

Software Verification

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

Ahmed Rezine

IDA, Linköpings Universitet

Vårtermin 2024

# Outline

The  $CTL^*$  Temporal Logic

Model Checking CTL

$LTL$  and Büchi acceptance

Spin: flagship  $LTL$  explicit model checking

# Outline

The *CTL\** Temporal Logic

Model Checking CTL

*LTL* and Büchi acceptance

Spin: flagship *LTL* explicit model checking

# The $CTL^*$ Temporal Logic: syntax

The following are *state formulas*

- ▶  $p$  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  $f$  is a path formula

The following are *path formulas*

- ▶  $f$  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

$CTL^*$  is the set of state formulas generated by the above rules.

# The $CTL^*$ Temporal Logic: notation

- ▶ A path  $\pi = s_0s_1 \dots$  in a computation tree (obtained from a Kripke structure) is any infinite sequence of states with  $R(s_i, s_{i+1})$  for each  $i \in \mathbb{N}$
- ▶ Write  $\pi^i$  to mean the path starting from  $s_i$  in  $\pi = s_0s_1 \dots$
- ▶ Write  $M, s \models f$  to mean that state formula  $f$  holds at state  $s$  in the Kripke structure  $M$
- ▶ Write  $M, \pi \models f$  to mean that path formula  $f$  holds along path  $\pi$  in the Kripke structure  $M$

# The $CTL^*$ Temporal Logic: semantics

$f_1$  and  $f_2$  are state formulas,  $g_1$  and  $g_2$  are path formulas.

$$M, s \models p \quad \Leftrightarrow \quad p \in L(s)$$

$$M, s \models \neg f_1 \quad \Leftrightarrow \quad M, s \not\models f_1$$

$$M, s \models f_1 \vee f_2 \quad \Leftrightarrow \quad M, s \models f_1 \text{ or } M, s \models f_2$$

$$M, s \models f_1 \wedge f_2 \quad \Leftrightarrow \quad M, s \models f_1 \text{ and } M, s \models f_2$$

$$M, s \models \mathbf{E} g_1 \quad \Leftrightarrow \quad \text{there is a path } \pi \text{ from } s \text{ s.t. } M, \pi \models g_1$$

$$M, s \models \mathbf{A} g_1 \quad \Leftrightarrow \quad \text{for every path } \pi \text{ starting from } s, M, \pi \models g_1$$

## The $CTL^*$ Temporal Logic: semantics (cont.)

$f_1$  and  $f_2$  are state formulas,  $g_1$  and  $g_2$  are path formulas.

$$M, \pi \models f_1 \quad \Leftrightarrow \quad \text{if } \pi = s_0 s_1 \dots \text{ then } M, s_0 \models f_1$$

$$M, \pi \models \neg g_1 \quad \Leftrightarrow \quad M, \pi \not\models g_1$$

$$M, \pi \models g_1 \vee g_2 \quad \Leftrightarrow \quad M, \pi \models g_1 \text{ or } M, \pi \models g_2$$

$$M, \pi \models g_1 \wedge g_2 \quad \Leftrightarrow \quad M, \pi \models g_1 \text{ and } M, \pi \models g_2$$

$$M, \pi \models \mathbf{X} g_1 \quad \Leftrightarrow \quad M, \pi^1 \models g_1$$

$$M, \pi \models \mathbf{F} g_1 \quad \Leftrightarrow \quad \text{there exists a } k \geq 0 \text{ s.t. } M, \pi^k \models g_1$$

$$M, \pi \models \mathbf{G} g_1 \quad \Leftrightarrow \quad \text{for all } k \geq 0 \text{ s.t. } M, \pi^k \models g_1$$

$$M, \pi \models g_1 \mathbf{U} g_2 \quad \Leftrightarrow \quad \text{there exists a } k \geq 0 \text{ s.t. } M, \pi^k \models g_2 \\ \text{and for all } 0 \leq j < k, M, \pi^j \models g_1$$

$$M, \pi \models g_1 \mathbf{R} g_2 \quad \Leftrightarrow \quad \text{for all } j \geq 0 \text{ if for every } i < j, M, \pi^i \not\models g_1 \\ \text{then } M, \pi^j \models g_2$$

