



ALBERT-LUDWIGS-  
UNIVERSITÄT FREIBURG

# Network Protocol Design and Evaluation

## 05 - Validation, Part III

**Stefan Rührup**

University of Freiburg  
Computer Networks and Telematics  
Summer 2009



# Overview

- ▶ **In the first parts of this chapter:**
  - Validation models in Promela
  - Defining and checking correctness claims with SPIN
  
- ▶ **In this part:**
  - Correctness Claims with Linear Temporal Logic
  - Example (continued): Validation of the Alternating Bit Protocol with LTL and SPIN

slides referring to this example are marked with



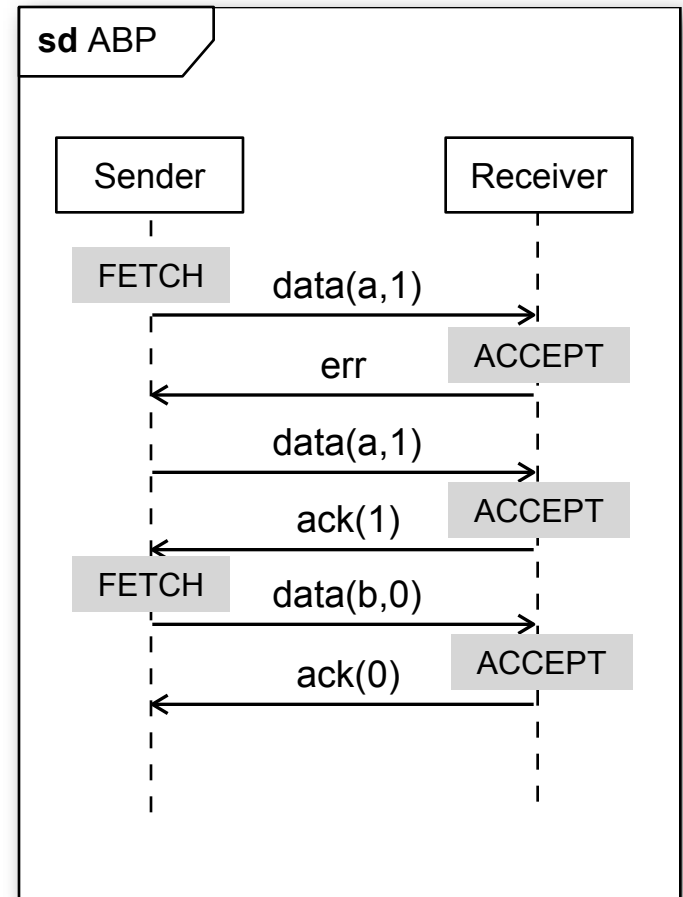
ABP

# Temporal Logic

- ▶ Transforming requirements into never claims is not always easy
- ▶ A more convenient way of formalization is by using **Linear Temporal Logic (LTL)**
- ▶ Example for describing a valid execution sequences:  
*Every state satisfying  $p$  is eventually followed by one which satisfies  $q$ .*  
In LTL:  $\Box(p \rightarrow \Diamond q)$
- ▶ LTL formulae are often easier to understand than never claims

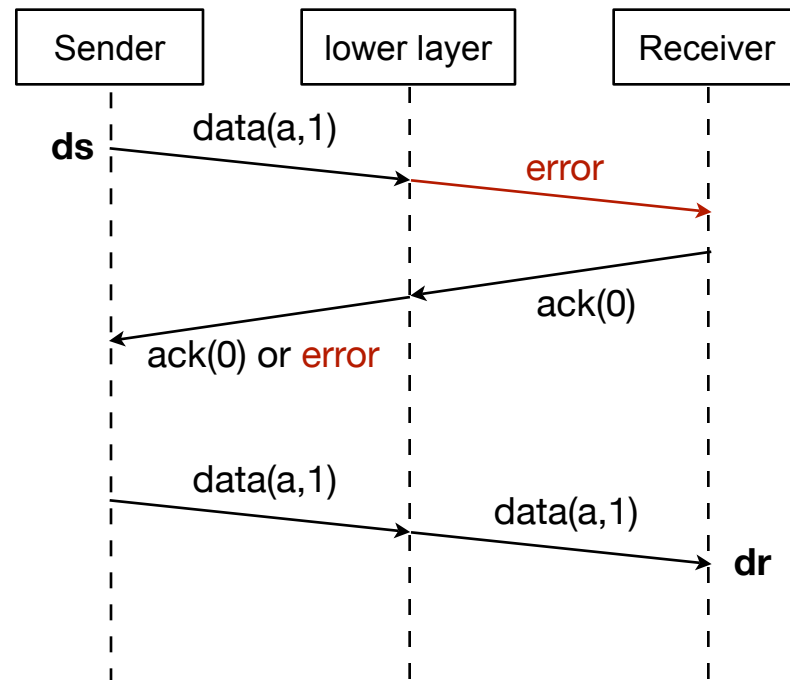
# Motivation: LTL and Validation (1)

- ▶ Example (Alternating Bit Protocol):  
We want to assert that a data message is finally received (unless there is an error cycle)
- ▶ More precisely: After a message has been sent, there might be errors and retransmissions until it is received by the receiver or an error occurs infinitely often
- ▶ We can express this in LTL ...



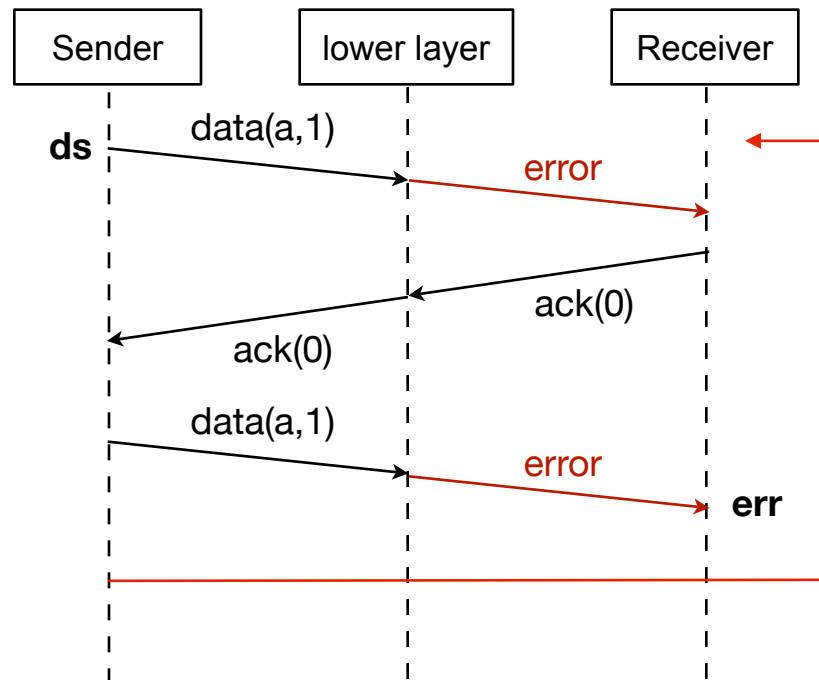
# Desired Behaviour (1)

- ▶ Every data message sent is finally received by the receiver



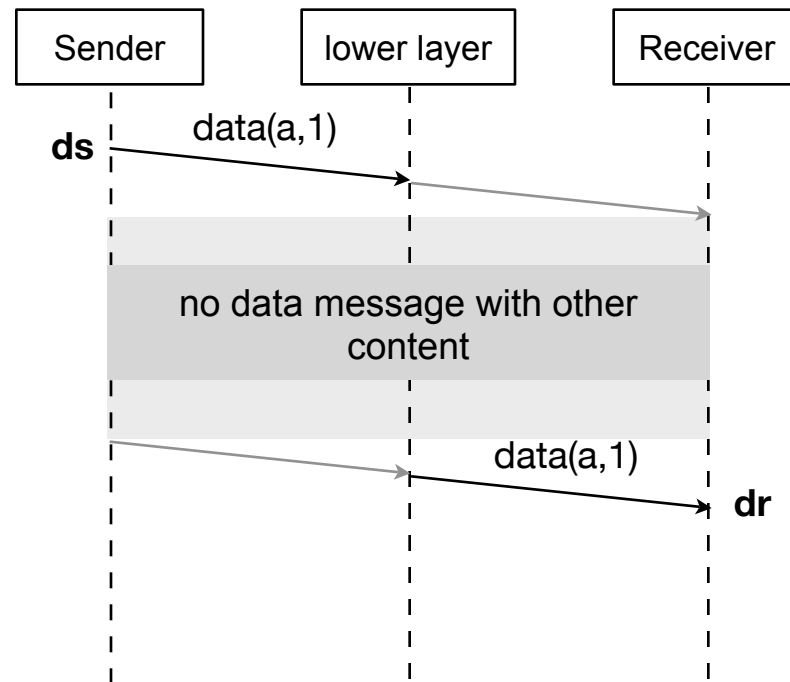
# Desired Behaviour (2)

- ▶ But there might be an error cycle due to repeated message distortion by the lower layer



# Desired Behaviour (3)

- ▶ However, between sending and receiving a data message, there is no other data message transmitted



# Motivation: LTL and Validation (2)

- ▶ Claim: *After a message  $x$  has been sent, there might be errors and retransmissions (but no other data is sent) until  $x$  is received by the receiver or an error occurs infinitely often*
- ▶ We define: **ds** - data sent, **dr** - data received  
**od** - other data sent (with other content),  
**err** - error message received
- ▶ A little bit more formal:  
Always after **ds** there is no **od** until (**dr** or **err**)
- ▶ In LTL:  $\square(\mathbf{ds} \rightarrow \neg\mathbf{od} \text{ U } (\mathbf{dr} \vee \mathbf{err}))$   
(Always **ds** implies not **od** until (**dr** or **err**))



