SPIN: Verifying LTL properties *

Alessandra Giordani
agiordani@disi.unitn.it
http://disi.unitn.it/~agiordani

Formal Methods Lab Class, March 26, 2014

Università degli Studi di Trento

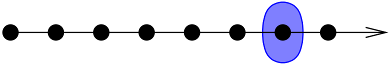
*These slides are derived from those by Stefano Tonetta, Alberto Griggio, Silvia Tomasi, Thi Thieu Hoa Le for FM lab 2011/13

Contents

Contents

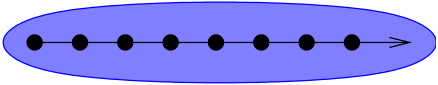
LTL specifications

finally p



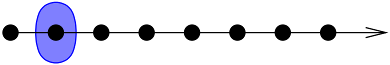
F p

globally p



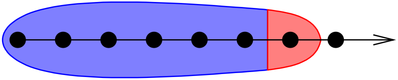
G p

next p



X p

p until q



p U q

LTL syntax with SPIN

- Grammar:
  - `ltl ::= opd | ( ltl ) | ltl binop ltl | unop ltl`
- Operands (`opd`):
  - `true`, `false`, and user-defined names starting with a lower-case letter
- Unary Operators (`unop`):
  - `[]` (the temporal operator always)
  - `<>` (the temporal operator eventually)
  - `!` (the boolean operator for negation)
- Binary Operators (`binop`):
  - `U` (the temporal operator strong until)
  - `V` (the dual of U, release): (p V q) means !(!p U !q))
  - `&&` (the boolean operator for logical and)
  - `||` (the boolean operator for logical or)
  - `->` (the boolean operator for logical implication)
  - `<->` (the boolean operator for logical equivalence)

LTL model checking: intuition

To model check if $M \models \phi$, Spin does

- build an automaton $A_{\neg\phi}$ that encodes all violations of ϕ,
- consider the synchronous execution of M and $A_{\neg\phi}$
  $\Longrightarrow A_M \times A_{\neg\phi}$ represents the paths in M that do not satisfy ϕ.

$A_{\neg\phi}$ ("never claim") can be seen as a monitoring machine that accepts some infinite executions of the system. If there exists an execution accepted by $A_{\neg\phi}$, that execution is a violation of ϕ.

Verifying LTL properties with SPIN 1/2

- Suppose we want to verify that a system satisfies a property.
  Example: in the system `foo.pml`, a boolean variable `b` is always true.
- Write the corresponding LTL formula using some simple symbols as atomic propositions (usually, single characters): `[] p`.
- Write the symbol definitions:
  ```
  > echo ``#define p (b==true)'' > foo.aut
  ```
- Generate the never claim corresponding to the negation of the property:
  ```
  > spin -f '!([] p)' >> foo.aut
  ```

Verifying LTL properties with SPIN 2/2

- Generate the verifier:

  `> spin -a -N foo.aut foo.pml`
- Option `-N file.aut` adds the never claim stored in `file.aut`
- Compile and run the verifier:

  `> gcc -o pan pan.c`

  `> ./pan -a`
- When a never claim is present and `-a` option is used, the verifier reports the existence of an execution accepted by the never claim. This execution corresponds to a violation of the property.

Remote references

- Typically, in order to test the local control state of active processes, we use the remote reference `procname[pid]@label`.
- This function return a non-zero value iff the process `procname[pid]` is currently in the local control state marked by `label`.
- Example:

  `[]!(mutex[0]@critical && mutex[1]@critical)`
- We can also refer to the current value of local variable by using `procname[pid]:var`

Predefined global variables and functions

- The predefined local variable `_pid` stores the process instantiation number (pid) of a process.
- The predefined global variable `_last` stores the pid of the process that performed the last execution.
- The function `enabled(pid)` returns true if the process with identifier `pid` has at least one executable statement in its current control state.

Contents

Weak Fairness

An event E occurs infinitely often. Example:

- Let R_i be true iff the process i is running.
- Weak Fairness: every process runs infinitely often.

$$\bigwedge_i \mathbf{GF} R_i$$

- In the following, we will use the following abbreviation:

$$FAIRRUN := \bigwedge_i \mathbf{GF} R_i$$

- It is often used as condition for other properties.
- In SPIN:

```
[]<> _last==0 && []<> _last==1 ...
```

Strong Fairness

If an event $E1$ occurs infinitely often, then the event $E2$ occurs infinitely often. Example:

- Let E_i be true iff the process i can execute a statement.
- Strong Fairness: if a process is infinitely often ready to execute a statement , then that process runs infinitely often.

$$\bigwedge_i (\mathbf{GF} E_i \rightarrow \mathbf{GF} R_i)$$

- In SPIN:

```
([]<> enabled(0) -> []<> _last==0) && ...
```

Exercise

Consider the following system:

```
int count;
bool incr;

active proctype counter() {
        do
        :: incr ->
                count++
        od
}
active proctype env() {
        do
        :: incr = false
        :: incr = true
        od
}
```

- Verify the property count reaches the value 10.

