Software Verification

Model checking CTL, Büchi acceptance for LTL
The Spin Model Checker

Ahmed Rezine

IDA, Linköpings Universitet

Vårtermin 2022

Outline

Model Checking CTL

LTL and Büchi acceptance

Spin: flagship *LTL* explicit model checking

Outline

Model Checking CTL

LTL and Büchi acceptance

Spin: flagship *LTL* explicit model checking

Branching Time Logic (*CTL*)

Each of **X**, **F**, **G**, **U**, **R** is immediately preceded by **E** or **A**.

The following are *state fromulas*

- p if $p \in AP$
- $\neg f$, $f \wedge g$ and $f \vee g$ if f, g are state formulas
- **A**f, **E**f if f a path formula

The following are *path fromulas*

- f if it is also a state formula
- $\neg f$, $f \wedge g$, $f \vee g$, **X** f, **F** f, **G** f, f**U** G and f**R** g if f, g are ~~path formulas~~ state formulas

Branching Time Logic (*CTL*)

Each of **X**, **F**, **G**, **U**, **R** is immediately preceded by **E** or **A**.

The following are *state fromulas*

- p if $p \in AP$
- $\neg f$, $f \wedge g$ and $f \vee g$ if f, g are state formulas
- $\mathbf{A}f$, $\mathbf{E}f$ if f a path formula

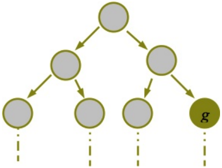
The following are *path fromulas*

- f if it is also a state formula
- $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\mathbf{U}\ G$ and $f\mathbf{R}\ g$ if f, g are ~~path formulas~~ state formulas

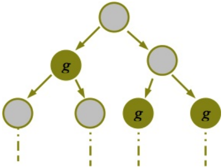
The most used operators are:

- $M, s_0 \models \mathbf{EF}\ g$,
- $M, s_0 \models \mathbf{AF}\ g$,
- $M, s_0 \models \mathbf{EG}\ g$,
- $M, s_0 \models \mathbf{AG}\ g$