# Temporal Logic

- ▶ Why “Temporal Logic”?
- ▶ Logic formulas expressing some system properties are not *statically* true or false
- ▶ Formulas may change their truth values *dynamically* as the system changes its state  
→ **Temporal Logic**
- ▶ LTL formulae are defined over infinite transition sequences (“runs”). *Linear* refers to single sequential runs

# LTL Formulae

- ▶ LTL extends propositional logic by modal operators
- ▶ Well-formed LTL formulae
  - Propositional state formulae, including true or false are well-formed
  - If  $p$  and  $q$  are well-formed formulae, then  $\alpha p$ ,  $p \beta q$ , and  $(p)$  are well-formed formulae, where  $\alpha$  and  $\beta$  are unary/binary temporal operators
- ▶ Grammar:

```
ltl ::= operand | ( ltl ) | ltl binary_operator ltl |  
      unary_operator ltl
```

(where operand is either true, false, or a user-defined symbol)

# Linear Temporal Logic

## ▶ LTL Operators:

Operator	Description	Definition
X	Next	$\sigma[i] \models X p \Leftrightarrow \sigma_{i+1} \models p$
U	Weak Until	$\sigma[i] \models (p U q) \Leftrightarrow \sigma_i \models q \vee (\sigma_i \models p \wedge \sigma[i+1] \models (p U q))$
$U$	Strong Until	$\sigma[i] \models (p U q) \Leftrightarrow \sigma_i \models (p U q) \wedge \exists j, j \geq i \sigma_j \models q$
$\square$	Always	$\sigma \models \square p \Leftrightarrow \sigma \models (p U \text{false})$
$\diamond$	Eventually	$\sigma \models \diamond p \Leftrightarrow \sigma \models (\text{true} U p)$

$\sigma_i$  = i-th element of the run  $\sigma$

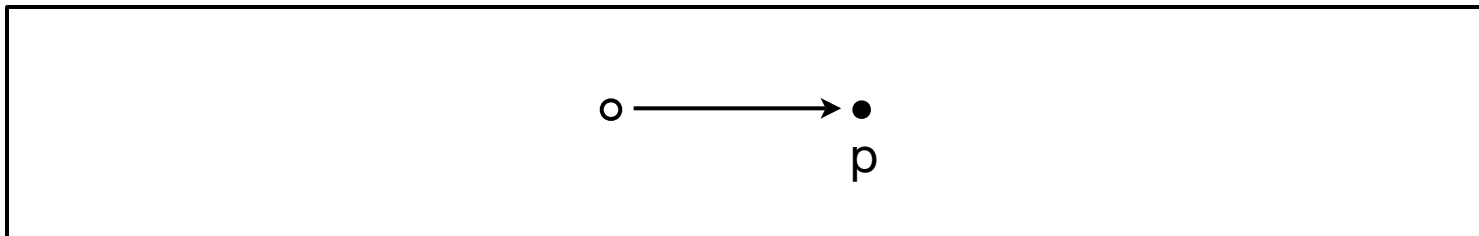
$\sigma[i]$  = suffix of  $\sigma$  starting at the i-th element

# LTL Operators (1)

- ▶ **Next**

$X p$  = Property  $p$  is true in the following state

Operator	Description	Definition
$X$	Next	$\sigma[i] \models X p \Leftrightarrow \sigma_{i+1} \models p$

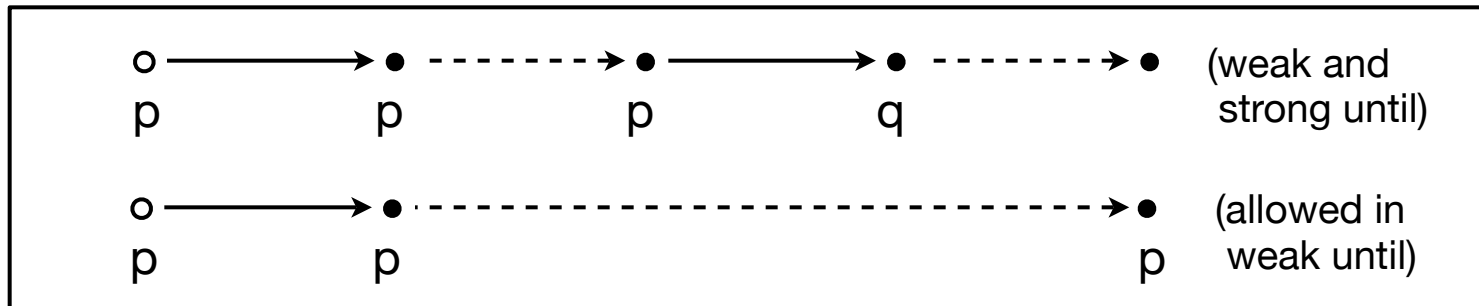


# LTL Operators (2)

## ▶ Until

$p \text{ U } q$  = Property  $p$  holds until  $q$  becomes true. After that  $p$  does not have to hold any more. Weak until does not require that  $q$  ever becomes true

Operator	Description	Definition
W	Weak Until	$\sigma[i] \models (p \text{ W } q) \Leftrightarrow \sigma_i \models q \vee (\sigma_i \models p \wedge \sigma[i+1] \models (p \text{ W } q))$
U	Strong Until	$\sigma[i] \models (p \text{ U } q) \Leftrightarrow \sigma_i \models (p \text{ W } q) \wedge \exists j, j \geq i \sigma_j \models q$



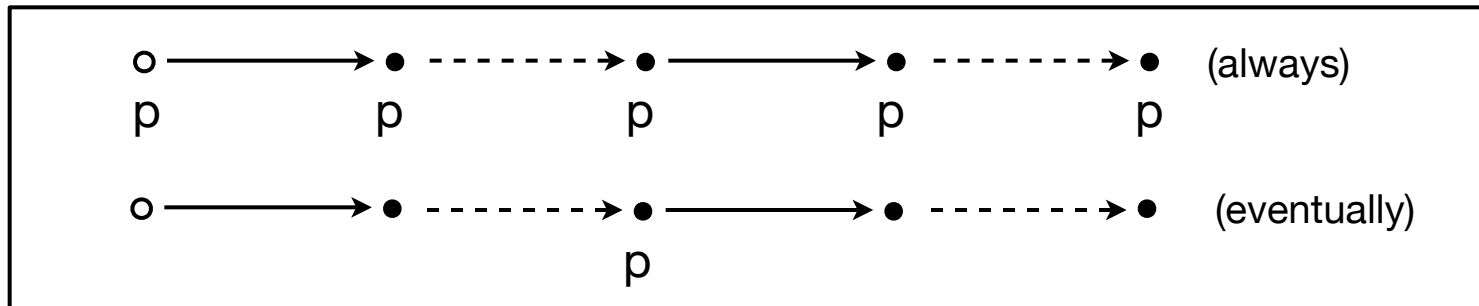
# LTL Operators (3)

▶ **Always and Eventually**

$\square p$  = Property  $p$  remains invariantly true.

$\diamond p$  = Property  $p$  becomes eventually true at least once in a run

Operator	Description	Definition
$\square$	Always (also called Globally, G)	$\sigma \models \square p \Leftrightarrow \sigma \models (p \text{ W false})$
$\diamond$	Eventually (also called Finally, F)	$\sigma \models \diamond p \Leftrightarrow \sigma \models (\text{true U } p)$



# LTL Rules

Alternative definition of **Weak Until**

$$p \text{ W } q \equiv (p \text{ U } q) \vee \Box p$$

LTL Formula	Equivalent
$\neg \Box p$	$\Diamond \neg p$
$\neg \Diamond p$	$\Box \neg p$
$\Box (p \wedge q)$	$\Box p \wedge \Box q$
$\Diamond (p \vee q)$	$\Diamond p \vee \Diamond q$
$\neg (p \text{ U } q)$	$\neg q \text{ W } (\neg p \wedge \neg q)$
$p \text{ U } (q \vee r)$	$(p \text{ U } q) \vee (p \text{ U } r)$
$(p \text{ U } q) \vee r$	$(p \text{ U } r) \vee (p \text{ U } r)$
$\Box \Diamond (p \vee q)$	$\Box \Diamond p \vee \Box \Diamond q$
$\Diamond \Box (p \wedge q)$	$\Diamond \Box p \wedge \Diamond \Box q$

[Holzmann 2003]

# Using LTL (1)

- ▶ **A simple property:** Every system state in which  $p$  is true is eventually followed by a system state in which  $q$  is true
- ▶ Can't we simply express this by the implication  $p \rightarrow q$  ?
- ▶ No,  $p \rightarrow q$  has no temporal operators. It is simply  $(!p \vee q)$  and applies as a propositional claim to the first system state.