Exercise

Consider the following system:

```
int count;
bool incr;

active proctype counter() {
        do
        :: incr ->
                count++
        od
}
active proctype env() {
        do
        :: incr = false
        :: incr = true
        od
}
```

- Verify the property count reaches the value 10.
- Verify the property above under the fairness condition: `[]<> (incr && _last==0)`.

Note: `iSpin` does not accept the variable `_last`.

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- Eventually, a leader will be elected

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- Eventually, a leader will be elected
- In LTL:

$$\mathbf{F}(nLeaders > 0)$$

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- There is never more than one leader

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- There is never more than one leader
- In LTL:

$$\mathbf{G}!(nLeaders > 1)$$

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- When a process is elected, it will remain leader forever

Leader Election

The system

- N processes in a unidirectional ring network: each of them can send messages to its next neighbor and receive from its prev neighbor.
- Eventually, the process with the highest identifier will be elected leader.
- The variable *nLeaders* stores the number of leaders.

The properties:

- When a process is elected, it will remain leader forever
- In LTL:

$$\mathbf{G}(elected \rightarrow \mathbf{G}oneLeader)$$

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Mutual exclusion: there is no reachable state in which more processes are in the critical session.

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Mutual exclusion: there is no reachable state in which more processes are in the critical session.
- In LTL:

$$\mathbf{G}!(\bigvee_{i \neq j}(C_i \wedge C_j))$$

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Progress: if one process is in T, then eventually some process will enter C.

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Progress: if one process is in T, then eventually some process will enter C.
- In LTL:

$$\mathbf{G}(\bigvee_i T_i \rightarrow \mathbf{F} \bigvee_i C_i)$$

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Lockout-freedom: in a fair path, if a process is in T, eventually it enters C.

Mutual Exclusion

The system

- N processes are trying to access a critical session.
- Let T_i be true iff the process i is resp. in the trying session and C_i be true iff it is in the critical session.

The properties:

- Lockout-freedom: in a fair path, if a process is in T, eventually it enters C.
- In LTL:

$$FAIRRUN \rightarrow \mathbf{G}(\bigwedge_i (T_i \rightarrow \mathbf{F} C_i))$$

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let *sentA* be true iff P_1 has just sent the message A and *recA* be true iff P_2 has just received the message A. Similarly for *sendB* and *recB*.
- Let *loss* be true iff the channel lost last message.

The properties:

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let *sentA* be true iff P_1 has just sent the message A and *recA* be true iff P_2 has just received the message A. Similarly for *sendB* and *recB*.
- Let *loss* be true iff the channel lost last message.

The properties:

- Response to Impulse: in a fair path, if a message is sent, then it is eventually received.

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let $sentA$ be true iff P_1 has just sent the message A and $recA$ be true iff P_2 has just received the message A. Similarly for $sendB$ and $recB$.
- Let $loss$ be true iff the channel lost last message.

The properties:

- Response to Impulse: in a fair path, if a message is sent, then it is eventually received.
- In LTL:

$$(FAIRRUN \wedge \mathbf{GF}!loss) \rightarrow (\mathbf{G}(sendA \rightarrow \mathbf{F}recA))$$

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let $sentA$ be true iff P_1 has just sent the message A and $recA$ be true iff P_2 has just received the message A. Similarly for $sendB$ and $recB$.
- Let *loss* be true iff the channel lost last message.

The properties:

- Absence of Unsolicited Response: if a message is received, then it has been previously sent.

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let *sentA* be true iff P_1 has just sent the message A and *recA* be true iff P_2 has just received the message A. Similarly for *sendB* and *recB*.
- Let *loss* be true iff the channel lost last message.

The properties:

- Absence of Unsolicited Response: if a message is received, then it has been previously sent.
- In LTL:

$$\mathbf{F}\, recA \rightarrow ((\neg recA)\,\mathbf{U}\, sentA)$$

Alternating Bit Protocol

The system

- A process P_1 is trying to send messages to the process P_2 by means of a non-reliable channel, which can lose or duplicate the messages.
- Let *sentA* be true iff P_1 has just sent the message A and *recA* be true iff P_2 has just received the message A. Similarly for *sendB* and *recB*.
- Let *loss* be true iff the channel lost last message.

The properties:

- Absence of Unsolicited Response: if a message is received, then it has been previously sent.
- In LTL:

$$\mathbf{F}\, recA \rightarrow ((\neg recA)\mathbf{U} sentA)$$

- Alternative:

$$\neg((\neg sentA)\mathbf{U} recA)$$

-