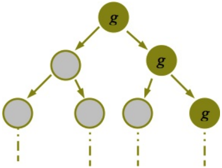
CTL



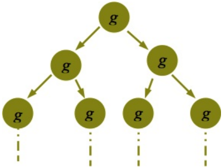
$M, s_0 \models \textbf{EF}\ g$



$M, s_0 \models \textbf{AF}\ g$

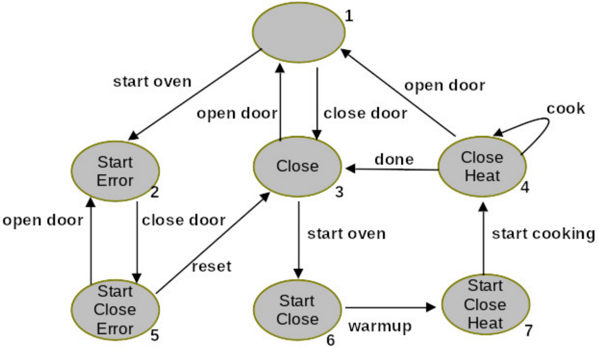


$M, s_0 \models \textbf{EG}\ g$



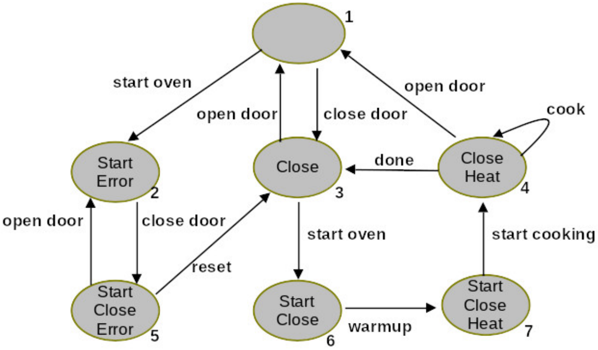
$M, s_0 \models \textbf{AG}\ g$

Model checking CTL properties



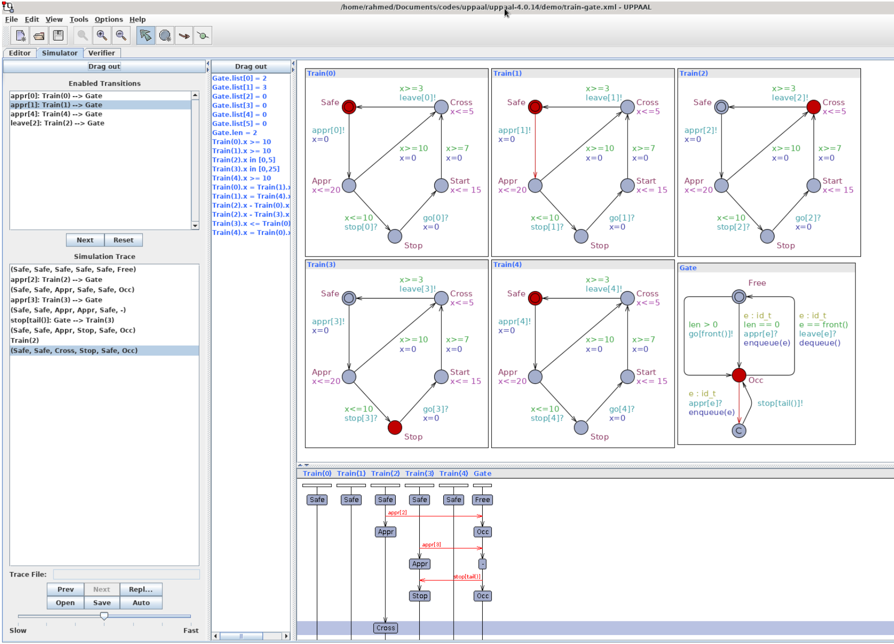
- $\mathbf{AX}(\mathrm{Heat}) = \neg(\mathbf{EX}(\neg \mathrm{Heat}))$
- $\mathbf{EG}(\mathrm{Error}) = \neg(\mathbf{AF}(\neg \mathrm{Error}))$
- $\mathbf{AG}(\mathrm{Start} \implies \mathbf{AF}(\mathrm{Heat})) = \neg(\mathbf{EF}(\mathrm{Start} \wedge \mathbf{EG}(\neg \mathrm{Heat})))$

Model checking CTL properties: adding fairness



- **AG**(Start $\implies$ **AF**(Heat)) considering paths where (Start $\wedge$ Close $\wedge$ ($\neg$Error)) appears infinitely often (the user operates the oven correctly infinitely often).

The UPPAAL model checker



Outline

Linear Time Logic (*LTL*)

LTL formulas are of the form $\mathbf{A}f$ where f is a path formula where the only allowed state formulas are atomic propositions, i.e., path formulas are of the form:

- f if ~~state formula~~ in AP
- $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\mathbf{U}\ G$ and $f\mathbf{R}\ g$ if f, g are path formulas

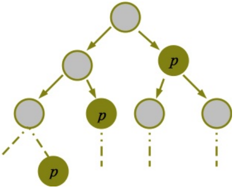
LTL examples

- invariance: $\mathbf{G}(\neg \mathrm{Error})$
- guarantee: $\mathbf{F}(\mathrm{Ok})$
- response: $\mathrm{Req} \implies \mathbf{F}(\mathrm{Ack})$
- precedence: $\mathrm{Req} \implies (\mathrm{Busy}\ \mathbf{U}\ \mathrm{Ack})$
- progress: $\mathbf{GF}(\mathrm{Move})$
- stability: $\mathbf{FG}(\mathrm{Stable})$
- weak fairness: $\mathbf{GF}(\neg \mathrm{Enabled} \vee \mathrm{Executed})$
- strong fairness: $\mathbf{GF}(\mathrm{Enabled}) \implies \mathbf{GF}(\mathrm{Executed})$

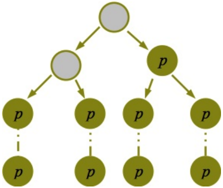
Linear Time Logic (*LTL*)

LTL formulas are of the form $\mathbf{A}f$ where f is a path formula where the only allowed state formulas are atomic propositions, i.e., path formulas are of the form:

- f if ~~state formula~~ in AP
- $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$, $f\mathbf{U}\ G$ and $f\mathbf{R}\ g$ if f, g are path formulas



AG(**EF** p) is in *CTL* but not *LTL*There is always a path to a state where p holds (e.g. reset).