# Using LTL (2)

- ▶ We can apply this claim to all states by using the always operator:

$$\square(p \rightarrow q)$$

- ▶ There is still the temporal implication missing: “q is *eventually* reached”:

$$\square(p \rightarrow \diamond q)$$

# Standard Correctness Properties

LTL Formula	English	Type
$\Box p$	always p	Invariance
$\Diamond p$	eventually p	Guarantee
$p \rightarrow \Diamond q$	p implies eventually q	Response
$p \rightarrow q \cup r$	p implies q until r	Precedence
$\Box \Diamond p$	always eventually p	Recurrence (progress)
$\Diamond \Box p$	eventually always p	Stability (non-progress)
$\Diamond p \rightarrow \Diamond q$	eventually p implies eventually q	Correlation

[Holzmann 2003]

# LTL in SPIN

- ▶ **Spin accepts ...**
  - propositional symbols, including *true* and *false*
  - temporal operators *always* ( [ ] ), *eventually* ( <> ), and *strong until* ( U )
  - logical operators *and* ( && ), *or* ( || ) and *not* ( ! )
  - Implication ( -> ) and equivalence ( <-> )
- ▶ Arithmetic and relational expressions are not supported  
But they can be replaced by a propositional symbol.  
Example: `#define q (seqno <= last + 1)`

# Using LTL with SPIN

- ▶ Specify an LTL property
- ▶ Generate symbols: `#define p expression`
- ▶ Generate a never claim:  
`spin -f 'LTL formula' >> claim.ltl`
- ▶ Validate your model:
  - Generate the verifier:  
`spin -a model.pml -N claim.ltl`
  - Compile and run the verifier
- ▶ **Recommendation:** Use the LTL property manager of XSPIN

# Example

- ▶ The LTL formula  $\Box(p \rightarrow \langle \rangle q)$  can be translated into the following never claim:

```
never { /* ! $\Box(p \rightarrow \langle \rangle q)$  */
T0_init:
  if
  :: (! ((q)) && (p)) -> goto accept_S4
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! ((q))) -> goto accept_S4
  fi;
}
```

# The LTL Property Manager of XSPIN

LTL formula

SPIN will ask for definitions of unknown symbols if not specified

The never claim generated from the negated LTL formula

Result of verification

Output of the verifier

The screenshot shows the XSPIN LTL Property Manager interface. It features a 'Formula' field containing `[] (msg0 -> <> msg1)`, a 'Load...' button, and a set of operators: `[]`, `<>`, `U`, `->`, `and`, `or`, and `not`. Below this is a radio button selection for 'Property holds for:' with 'All Executions (desired behavior)' selected. A 'Notes' section contains the text 'Use Load to open a file or a template.' The 'Symbol Definitions' section contains `#define msg0 (toR?[data(_,0)])` and `#define msg1 (toR?[data(_,1)])`. The 'Never Claim' section contains a multi-line comment block and a `never { /* !([] (msg0 -> <> msg1)) */` line. A 'Generate' button is located to the right of this section. The 'Verification Result' section shows the word 'valid' in a green oval, with a 'Run Verification' button to its right. The bottom section contains the output of the verifier, including a warning about stutter-invariance and state reachability information. The interface also includes 'Help', 'Clear', 'Close', and 'Save As...' buttons at the bottom.

1

2

3

4

# Example: Validation of ABP with LTL

## ► Overview

1. Build the Promela model (`alternating.pml`)
2. Define symbols `ds` (data sent), `dr` (data received), ...
3. Define the correctness claim in LTL:
  - $ds \rightarrow \neg od \text{ U } (dr \vee err)$
4. Generate a never claim

```
spin -f "[)]" >> alternating.ltl
```
5. Generate the verifier

```
spin -a alternating.pml -N alternating.ltl
```
6. Build an run the verifier

# Validation of ABP with LTL

## ► Overview

1. Build the Promela model (`alternating.pml`) **done** ✓
2. Define symbols `ds` (data sent), `dr` (data received), ...
3. Define the correctness claim in LTL:
  - `ds` → ¬`od` U (`dr` ∨ `err`)
4. Generate a never claim
 

```
spin -f “)” >> alternating.ltl
```
5. Generate the verifier
 

```
spin -a alternating.pml -N alternating.ltl
```
6. Build an run the verifier





# Defining Symbols (1)

- ▶ **Symbols have to be defined for**
  - ds - data sent
  - dr - data received
  - od - other data sent (with other content),
  - err - error message received
- ▶ These symbols refer to receive operations on message channels
- ▶ Executability of any such operation can be expressed by the poll statement:  
`channel?[message]`

# Defining Symbols (2)

Recall: Sender and Receiver are connected via these four channels

These channels have to be defined *globally*

```

#define N 2
#define MAX 8
#define FETCH mt = (mt+1)%MAX
#define ACCEPT assert(mr==(last_mr+1)%MAX)

mtype = { data, ack, error }

proctype lower_layer(chan fromS, toS, fromR, toR) {...}
proctype Sender(chan in, out) {...}
proctype Receiver(chan in, out) {...}

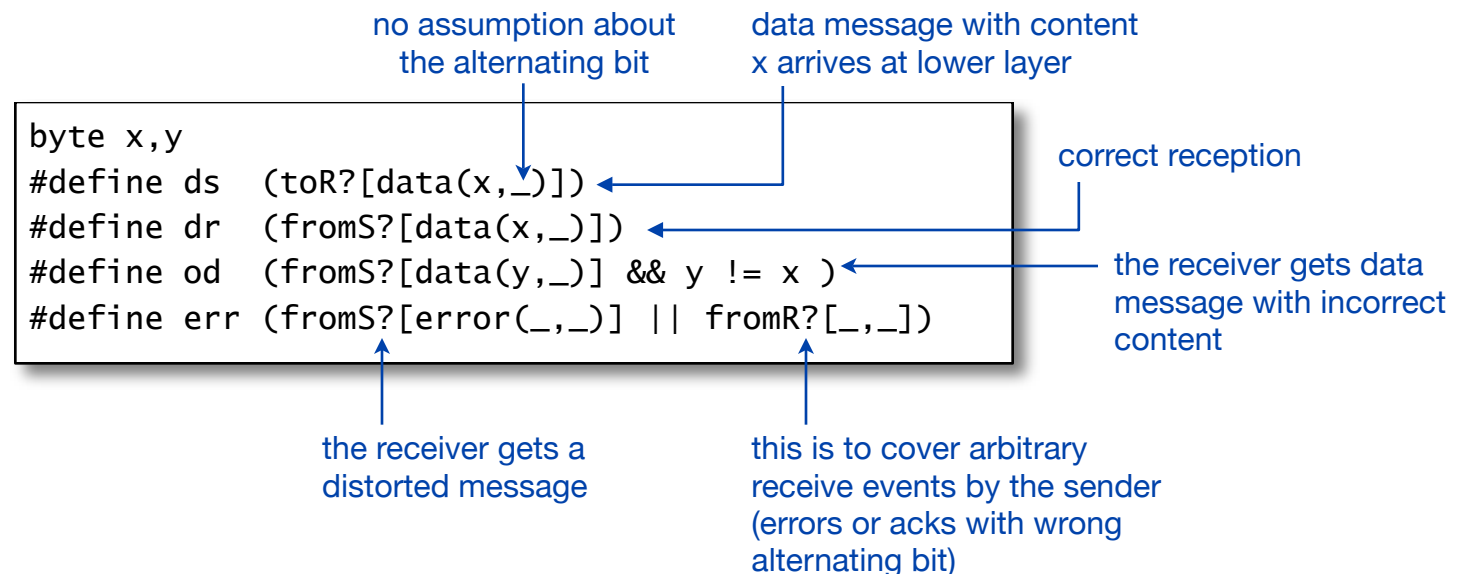
chan fromS = [N] of { byte, byte, bit };
chan toR = [N] of { byte, byte, bit };
chan fromR = [N] of { byte, bit };
chan toS = [N] of { byte, bit };

init {
    atomic {
        run Sender(toS, fromS);
        run Receiver(toR, fromR);
        run lower_layer(fromS, toS, fromR, toR) }
}

```

# Defining Symbols (3)

- ▶ Symbols to be defined:  
 ds - data sent, dr - data received  
 od - other data sent, err - error message received



# Defining Symbols (4)

- ▶ Symbols to be defined:  
ds - data sent, dr - data received  
od - other data sent, err - error message receiveden
- ▶ **Alternative definition** with constant values:

```
#define N 2
#define MAX 3
#define FETCH mt = (mt+1)%MAX
#define ACCEPT assert(mr==(last_mr+1)%MAX)
```

← data content restricted to values {0,1,2}

```
#define ds (toR?[data(0,_)])
#define dr (fromS?[data(0,_)])
#define od (fromS?[data(1,_)] || fromS?[data(2,_)])
#define err (fromS?[error(_, _)] || fromR?[_,_])
```

# Generating the never claim (1)

- ▶ The never claim captures the negated LTL formula
- ▶ Negation:

$$\begin{aligned}
 & \neg \Box (\mathbf{ds} \rightarrow \neg \mathbf{od} \cup (\mathbf{dr} \vee \mathbf{err})) \\
 \Leftrightarrow & \Diamond \neg (\mathbf{ds} \rightarrow \neg \mathbf{od} \cup (\mathbf{dr} \vee \mathbf{err})) \\
 \Leftrightarrow & \Diamond \neg (\neg \mathbf{ds} \vee (\neg \mathbf{od} \cup (\mathbf{dr} \vee \mathbf{err}))) \\
 \Leftrightarrow & \Diamond (\mathbf{ds} \wedge \neg (\neg \mathbf{od} \cup (\mathbf{dr} \vee \mathbf{err}))) \\
 \Leftrightarrow & \Diamond (\mathbf{ds} \wedge (\neg (\mathbf{dr} \vee \mathbf{err}) \text{ W } (\mathbf{od} \wedge \neg (\mathbf{dr} \vee \mathbf{err})))) \\
 \Leftrightarrow & \Diamond (\mathbf{ds} \wedge ((\neg \mathbf{dr} \wedge \neg \mathbf{err}) \text{ W } (\mathbf{od} \wedge \neg \mathbf{dr} \wedge \neg \mathbf{err})))
 \end{aligned}$$

