M$ we have
  $$M \models \varphi \iff M \models \psi$$

• For example, the following holds:
  $$\neg F \varphi \equiv true \ U \varphi$$
• **Def. 3:** Let $\pi$ be a run. Then $\pi \vdash \varphi$ is defined as follows
  
  - $\pi \vdash p$, $p \in AP$, iff $p \in L(s_0)$, i.e., $p$ holds in the first state of $\pi$
  - $\pi \vdash \neg \varphi$ iff not $\pi \vdash \varphi$
  - $\pi \vdash \varphi \lor \psi$ iff $\pi \vdash \varphi$ or $\pi \vdash \psi$
  - $\pi \vdash X \varphi$ iff $\pi^1 \vdash \varphi$
  - $\pi \vdash G \varphi$ iff $\forall \geq 0 : \pi^i \vdash \varphi$
  - $\pi \vdash F \varphi$ iff $\exists \geq 0 : \pi^i \vdash \varphi$
  - $\pi \vdash \varphi U \psi$ iff $\exists \geq 0 : \pi^k \vdash \psi$
    and $\forall, \ 0 \leq i < k : \pi^i \vdash \varphi$. 
Proof: Eventualy by Until

• Proof: $F \varphi \equiv true \ U \varphi$

• We consider a run $\pi$

\[
\begin{align*}
\pi &\models F \varphi \\
\iff &\exists i \geq 0 : \pi^i \models \varphi & \text{(Def. F)} \\
\iff &\exists i \geq 0 : \pi^i \models \varphi \\
&\land \forall j, 0 \leq j < i : \pi^j \models true & \text{(true holds in all states)} \\
\iff &\pi \models true \ U \varphi & \text{(Def. U)}
\end{align*}
\]

• We show the equivalence for any run $\pi$,
  – so the equivalence also holds for all runs of any Kripke Structure
  – thus $F \varphi \equiv true \ U \varphi$ holds according to Def. 4
More Equivalences

• Duality
  – \(\neg G \varphi \equiv F \neg \varphi\)
  – \(\neg F \varphi \equiv G \neg \varphi\)
  – \(\neg X \varphi \equiv X \neg \varphi\)

• Idempotency
  – \(G G \varphi \equiv G \varphi\)
  – \(F F \varphi \equiv F \varphi\)
  – \(\varphi U (\varphi U \psi) \equiv \varphi U \psi\)
  – \((\varphi U \varphi) U \psi \equiv \varphi U \psi\)
More Equivalences

• Absorption
  - $F G F \varphi \equiv G F \varphi$
  - $G F G \varphi \equiv F G \varphi$
  - $\neg X \varphi \equiv X \neg \varphi$

• Distributivity
  - $X (\varphi U \psi) \equiv (X \varphi) U (X \psi)$

• Expansion
  - $\varphi U \psi \equiv \psi \lor (\varphi \land X (\varphi U \psi))$
  - $F \varphi \equiv \varphi \lor X F \varphi$
  - $G \varphi \equiv \varphi \lor X G \varphi$
Characterizing Properties

• Remember
  – safety: nothing bad ever happens
  – liveness: something good eventually happens

• $\phi$ is a safety formula iff for every run $\pi$ such that $\pi \not\models \phi$ $\pi$ has a prefix $\pi[0..k] = s_0, ..., s_k$ such that for all infinite extensions $\pi'$ of $\pi[0..k]$ $\pi' \not\models \phi$ holds.