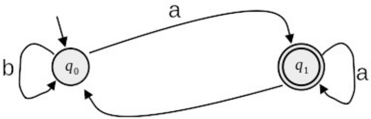


A(**FG**p) is in *LTL* but not *CTL*. Stability: there is a point after which p always hold.

Büchi automata

A Büchi automaton is a tuple $(Q, \Sigma, \Delta, q_0, F)$:

- Q a finite set of states
- Σ a finite alphabet
- $\Delta \subseteq Q \times \Sigma \times Q$ a transition relation
- q_0 an initial state
- $F \subseteq Q$ defines the acceptance condition: only those runs with at least one of the states in F appearing infinitely often.



LTL and Büchi acceptance

- invariance: $\mathbf{G}(\neg \mathrm{Error})$
- guarantee: $\mathbf{F}(\mathrm{Ok})$
- progress: $\mathbf{GF}(\mathrm{Move})$
- stability: $\mathbf{FG}(\mathrm{Stable})$

Outline

Model Checking CTL

LTL and Büchi acceptance

Spin: flagship *LTL* explicit model checking

Promela Models

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender(){
  /* Process body */
  ...
}

proctype Receiver() {
  ...
}

init{
  /* process creation */
  ...
}
```

A promela model consists of:

- type declarations
- channel declarations
- variable declarations
- init process

The model has to correspond to a finite kripke structure (usually a very large one). This means:

- bounded data,
- bounded channels,
- bounded number of processess
- bounded process creation

Promela Models (cont.)

A process:

- is defined by a proctype definition
- executes concurrently with all other processeses, irrespective of their relative speed
- communicate with other processes using shared variables and channels
- there can be several processes of the same type
- each process has its own local state defined by its process counter and values of its local variables

Promela Models (cont.)

- A process is created with the **run** statement which returns the process id
- processes can be created by other processes
- a created process starts executing after the the run statement
- processes can also be created by adding active in front of proctype

```
proctype Sender(chan a){
  ...
}

init{
  chan c = [1] of {bit};
  int pid2 = run Sender(c);
}

active[3] proctype Writer(){
  ...
}
```

Promela Models (cont.)

A process (proctype) in promela consists of:

- a name
- a list of formal parameters
- declarations of local variables
- body of the process: a sequence of statements

```
proctype Sender(chan in; chan out){
  bit sndB, rcvB; /* local variables */
  do
    :: out ! MSG, sndB ->
         in = ACK, rcvB;
         if
        ::sndB == rcvB -> sndB = 1 - sndB
        :: else -> skip

         fi
  od
}
```

Variables and Types

```
/*basic types*/
bit turn=1;
bool flag;
byte counter;
short s;
int msg;

/*arrays*/
byte a[27];
bit flags[4];
```

```
/*records*/
typedef Record{
  short f1;
  byte f2;
}

Record rr;

rr.f1=...
```

Statements

- Depending on the global state of the systems, a statement is either:
  - **executable**: can be executed immediately
  - **blocked**: cannot be executed immediately
- assignments are always executable
- expressions are executable if they evaluate to non-zero:
  - $2 < 3$ always executable
  - $x < 27$ executable if x is smaller than 27
  - $3 + x$ executable if $x \neq -3$
- **skip** is always executable
- **run** is executable if a new process can be created

Statements (cont.)

- **assert(<expr>)** is always executable
- if **expr** evaluates to zero, SPIN exits and reports the assertion has been violated
- Used to check validity of properties

```
proctype monitor(){
  assert(n <= 3);
}

proctype receiver(){
  ...
  toReveiver ? msg;
  assert(msg != ERROR);
  ...
}
```

Interleaving Semantics

- Processes execute concurrently
- Non-deterministic scheduling of the processes
- Executions of the processes are interleaved: one statement executes at a time except for the rendez-vous communication
- All statements are atomic, i.e. executed without interleaving with other processes' statements
- Sometimes, the same process can choose among several executable actions. One of them is chosen non-deterministically

if-statement