- ▶ Luckily, SPIN can do the negation and generate the never claim from the negated formula

# Generating the never claim (2)

**LTL:** `[] (ds -> (!od) U (dr || err))`

SPIN

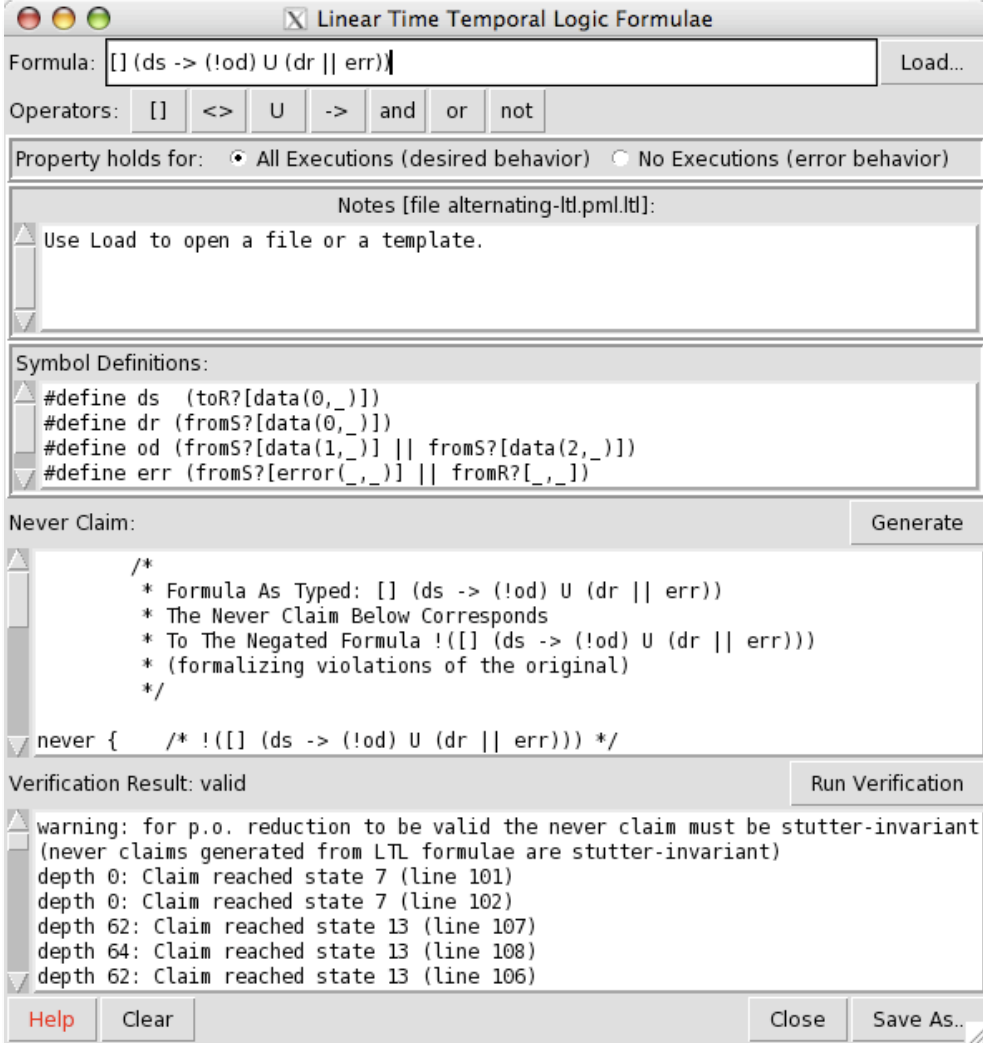


```

never { /* !([] (ds -> (!od) U (dr || err))) */
T0_init:
    if
        :: (! ((dr)) && ! ((err)) && (ds)) -> goto accept_S4
        :: (! ((dr)) && ! ((err)) && (ds) && (od)) -> goto accept_all
        :: (1) -> goto T0_init
    fi;
accept_S4:
    if
        :: (! ((dr)) && ! ((err))) -> goto accept_S4
        :: (! ((dr)) && ! ((err)) && (od)) -> goto accept_all
    fi;
accept_all:
    skip
}

```

# Validation with XSPIN



The screenshot shows the XSPIN Linear Time Temporal Logic Formulae editor. The main window contains the following elements:

- Formula:** `[] (ds -> (!od) U (dr || err))`
- Operators:** `[]`, `<>`, `U`, `->`, `and`, `or`, `not`
- Property holds for:**  All Executions (desired behavior)  No Executions (error behavior)
- Notes [file alternating-ldt.pml.ltl]:** Use Load to open a file or a template.
- Symbol Definitions:**

```
#define ds (toR?[data(0,_)])
#define dr (fromS?[data(0,_)])
#define od (fromS?[data(1,_) ] || fromS?[data(2,_)])
#define err (fromS?[error(,_) ] || fromR?[,_) ]
```
- Never Claim:**

```
/*
 * Formula As Typed: [] (ds -> (!od) U (dr || err))
 * The Never Claim Below Corresponds
 * To The Negated Formula !([] (ds -> (!od) U (dr || err)))
 * (formalizing violations of the original)
 */
never { /* !([] (ds -> (!od) U (dr || err))) */
```
- Verification Result:** valid
- Warnings:**