• $\phi$ is a liveness formula iff for every finite sequence of states $s_0, ..., s_k$ can be extended so that $\pi \models \phi$ holds.
Expressive Power of LTL

• Are there properties that cannot be expressed in LTL?
  – Yes

• Properties that refer to the branching structure of the Kripke structure: “There exist a path where ...”
  – can be expressed in CTL (later)

• No Counting:
  – “There are as many occurrences of states where $p$ holds as there are states where $q$ holds”
    • requires and infinite counter
  – A property that is true in states after even occurrences of $p$
    • requires counting to two
Design and Analysis of Distributed Interacting Systems

Lecture 5 – The Spin Model Checker

Prof. Dr. Joel Greenyer

April 25, 2013
The Spin Model Checker

- **Spin** *(Simple Promela Interpreter)*
  - tool for simulating and verifying multi-threaded software and distributed system designs
    - “...operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, control software for spacecraft, nuclear power plants, etc.”
      (from: [http://spinroot.com/spin/what.html](http://spinroot.com/spin/what.html))
    - Verification of LTL formulae

- **Promela**: Process Meta Language
  - C-like language for describing concurrent processes
The Spin Model Checker

- **Success Stories** *(from: http://spinroot.com/spin/success.html)*
  - “Mission Critical Software: Selected algorithms for a number of space missions were verified with the Spin model checker. The missions include Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, etc.”
  - “Verification of medical device transmission protocols: Spin was used for about ten years in the verification of international standards that are used worldwide.”
  - ...

- **ACM Software System Award 2001**

(see http://awards.acm.org/homepage.cfm?sort=all&awd=149)
Promela

- Allows us to describe concurrent, communicating processes
  - concurrent processes are executed in an *interleaved* fashion

- Communication via channels
  - synchronous and asynchronous

- Communication via shared variables
• A typical structure of a Promela model:

```promela
byte brightness;

mtype = {press, hold};

chan c = [0] of { mtype };

proctype light(){
  ...
}

proctype switch(){
  ...
}

init{
  run light(); run switch()
}
```

- **variable declarations**: bit (1), bool(1), byte(8), short(16), int(32)
- **mtype**: symbolic names of numeric constants (press=1, hold=2, ...)
- **channel declaration**: with finite buffer size (0: synchronous channel); channels transfer messages with fields of different types
- **procedure declaration**: (can have parameters)
- **initialization**: of the model, instantiation of processes
Variables and Types

- Basic types: bit (1), bool(1), byte(8), short(16), int(32)

  ```
  byte brightness; bool lightOn
  ```

- Arrays

  ```
  bit lightsOn[3];
  ```

- Records

  ```
  typedef Record { short f; byte g; } 
  Record r;
  ...
  r.f = ...;
  ```

- Constants

  ```
  #define MAXBRIGHTNESS 3;
  ```
Processes

• Process definition

\[
\text{proctype <name> (<parameters>)}\{ \\
\quad \text{< body>} \\
\}
\]

• Process execution
  – initialize in init:

\[
\text{init}\{ \\
\quad \text{run light(); run switch(); run switch()} \\
\}
\]

  – declare as active:

\[
\text{active proctype light()}\{ \\
\quad \ldots \\
\}
\]

\[\text{two running instances of switch()}\]
Branching

• Example:

if
:: (counter < x) -> counter++;
:: (counter >= x) -> printf("Done")
fi

• Non-deterministic choice if multiple guards hold:

if
:: (counter < x+4) -> counter++;
:: (counter >= x-3) -> printf("Done")
fi

• else branch is taken if no other option is executable:

if
:: (counter < x) -> counter++;
:: else -> printf("Done")
fi
Labels and Jumps

• Example:

```c
proctype sum(byte x){
    int s, counter;
    printf("Calculating sum from 0 to %d\n",x);

    AGAIN:
    counter++;
    s = s + counter;
    if :: (counter < x) -> goto AGAIN
    :: (counter >= x) -> goto DONE
    fi;
    printf("This text will not be printed.\n");

    DONE:
    printf("The sum from 0 to %d is %d\n", x, s);
}
```
Loops

• do loops with different alternative options
  – Non-deterministic choice if multiple guards hold:

```c
proctype sum(byte x){
    int s, counter;
    printf("Calculating sum from 0 to %d\n", x);
    do
        :: (counter > 2 & counter <= x) ->
            s = s + counter;
            counter++
        :: (counter < 4 & counter <= x-1) ->
            s = s + counter + counter + 1;
            counter = counter + 2
        :: (counter > x) -> break;
    od;
    printf("The sum from 0 to %d is %d\n", x, s);
}
```
Communication via Channels