# The $CTL^*$ Temporal Logic (cont.)

**Assignment:** Express each of the following using  $f, g, \neg, \mathbf{U}, \mathbf{E}$ :

▶  $(\mathbf{F} f) = ?$

▶  $(\mathbf{G} f) = ?$

▶  $(\mathbf{A} f) = ?$

▶  $(f\mathbf{R} g) = ?$

# Outline

The *CTL\** Temporal Logic

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 formulas*

- ▶  $p$  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 formulas*

- ▶  $f$  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

# Branching Time Logic (CTL)

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

The following are *state formulas*

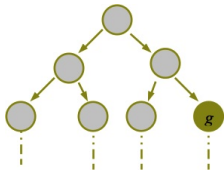
- ▶  $p$  if  $p \in AP$
- ▶  $\neg f$ ,  $f \wedge g$  and  $f \vee g$  if  $f, g$  are state formulas
- ▶ **A** $f$ , **E** $f$  if  $f$  a path formula

The following are *path formulas*

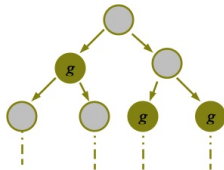
- ▶  $f$  if it is also a state formula
- ▶  $\neg f$ ,  $f \wedge g$ ,  $f \vee g$ , **X** $f$ , **F** $f$ , **G** $f$ ,  $f$ **U** $g$  and  $f$ **R** $g$  if  $f, g$  are path formulas

The most used operators are:

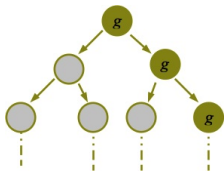
- ▶  $M, s_0 \models \mathbf{EF} g$ ,
- ▶  $M, s_0 \models \mathbf{AF} g$ ,
- ▶  $M, s_0 \models \mathbf{EG} g$ ,
- ▶  $M, s_0 \models \mathbf{AG} g$



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



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

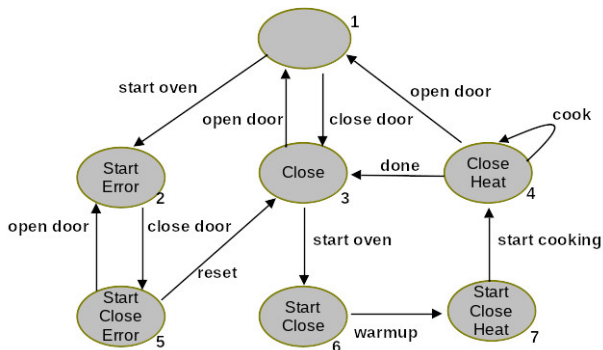


$$M, s_0 \models \mathbf{EG} g$$



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

# Model checking CTL properties



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

# The UPPAAL model checker

The screenshot displays the UPPAAL model checker interface for a multi-process model of a train gate system. The main window is titled "UPPAAL" and shows the file path: `/home/rahmed/Documents/codes/uppaal/uppaal-4.0.14/demo/train-gate.xml`.