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 7 (line 101)
depth 0: Claim reached state 7 (line 102)
depth 62: Claim reached state 13 (line 107)
depth 64: Claim reached state 13 (line 108)
depth 62: Claim reached state 13 (line 106)
```

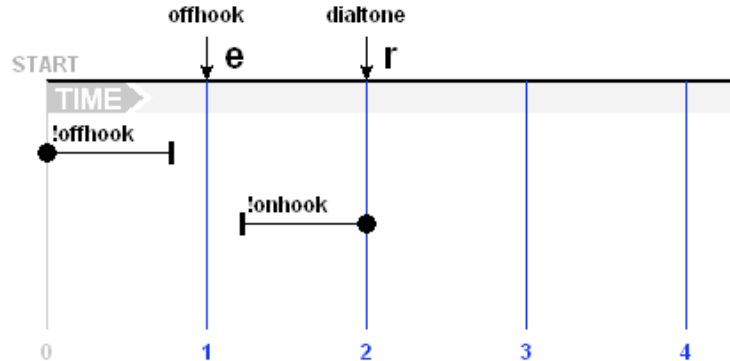
A green arrow points to the "Verification Result: valid" section.

**... Result: valid.**

# Timelines

- ▶ A further method to define temporal claims: **Timelines**
- ▶ Timelines define causal relations between events

- ▶ Graphical representation:

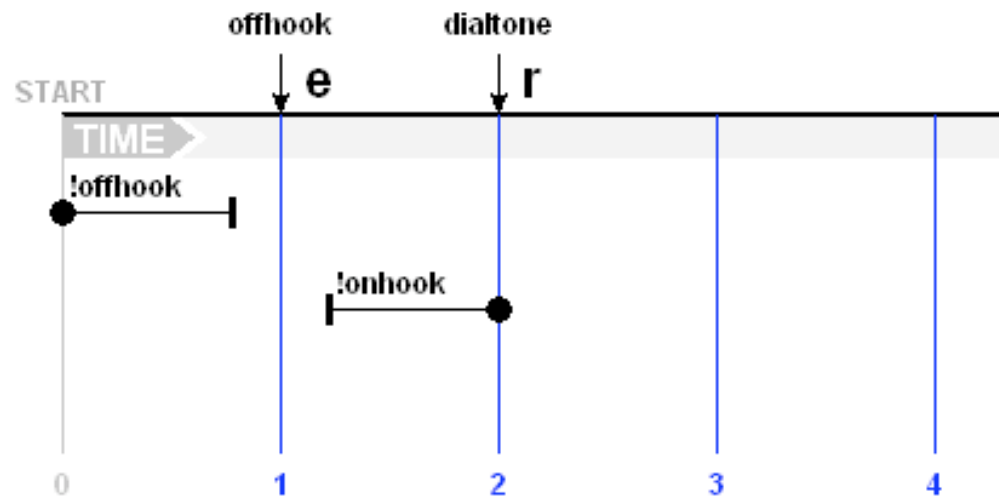


- ▶ **The Timeline Editor**
- ▶ Download: <http://www.bell-labs.com/project/timeedit/>
- ▶ [Smith, Holzmann, Etesami: “Events and Constraints a graphical editor for capturing logic properties of programs”, RE’01, pp. 14-22, Aug. 2001]



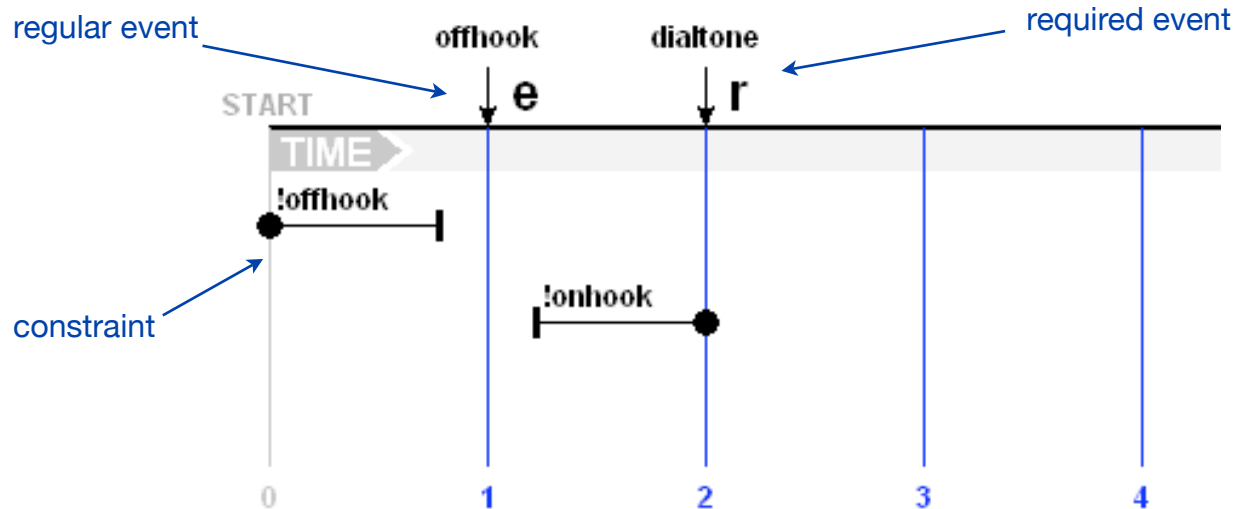
# Example (1)

- ▶ Requirement:  
*When the user lifts the receiver, the phone should provide a dialtone. (There are no intervening onhook events)*
- ▶ Timeline specification:



# Example (2)

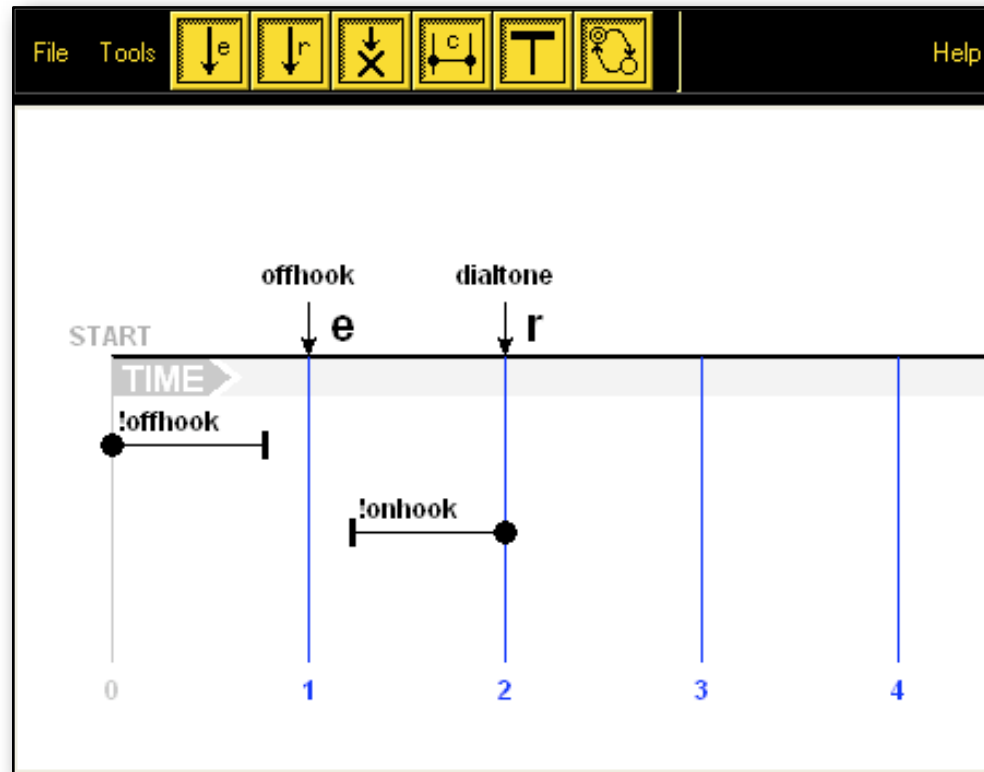
- ▶ Requirement: *When the user lifts the receiver, the phone should provide a dialtone.*



In **LTL**:  $\neg(\neg\text{offhook} \text{ U } (\text{offhook} \wedge \text{X}\square(\neg\text{dialtone} \wedge \neg\text{onhook})))$

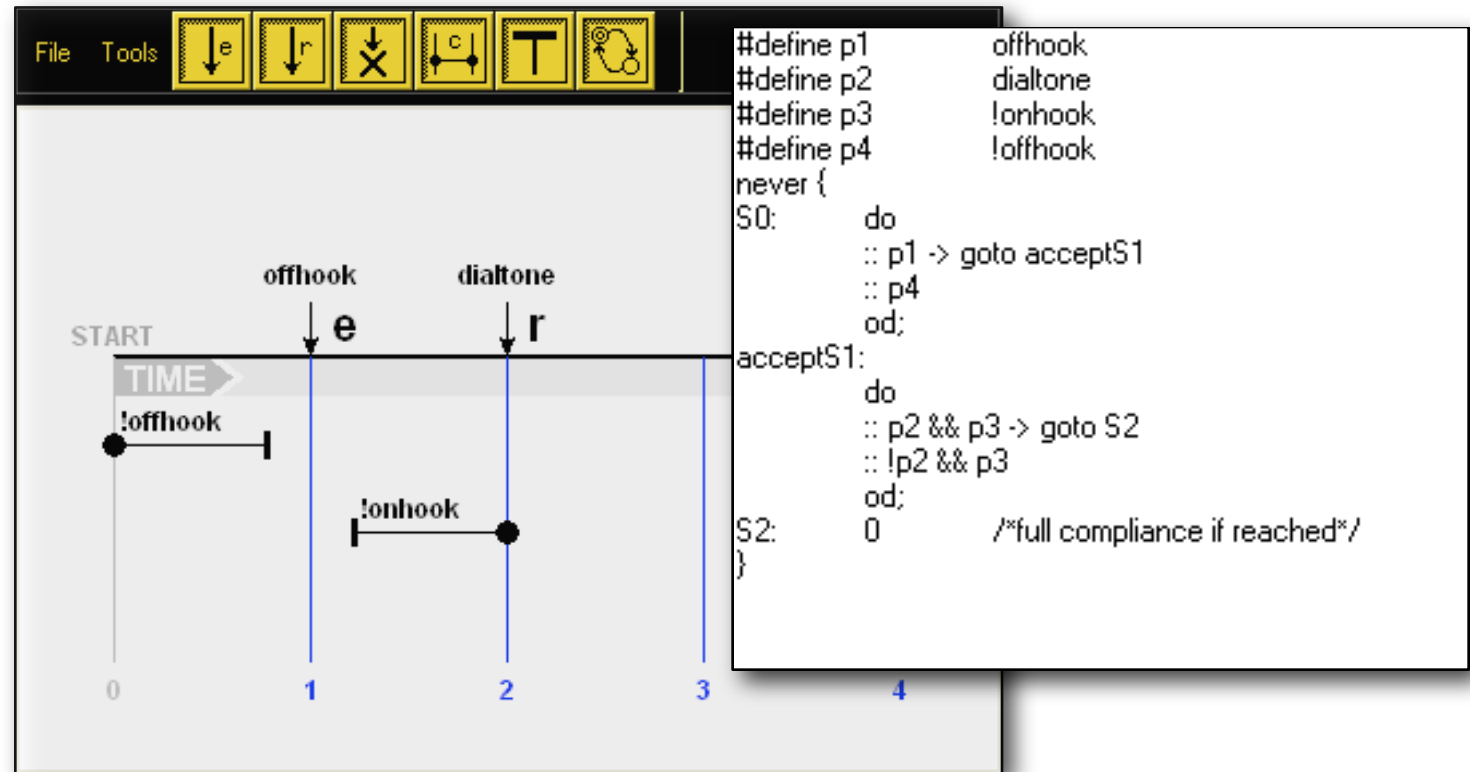
# The Timeline Editor (1)

- ▶ Timeline specification:



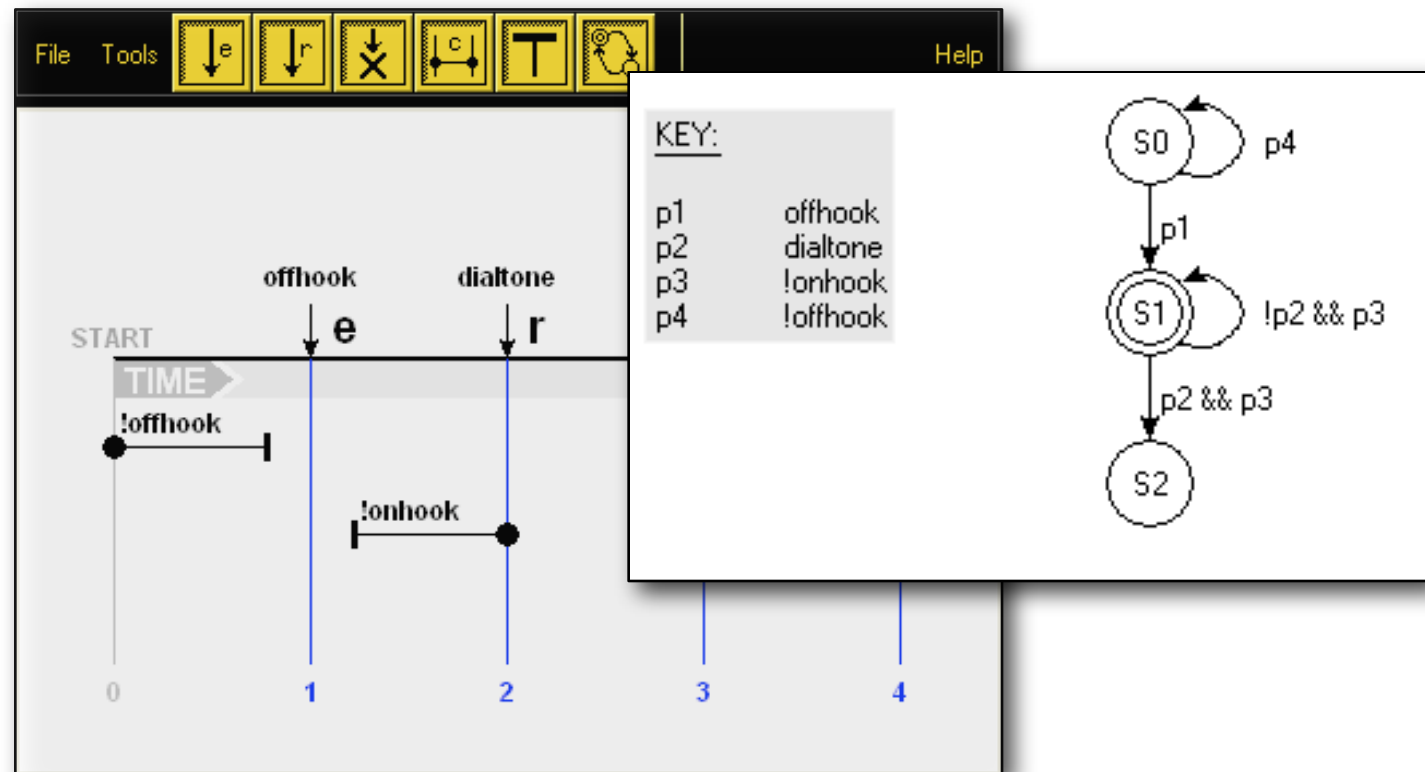
# The Timeline Editor (2)

- ▶ TimeEdit generates never claims:



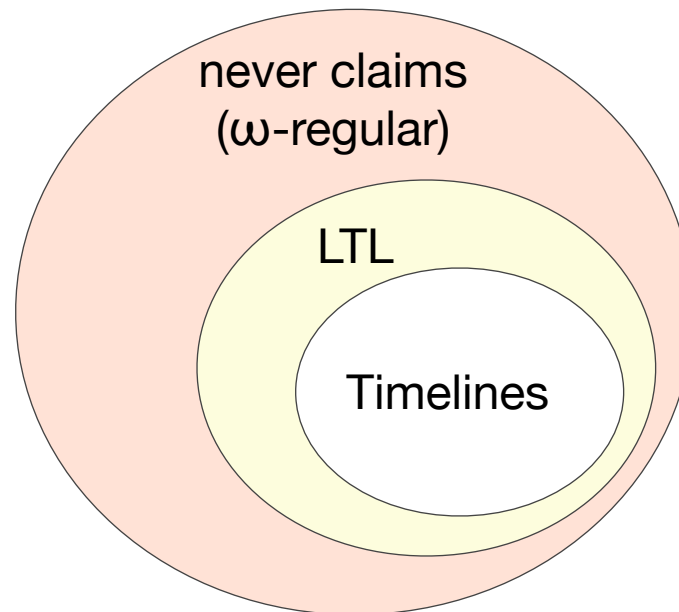
# The Timeline Editor (3)

- ▶ ... and shows the corresponding automaton:



# Timeline Specification

- ▶ Timeline specifications are less expressive than LTL
- ▶ However, it is sometimes easier to describe simple event sequences by timelines.



# Behind the Scenes

- ▶ How does SPIN check correctness properties that are specified by LTL formulae or never claims?
- ▶ Promela models describe processes, which are communicating finite state machines
- ▶ Processes can be described by finite automata. The product of the process automata gives the **state space**.
- ▶ Never claims are processes as well. An accepting run of the never claim states a violation of the claim.

# Acceptance

- ▶ With the standard notion of acceptance we cannot express ongoing, potentially infinite executions.
- ▶ **Standard acceptance**  
An accepting run of a finite state automaton is a finite transition sequence leading to an accepting end state
- ▶ Here we deal with infinite transition sequences, called  **$\omega$ -runs**.

[Holzmann 2003]



# Büchi Acceptance

- ▶ **Büchi acceptance** (Omega acceptance)  
An accepting  $\omega$ -run of a finite state automaton is any infinite run containing an accepting state.
- ▶ Büchi automata accept input sequences that are defined over infinite runs: A Büchi automaton accepts *if and only if* an accepting state is visited infinitely often.
- ▶ How to accept “normal” end states?  
**Stutter extension:** Each end state is extended by a predefined null-transition as a self-loop.

[Holzmann 2003]

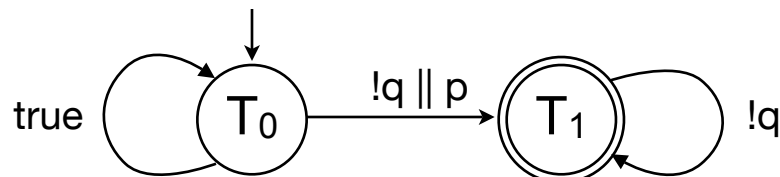
# LTL and Automata

- ▶ LTL has a direct connection to Büchi automata:  
It can be shown that for every LTL formula there exists a Büchi automaton that accepts exactly the runs specified by the formula.
- ▶ SPIN translates LTL formulae into never claims, which represent Büchi automata. The verifier then checks whether the Büchi automaton matches a run of the system (i.e. a path in the reachability graph)

# Example 1

- ▶ The LTL formula  $\Box(p \rightarrow \langle \rangle q)$  with the corresponding never claim (negated!) and the Büchi automaton