• Channel declaration

\[
\text{chan } \langle \text{name} \rangle = [\langle \text{length} \rangle] \text{ of } \{\langle \text{type1} \rangle, \ldots, \langle \text{typen} \rangle\}
\]

• For example:

\[
\text{chan intQueue = [5] of } \{\text{int}\} \quad //\text{asynchronous}
\]
\[
\text{chan bb = [0] of } \{\text{byte, byte}\} \quad //\text{synchronous}
\]

• mtypes: symbolic names of numeric constants

\[
\text{mtype = } \{\text{press, hold}\};
\]
\[
\text{chan c = [0] of } \{\text{ mtype } \};
\]
Communication via Channels

- Channels are FIFO queues
- Receiver has to wait when channel is empty
- Sender has to wait when channel is full
  - or messages are lost (depends on settings of Spin)
- Functions on channels

```
len(c) // number of messages in c
empty(c) // is channel empty?
nempty(c) // is channel not empty?
full(c) // is channel full?
nfull(c) // is channel not full?
```
Communication via Channels

- Sending and receiving

```c
chan bb = [5] of {byte, byte};

active proctype A(){
    byte x, y;
    bb?x,y;
    printf("x is %d, y is %d\n", x, y);
    if
        :: bb?x,4 -> printf("x is %d\n", x)
        :: bb?3,y -> printf("y is %d\n", y)
    fi;
}

active proctype B(){
    byte x = 2;
    bb!x,5;
    bb!x+1,x*2
}
```

- Receiving and assigning the message values to (local) variables
- Printf, printing values of decimal variables (%d)
- If block with different choices.
- Conditional receiving
- Non-determinism if multiple choices valid
- Sending values over a channel

; and -> are statement separators (same meaning)
Example: Light Switch

mtype = {press, hold};

chan c = [0] of { mtype };;

active proctype switch(){
    RELEASED:
        if
            :: c!press; goto PRESSED
        fi;
    PRESSED:
        if
            :: c!hold; goto PRESSED
        :: goto RELEASED
    fi;
}

active proctype light(){
    OFF:
        if
            :: c?press; goto LOW
        fi;
    LOW:
        if
            :: c?press; goto OFF
            :: c?hold; goto HIGH
        fi;
    HIGH:
        if
            :: c?press; goto OFF
            :: c?hold; goto LOW
        fi;
}

(this is a possible pattern to model state machines in Promela)
Simulating the Light Switch

```
0: proc - (.root:) creates proc 0 (light)
0: proc - (.root:) creates proc 1 (switch)
Selected: 1
1: proc 1 (switch) light_simple.pml:25 (state 1) [c!press]
1: proc 0 (light) light_simple.pml:8 (state 1) [c?press]
Selected: 1
2: proc 1 (switch) light_simple.pml:29 (state 5) [c!hold]
2: proc 0 (light) light_simple.pml:13 (state 7) [c?hold]
Selected: 2
3: proc 1 (switch) light_simple.pml:28 (state 8) [goto RELEASED]
Selected: 1
4: proc 1 (switch) light_simple.pml:25 (state 1) [c!press]
4: proc 0 (light) light_simple.pml:17 (state 11) [c?press]
Selected: 2
5: proc 1 (switch) light_simple.pml:28 (state 8) [goto RELEASED]
```

Diagram:

```
light:0
  1!press
  1?hold
  1!hold
switch:1
  1!press
  1?press
```

Select a statement

1: proc 1 (switch) light_simple.pml:24 (state 3) [c!press]
quit
Atomic Sequences

• Sequences of statements that will not be interleaved with statements in other processes
  – (unless there is synchronous communication involved...)