**Editor:** The "Enabled Transitions" list shows the following transitions:

- `appr[0]: Train(0) -> Gate`
- `appr[1]: Train(1) -> Gate`
- `appr[4]: Train(4) -> Gate`
- `leave[2]: Train(2) -> Gate`

**Simulation Trace:** The trace shows the sequence of events:

```
(Safe, Safe, Safe, Safe, Safe, Free)
appr[2]: Train(2) -> Gate
(Safe, Safe, Appr, Safe, Safe, Occ)
appr[3]: Train(3) -> Gate
(Safe, Safe, Appr, Appr, Safe, -)
stop(tail): Gate -> Train(3)
(Safe, Safe, Appr, Stop, Safe, Occ)
Train(2)
(Safe, Safe, Cross, Stop, Safe, Occ)
```

**Model Structure:** The model consists of several processes:

- Train(0) to Train(4):** Each train process has states: Safe, Appr, Start, Cross, and Stop. Transitions are labeled with conditions like `x >= 3` and `x <= 5`, and actions like `leave[i]!` and `go[i]?`.
- Gate:** The Gate process has states: Free, Occ, and Stop. It manages the queue of trains with transitions like `enqueue(e)` and `dequeue()`.

**Trace File:** The trace file shows the sequence of events for each process:

Process	State	Event
Train(0)	Safe	
Train(1)	Safe	
Train(2)	Safe	appr[2]
Train(3)	Safe	
Train(4)	Safe	
Gate	Free	
Gate	Occ	appr[2]
Gate	Occ	appr[3]
Gate	Occ	appr[4]
Gate	Occ	stop(tail)!
Train(2)	Stop	
Train(3)	Stop	
Train(4)	Stop	
Train(0)	Cross	

# Outline

The *CTL\** Temporal Logic

Model Checking CTL

***LTL*** and Büchi acceptance

Spin: flagship *LTL* explicit model checking

# 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:

- ▶  $f$  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

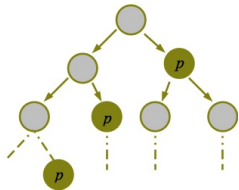


- ▶ invariance:  $\mathbf{G}(\neg \text{Error})$
- ▶ guarantee:  $\mathbf{F}(\text{Ok})$
- ▶ response:  $\text{Req} \implies \mathbf{F}(\text{Ack})$
- ▶ precedence:  $\text{Req} \implies (\text{Busy} \mathbf{U} \text{Ack})$
- ▶ progress:  $\mathbf{GF}(\text{Move})$
- ▶ stability:  $\mathbf{FG}(\text{Stable})$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

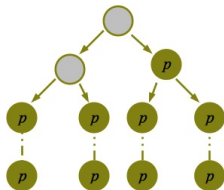
# Linear Time Logic (*LTL*)

*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



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

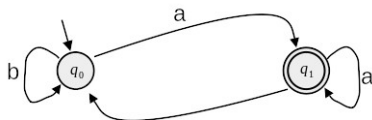


$\mathbf{A}(\mathbf{FG} p)$  is in *LTL* but not *CTL*. Stability: there is a point after which  $p$  always hold.

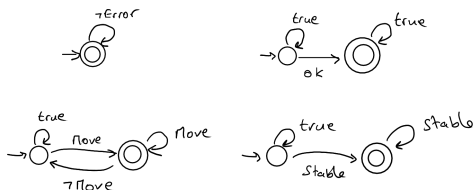
# Büchi automata

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

- ▶  $Q$  a finite set of states
- ▶  $\Sigma$  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:  $\mathbf{G}(\neg\text{Error})$
- ▶ guarantee:  $\mathbf{F}(\text{Ok})$
- ▶ progress:  $\mathbf{GF}(\text{Move})$
- ▶ stability:  $\mathbf{FG}(\text{Stable})$
- ▶ weak fairness:  $\mathbf{GF}(\neg\text{En} \vee \text{Ex})$
- ▶ strong fairness:  $\mathbf{GF}(\text{En}) \implies \mathbf{GF}(\text{Ex})$

# Outline

The  $CTL^*$  Temporal Logic

Model Checking CTL

$LTL$  and Büchi acceptance

Spin: flagship  $LTL$  explicit model checking

# Promela Models

```
1  mtype = {MSG, ACK};
2  chan toS = ...
3  chan toR = ...
4  bool flag;
5
6  proctype Sender(){
7      /* Process body */
8      ...
9  }
10
11 proctype Receiver() {
12     ...
13 }
14
15 init{
16     /* process creation */
17     ...
18 }
```