```
never { /*  $\Box(p \rightarrow \langle \rangle q)$  */
T0_init:
  if
  :: (! ((q)) && (p)) -> goto accept_S4
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! ((q))) -> goto accept_S4
  fi;
}
```



# Example 2

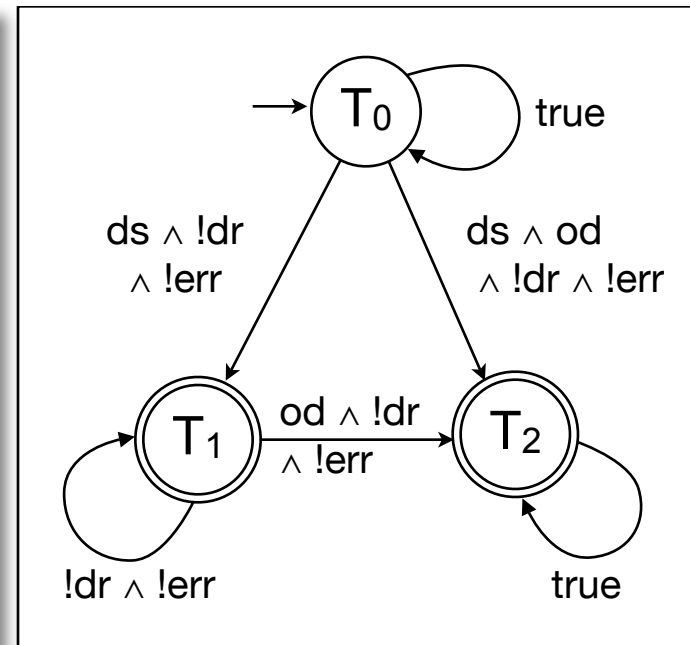
## ▶ Correctness of ABP:

LTL formula, Never claim, and Büchi Automaton

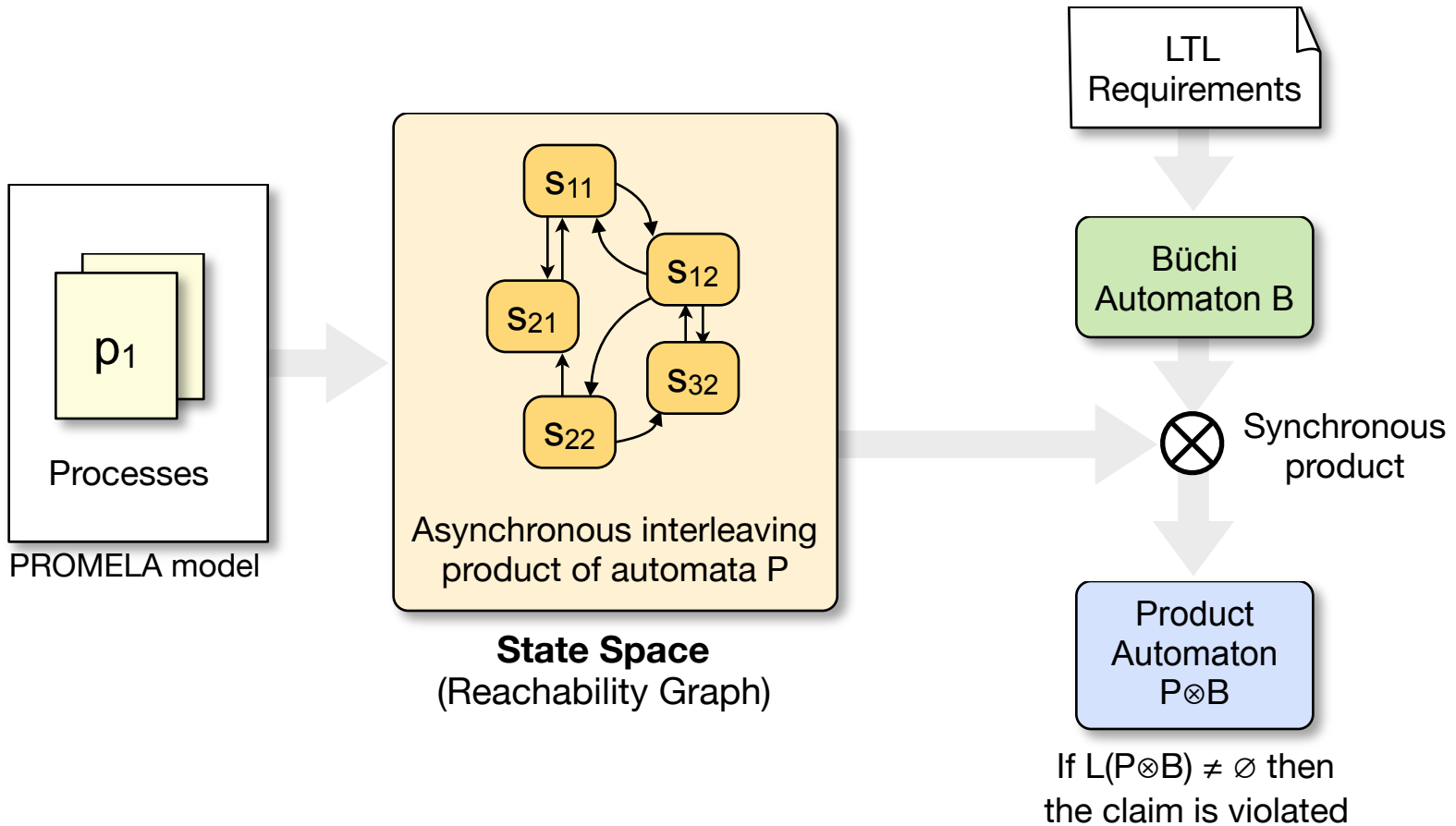
```