• Example:

```java
active proctype TableSensor()
{
    do
    :: atomic{ blankOnTable = true;
        ts2c!blankArrived;
    }
    od
}
```
Verification Options with Spin

• Spin supports a number of verification options
  – check *assertions*
  – find invalid end states (deadlocks)
  – check liveness (progress conditions, similar to LTSA)
  – check *traces assertions*: assertions on the order of sendings and receivings of messages
  – check *never claims*: sequence of Boolean expressions over variables in the model that must never happen
  – check LTL formulae
Spin Models and Kripke Structures

- A Spin model can be translated to a Kripke Structure
  - data types, channels, max. no. of processes is finite
  - Spin can do an exhaustive analysis of the corresponding KS
  - Spin constructs KS “on-the-fly”, i.e., sometimes it finds results without constructing the complete KS

```c
byte x, y;
active proctype mini(){
  do
    :: (x < 2) -> x++
    :: (y < 2) -> y++
    :: else -> break
  od
}
```
Spin Models and Kripke Structures

- There can be multiple paths to the same state

- Equal states must also be the same states! How?

- Roughly, Spin uses a Hash table to store and lookup states:
Spin Verification more Technically...

Promela model

C program

Spin settings

Output

```
byte x, y;
active proctype mini(){
do
:: (x < 2) ->
  x++
:: (y < 2) ->
  y++
:: else ->
  break
od
}
```

```
verification result:
spin -a light_brightness.pml
 gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFA /pan -m10000
 Pid: 9028
 pan:1: invalid end state (at depth 6)
 pan: wrote light_brightness.pml.trail
(Spin Version 6.2.3 -- 24 October 2012)
Warning: Search not completed
  + Partial Order Reduction
```
...  
#define trainOnCrossing 3  
#define carOnCrossing 2  
...

active proctype train(){
    byte state;
    ...
}

active proctype car(){
    byte state;
    ...
}

active proctype Inv(){
    assert(!(train:state == trainOnCrossing &&
             car:state == carOnCrossing))
}

during the exhaustive state space exploration during model checking, all possible interleavings of the other processes and executing this assertion will be checked

when is this assertion executed?
Never Claim

• Sequence of Boolean expressions over variables in the model that must never happen

• Simple example:

```java
byte x = 3;

active proctype P(){
    x = 1;
}

never{
    x == 3;
    x == 1
}
```
... 

active proctype light(){
    OFF:
        if
            :: c?press; goto LOW
        fi;
    LOW:
        if
            :: c?press; goto OFF
            :: c?hold; goto HIGH
        fi;
    HIGH:
        if
            :: c?press; goto OFF
            :: c?hold; goto LOW
        fi;
}

ever { 
    true;
    light@LOW;
    true;
    light@HIGH;
}
mtype = {press, hold};
chan c = [0] of { mtype };

active proctype switch(){
    RELEASED:
        if
            :: c!press; goto PRESSED
        fi;
    PRESSED:
        if
            :: c!hold; goto PRESSED
        :: goto RELEASED
        fi;
}

active proctype light(){
    OFF:
        if
            :: c?press; goto LOW
        fi;
    LOW:
        if
            :: c?press; goto OFF
        :: c?hold; goto HIGH
        fi;
    HIGH:
        if
            :: c?press; goto OFF
        :: c?hold; goto LOW
        fi;
}

\[
\text{ltl p0 } \{[]<> \text{light@LOW}\}
\]
\[
\text{ltl p1 } \{[]<> \text{light@HIGH}\}
\]
Simple Production Cell

- Simplified Production Cell example: Just one arm, no press
- <Demo>
Summary

• Equivalences of LTL properties
• Characterizing safety and liveness properties

• Introduction to Promela:
  – Variables and types, processes
  – Branching and loops, labels and jumps
  – Synchronous and asynchronous channels
  – Atomic sequences

• Verification with Spin
  – different options
  – assertions
  – never claims
  – LTL formulae