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 processes
- ▶ bounded process creation

## Promela Models (cont.)

A process:

- ▶ is defined by a proctype definition
- ▶ executes concurrently with all other processes, 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

```
1  proctype Sender(chan a){
2      ...
3  }
4
5  init{
6      chan c = [1] of {bit};
7      int pid2 = run Sender(c);
8  }
9
10 active[3] proctype Writer(){
11     ...
12 }
```

## 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

```
1  proctype Sender(chan in; chan out){
2      bit sndB, rcvB; /* local variables */
3      do
4          :: out ! MSG, sndB ->
5              in = ACK, rcvB;
6              if
7                  :: sndB == rcvB -> sndB = 1 - sndB
8                  :: else -> skip
9
10             fi
11     od
12 }
```

# Variables and Types

```
1  /*basic types*/
2  bit turn=1;
3  bool flag;
4  byte counter;
5  short s;
6  int msg;
7
8  /*arrays*/
9  byte a[27];
10 bit flags[4];
```

```
1  /*records*/
2  typedef Record{
3      short f1;
4      byte f2;
5  }
6
7  Record rr;
8
9  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

```
1  proctype monitor(){
2      assert(n <= 3);
3  }
4
5  proctype receiver(){
6      ...
7      toReceiver ? msg;
8      assert(msg != ERROR);
9      ...
10 }
```

# 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

```
1  if  
2  :: choice1 -> stat1,1; stat1,2; ...  
3  :: choice2 -> stat2,1; stat2,2; ...  
4  ...  
5  :: choicen -> statn,1; statn,2; ...  
6  (:: else -> statn,1; statn,2; ...)  
7  fi;
```

- ▶ if at least one of the *choice*<sub>*i*</sub> is executable, then SPIN non deterministically chooses one of them
- ▶ if none of the *choice*<sub>*i*</sub> is executable, the if statement is blocked
- ▶ “->” is used to separate the guards from the statements that follow it (actually equivalent to “;”).

# do-statement

```
1 do
2  :: choice1 -> stat1,1; stat1,2; ...
3  :: choice2 -> stat2,1; stat2,2; ...
4  ...
5  :: choicen -> statn,1; statn,2; ...
6 od;
```

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

# Communication

```
1  chan <name> = [<dim>] of {<t1, <t2>, ...};  
2  
3  chan c = [1] of {bit};  
4  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:

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

---

- ▶ Receiving: fetching a message from a channel:

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

---

# The **atomic** Statement

1 | `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

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

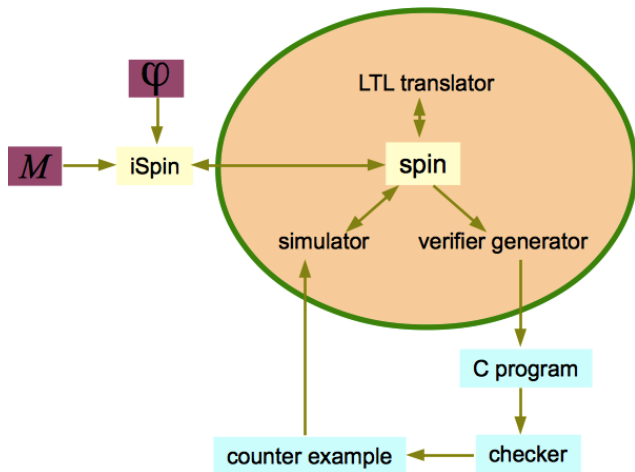
---

- ▶ more efficient than atomic: no intermediate states
- ▶ only deterministic statements allowed
- ▶ error if *stat*<sub>*i*</sub>, for *i* > 1, blocked