never { /* !([] (ds -> (!od) U (dr || err))) */
T0_init:
  if
  :: (! (dr) && ! (err) && (ds)) -> goto accept_S4
  :: (! (dr) && ! (err) && (ds) && (od)) -> goto accept_all
  :: (1) -> goto T0_init
  fi;
accept_S4:
  if
  :: (! (dr) && ! (err)) -> goto accept_S4
  :: (! (dr) && ! (err) && (od)) -> goto accept_all
  fi;
accept_all:
  skip
}

```



# How SPIN checks Never Claims



[G.J. Holzmann: "The Model Checker SPIN", IEEE Transactions on Software Engineering, 23(5), 1997]

# Automata Products

- ▶ A product automaton consists of the Cartesian product of the state sets of the involved automata and transitions
- ▶ **Asynchronous Product**
  - All possible interleavings of the processes of a system are described by an asynchronous product.
- ▶ **Synchronous Product**
  - Synchronous executions (processes and never claims) are represented by a synchronous product.

# Example Model

- ▶ Two processes using the “Half Or Triple Plus One” Rule.

```
#define N 4

int x = N;

active proctype Odd()
{
  do
    :: (x%2) -> x = 3*x+1;
  od;
}

active proctype Even()
{
  do
    :: !(x%2) -> x = x/2;
  od;
}
```

[Holzmann 2003]

## Side note:

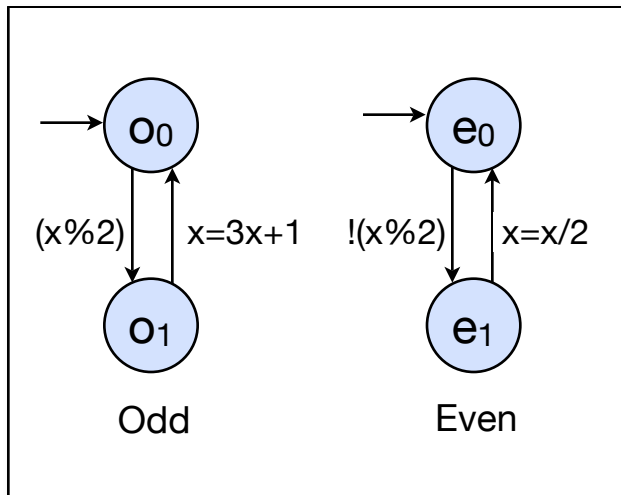
Collatz conjecture states that for all  $N \geq 1$  the sequences converge to 1.

The processes produce so-called hailstone sequences.

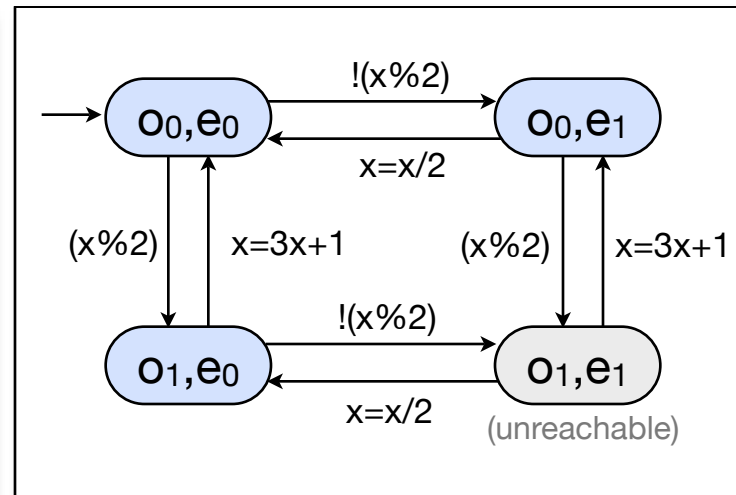
N	$x_1, x_2, \dots$
1	1
2	2, 1
3	3, 10, 5, 16, 8, 4, 2, 1
4	4, 2, 1
5	5, 16, 8, 4, 2, 1

# The State Space (1)

- ▶ The state space (reachability graph) for the HOTPO model, obtained from the **asynchronous product** of the process automata



Automata for *Even* and *Odd*



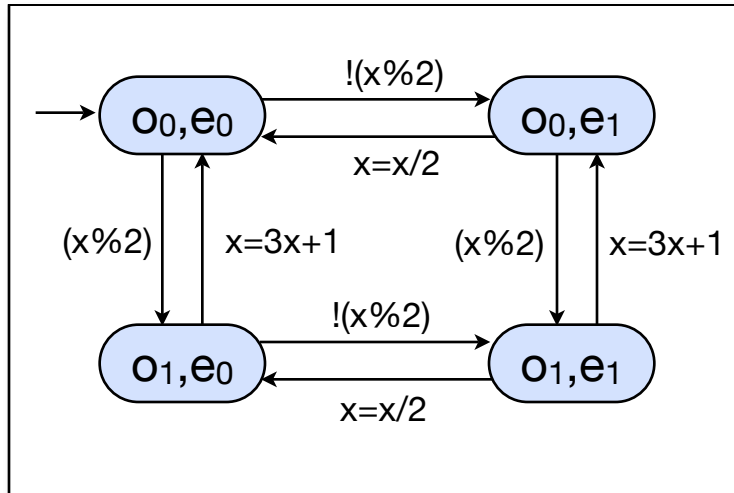
Asynchronous product of the automata

[Holzmann 2003]

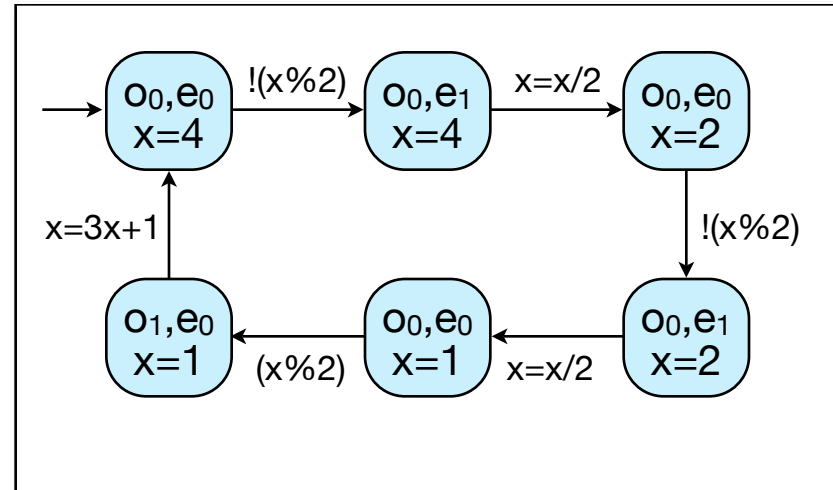


# The State Space (2)

- ▶ Expanding the asynchronous product for  $N=4$



Asynchronous product of the automata



Expanded asynchronous product

[Holzmann 2003]

# Asynchronous product

- ▶ An **asynchronous product** of finite state automata  $A_1..A_n$  is a finite state automaton  $A = (Q, q_0, L, T, F)$ , with
  - $Q = Q_1 \times \dots \times Q_n$ , the Cartesian product of the state sets
  - $q_0 = (q_0^1, \dots, q_0^n)$ , the tuple holding all start states
  - $L = L_1 \cup \dots \cup L_n$ , the union of all label sets (accept-state, end-state, and progress labels).
  - $T =$  set of transitions  $t = ((p_1, \dots, p_n), l, (q_1, \dots, q_n))$  where there is exactly one automaton  $A_i$  having  $(p_i, l, q_i)$  as a transition labeled with  $l$  ( $\forall j \neq i: p_j = q_j$ ).
  - $F =$  set of states  $q = (q_1, \dots, q_n)$  where at least one of the automata states  $q_1, \dots, q_n$  is a final state.

[Holzmann 2003]

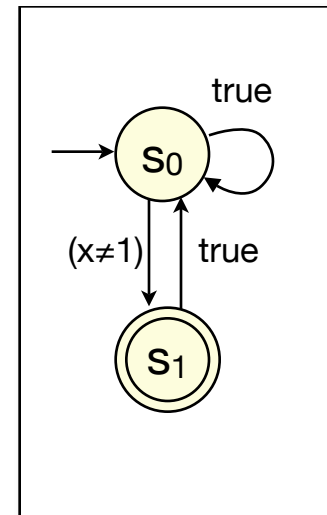
# Checking correctness (1)

- ▶ First, we define a never claim stating that x eventually becomes 1 (This is not true, as the sequence **1,4,2,1,4,2,...** will repeat infinitely often).