```
if
:: choice_1 -> stat_{1,1}; stat_{1,2}; ...
:: choice_2 -> stat_{2,1}; stat_{2,2}; ...
...
:: choice_n -> stat_{n,1}; stat_{n,2}; ...
(:: else -> stat_{n,1}; stat_{n,2}; ...)
fi;
```

- if at least one of the $choice_i$ is executable, then SPIN non deterministically chooses on of them
- if none of the $choice_i$ is executable, the if statement is blocked
- "->" is used to separate the guards from the statements that follow it (actually equivalent to ";").

do-statement

```
do
:: choice_1 -> stat_{1,1}; stat_{1,2}; ...
:: choice_2 -> stat_{2,1}; stat_{2,2}; ...
...
:: choice_n -> stat_{n,1}; stat_{n,2}; ...
od;
```

- similar to the if statement except for repetition
- **break** exits the do-loop

Communication

```
chan <name> = [<dim>] of {<t1, <t2>, ...};

chan c = [1] of {bit};
chan toR = [2] of {mtype, bit};
```

- Channels are used for communication
  - using bounded buffers channels for **message passing**
  - using “channels of size 0” for **rendez-vous** or handshake

Communication

- Sending: puts a message into a channel:

```
ch ! <expr1>, <expr2>, ... <exprn>;
```

- Receiving: fetching a message from a channel:

```
/* message passing */
ch ? <var1>, <var2>, ... <varn>;
/* message testing */
ch ? <const1>, <const2>, ... <constn>;
```

The **atomic** Statement

```
atomic {stat1; stat2; .... statn}
```

- used to group statements into an atomic sequence that executes without interleaving from other processes
- executable if $stat_1$ is executable
- if $stat_i$, for $i > 1$, is blocked, then atomicity is temporarily lost and other processes may do a step

The **d_step** Statement

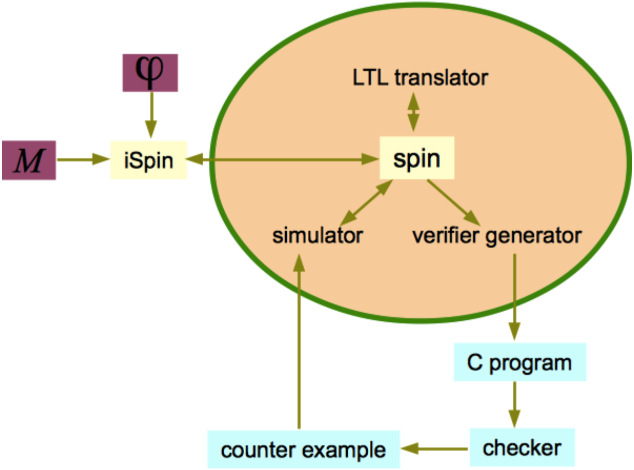
```
d_step {stat1; stat2; .... statn}
```

- more efficient than atomic: no intermediate states
- only deterministic statements allowed
- error if *stat*$_i$, for $i > 1$, blocked

Examples of Promela constructs

basic statements	`assignment`	always exec.
	`expression`	exec. if true (non zero)
	`send (ch!)`	exec. if ch not full
	`receive (ch?)`	exec. if ch not empty
	`assert(<expr>)`	always exec.
	`printf`	always exec.
expression statements	`skip`	always exec.
	`timeout`	true if no other statement is exec.
compound statements	`if`	exec. if at least one guard is exec.
	`do`	exec. if at least one guard is exec.
	`atomic`	exec. if first statement is exec.
	`d_step`	exec. if first statement is exec.
control flow	`goto`	jump to label
	`break`	exit do-statement

(i)Spin Architecture



Mutex: First Attempt

```
bit flag;        /* signal entering/leaving the CS */
byte mutex;      /* number of processes in the CS */

proctype P(bit i){
  flag != 1;
  flag = 1;
  mutex++;
  mutex--;
  flag = 0;
}

proctype monitor(){
  assert(mutex!=2);
}

init{
  atomic{ run P(0); run P(1); run monitor();}
}
```

Mutex: Second Attempt

```
bit x, y;            /* signal entering/leaving the CS */
byte mutex;          /* number of processes in the CS */

active proctype A(){
  x = 1;
  y == 0;
  mutex++;
  mutex--;
  x = 0;
}

active proctype B(){
  y = 1;
  x == 0;
  mutex++;
  mutex--;
  y = 0;
}

proctype monitor(){
  assert(mutex!=2);
}
```

Mutex: Dekker's Algorithm

```
bit x, y;         /* signal entering/leabin the CS */
byte mutex;       /* number of processes in the CS */
mtype {A_TURN, B_TURN}; /* who's turn is it?*/
byte turn;
```