# Examples of Promela constructs

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

# (i) Spin Architecture



# Mutex: First Attempt

```
1 bit flag;          /* signal entering/leaving the CS */
2 byte mutex;       /* number of processes in the CS */
3
4 proctype P(bit i){
5     flag != 1;
6     flag = 1;
7     mutex++;
8     mutex--;
9     flag = 0;
10 }
11
12 proctype monitor(){
13     assert(mutex!=2);
14 }
15
16 init{
17     atomic{ run P(0); run P(1); run monitor();}
18 }
```

## Mutex: Second Attempt

```
1  bit x, y;           /* signal entering/leaving the CS */
2  byte mutex;        /* number of processes in the CS */
3
4  active proctype A(){
5      x = 1;
6      y == 0;
7      mutex++;
8      mutex--;
9      x = 0;
10 }
11
12 active proctype B(){
13     y = 1;
14     x == 0;
15     mutex++;
16     mutex--;
17     y = 0;
18 }
19
20 proctype monitor(){
21     assert(mutex!=2);
22 }
```

# Mutex: Dekker's Algorithm

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

---

<pre>1 active proctype A(){ 2   x = 1; 3   turn = B_TURN; 4   y == 0 5        (turn == A_TURN); 6   mutex++; 7   mutex--; 8   x = 0; 9 }</pre>	<pre>1 active proctype B(){ 2   y = 1; 3   turn = A_TURN; 4   x == 0 5        (turn == B_TURN); 6   mutex++; 7   mutex--; 8   y = 0; 9 }</pre>
--	--

---

```
1 proctype monitor(){
2   assert(mutex!=2);
3 }
```

---

# Mutex: Bakery Algorithm

```
1 | byte turn[2];           /* who's turn is it? */
2 | byte mutex;           /* number of processes in the CS */
3 |
4 | proctype P(bit i){
5 | do
6 | :: turn[i] = 1;
7 |    turn[i] = turn[1-i] + 1;
8 |    (turn[1-i] == 0) || (turn[i] < turn[1-i]);
9 |    mutex++;
10 |   mutex--;
11 |   turn[i] = 0;
12 | od
13 | }
14 |
15 | proctype monitor(){
16 |     assert(mutex!=2);
17 | }
18 |
19 | init {
20 |     atomic{run P(0); run P(1); run monitor()}
21 | }
```

# 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

invariance	$G(p)$
response	$G(p \Rightarrow F(q))$
precedence	$G(p \Rightarrow (qUr))$
objective	$G(p \Rightarrow F(q  r))$

## Liveness properties:

- ▶ "something good eventually happens"
- ▶ termination: the system eventually terminates
- ▶ response: whenever  $X$  occurs,  $Y$  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.

$ltl \{[] p\}$
$ltl \{[](p \Rightarrow \langle \rangle q)\}$
$ltl \{[] (p \Rightarrow (q U r))\}$
$ltl \{[] (p \Rightarrow \langle \rangle (q    r))\}$

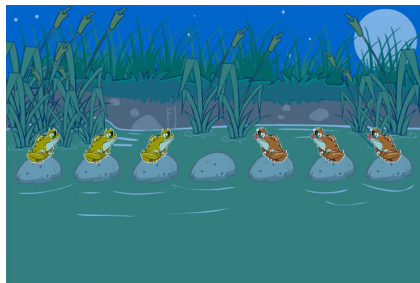
# never Claims

- ▶ LTL formulas can be captured using Büchi automata
- ▶ SPIN uses **never** claims to capture Büchi automata
- ▶ capture finite behaviors and  $\omega$ -acceptance cycles
- ▶ the kripke structure and the never claim execute in lockstep
- ▶ if the claim automaton does not have an enabled transition, the search 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<sup>1</sup>.

- ▶ 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.



<sup>1</sup>[https://data.bangtech.com/algorithm/switch\\_frogs\\_to\\_the\\_opposite\\_side.htm](https://data.bangtech.com/algorithm/switch_frogs_to_the_opposite_side.htm)