```
active proctype A(){
  x = 1;
  turn = B_TURN;
  y == 0
     || (turn == A_TURN);
  mutex++;
  mutex--;
  x = 0;
}
```

```
active proctype B(){
  y = 1;
  turn = A_TURN;
  x == 0
     || (turn == B_TURN);
  mutex++;
  mutex--;
  y = 0;
}
```

```
proctype monitor(){
  assert(mutex!=2);
}
```

Mutex: Bakery Algorithm

```
byte turn[2];         /* who's turn is it? */
byte mutex;           /* number of processes in the CS */

proctype P(bit i){
do
:: turn[i] = 1;
   turn[i] = turn[1-i] + 1;
   (turn[1-i] == 0) || (turn[i] < turn[1-i]);
   mutex++;
   mutex--;
   turn[i] = 0;
od
}

proctype monitor(){
  assert(mutex!=2);
}

init {
  atomic{run P(0); run P(1); run monitor()}
}
```

Alternating Bit Protocol

```
mtype = { msg0, msg1, ack0, ack1 };

chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };

inline phase(msg, good_ack, bad_ack)
{
 do
 :: sender?good_ack -> break
 :: sender?bad_ack
 :: timeout ->
  if
  :: receiver!msg;
  :: skip /* lose message */
  fi;
 od
}

inline recv(cur_msg, cur_ack,
            lst_msg, lst_ack)
{
 do
 :: receiver?cur_msg -> sender!cur_ack;
    break /* accept */
 :: receiver?lst_msg -> sender!lst_ack
 od;
}
```

```
active proctype Sender()
{
 do
 :: phase(msg1, ack1, ack0);
    phase(msg0, ack0, ack1)
 od
}

active proctype Receiver()
{
 do
 :: recv(msg1, ack1, msg0, ack0);
    recv(msg0, ack0, msg1, ack1)
 od
}
```

Properties

Safety property:

- "nothing bad ever happens"
- invariants: x is never 0
- deadlock freedom: the system never reaches a state where no actions are enabled
- SPIN finds a trace leading to the bad state. If no traces exist, the safety property holds

Liveness properties:

- "something good eventually happens"
- termination: the system eventually terminates
- response: whenever X occrus, Y eventually occurs
- SPIN finds a reachable loop in which the good property does not happen. If no such loop is reachable, then the property holds.

<table>
<tr><td>invariance</td><td>$G(p)$</td><td>ltl {[] p}</td></tr>
<tr><td>response</td><td>$G(p \Rightarrow F(q))$</td><td>ltl {[](p => <> q)}</td></tr>
<tr><td>precedence</td><td>$G(p \Rightarrow (qUr))$</td><td>ltl {[] (p=> (q U r))}</td></tr>
<tr><td>objective</td><td>$G(p \Rightarrow F(q \Vert r))$</td><td>ltl {[] (p => <>(q || r))}</td></tr>
</table>

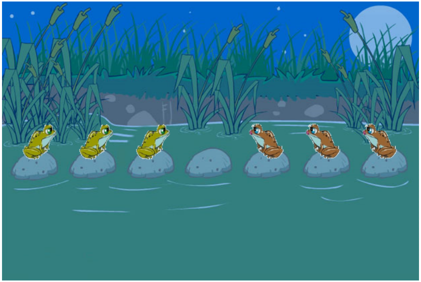
never Claims

- LTL formulas can be captured using Büchi automata
- SPIN uses **never** claims to capture Büchi automata
- capture finite behaviors and ω-acceptance cycles
- the kripke structure and the never claim execute in lockstep
- if the claim automaton does not have an enabled transition, the serach backtracks
- can be used to filter the execution or to establish an LTL property is verified

Assignment: puzzling frogs

Build a model and a property whose counter-example (in SPIN) is a sequence of moves that allows all frogs to switch side[1].

- 6 or 8 frogs, at most one on each rock, initially each half facing the other side from where it is sitting.
- can jump one step to the next rock if empty,
- can jump over a rock if occupied and following is empty.



[1] https://data.bangtech.com/algorithm/switch_frogs_to_the_opposite_side.htm