```
#define p (x==1)

never {      /* !<>[]p */
T0_init:
  if
  :: (!p) -> goto accept_S1
  :: true -> goto T0_init
  fi;
accept_S1:
  if
  :: true -> goto T0_init
  fi;
}
```

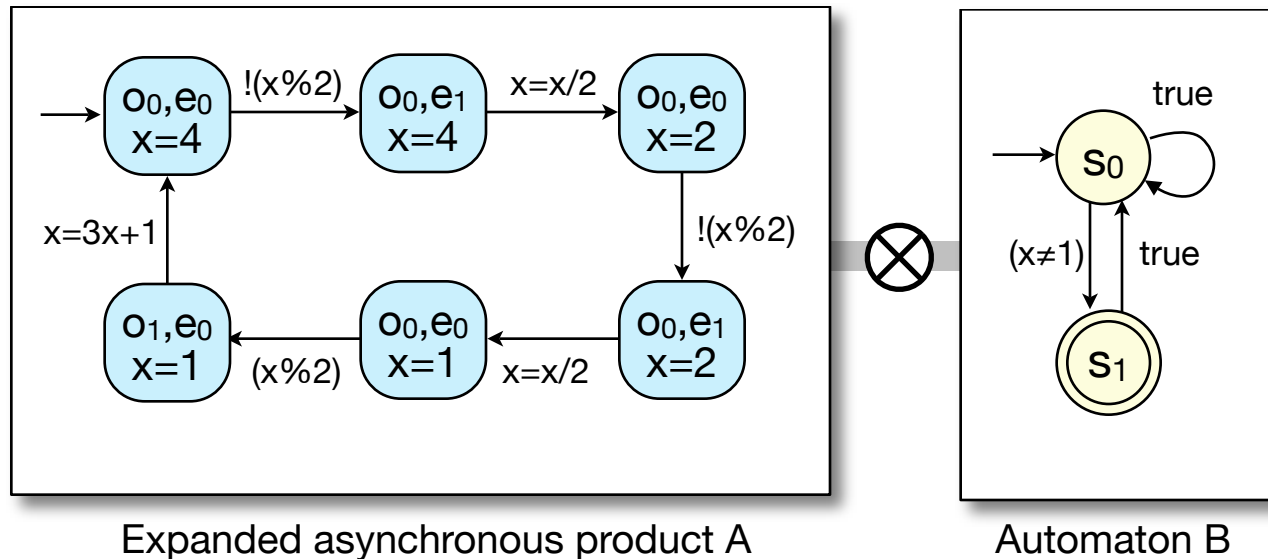
Never claim



Automaton B

# Checking correctness (2)

- ▶ Correctness of a never claim is checked by computing the **synchronous product** of the state space automaton and the claim automaton



[Holzmann 2003]

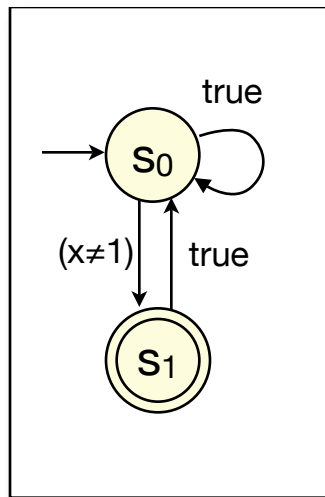
# Synchronous product

- ▶ A **synchronous product** of finite state automata  $P$  and  $B$  is a finite state automaton  $A = (Q, q_0, L, T, F)$ , with
  - $Q = Q_{P'} \times Q_B$ , the Cartesian product of the state sets, where  $P'$  is the stutter-closure of  $P$  having empty self-loops attached to every state without successor.
  - $q_0 = (q_0^{P'}, q_0^B)$ , the tuple holding both start states
  - $L = L_{P'} \times L_B$ , the product of both label sets.
  - $T =$  set of transitions  $t = (t_{P'}, t_B)$  where  $t_{P'} \in T_P$ ,  $t_B \in T_B$
  - $F =$  set of states  $q = (q_{P'}, q_B)$  where  $q_{P'}$  or  $q_B$  is a final state.

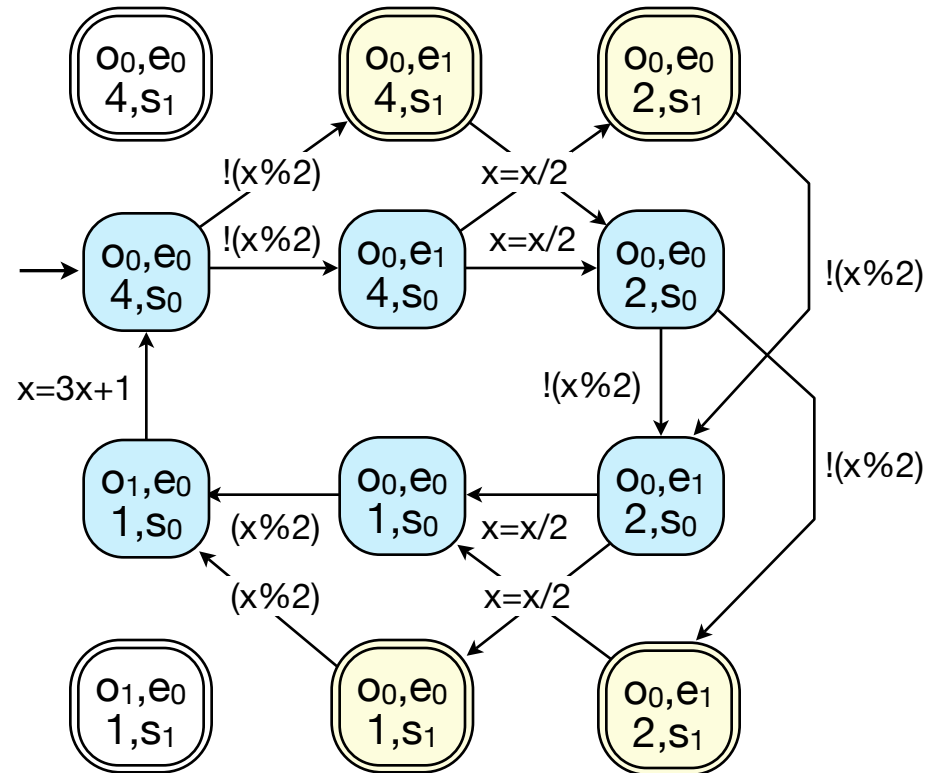
[Holzmann 2003]

# Checking correctness (3)

The synchronous product reflects the synchronous execution of automaton A with the claim automaton B



Automaton B



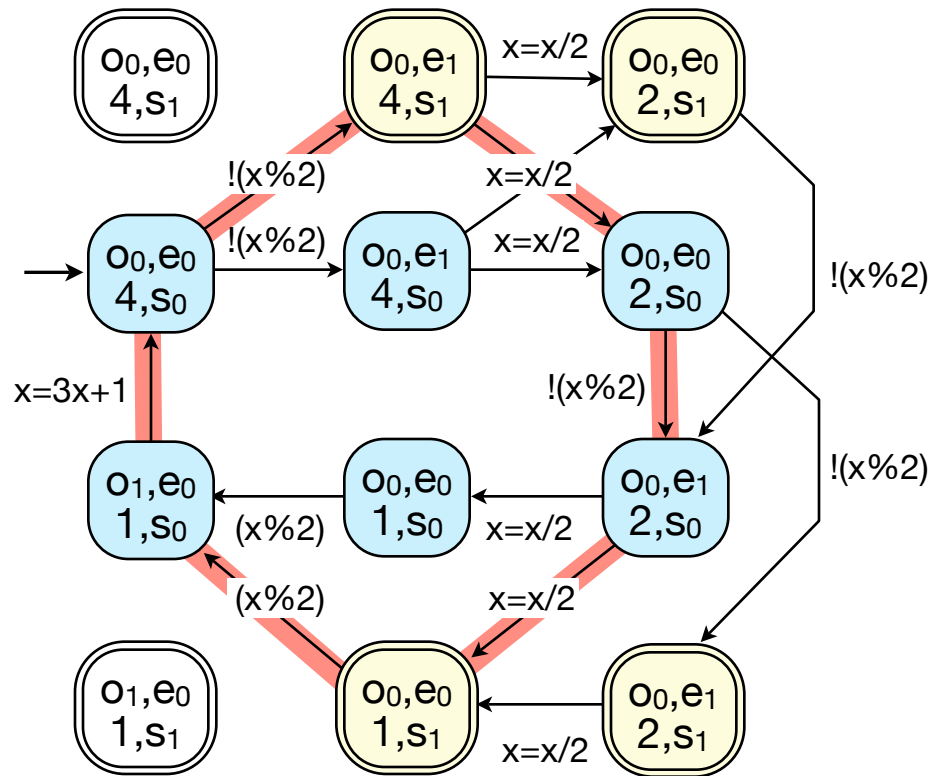
Synchronous product of A and B

# Checking correctness (4)

There is an acceptance cycle, i.e. an infinite execution sequence visiting an accept state.

Visiting such a state where  $!p$  holds implies that the claim is violated.

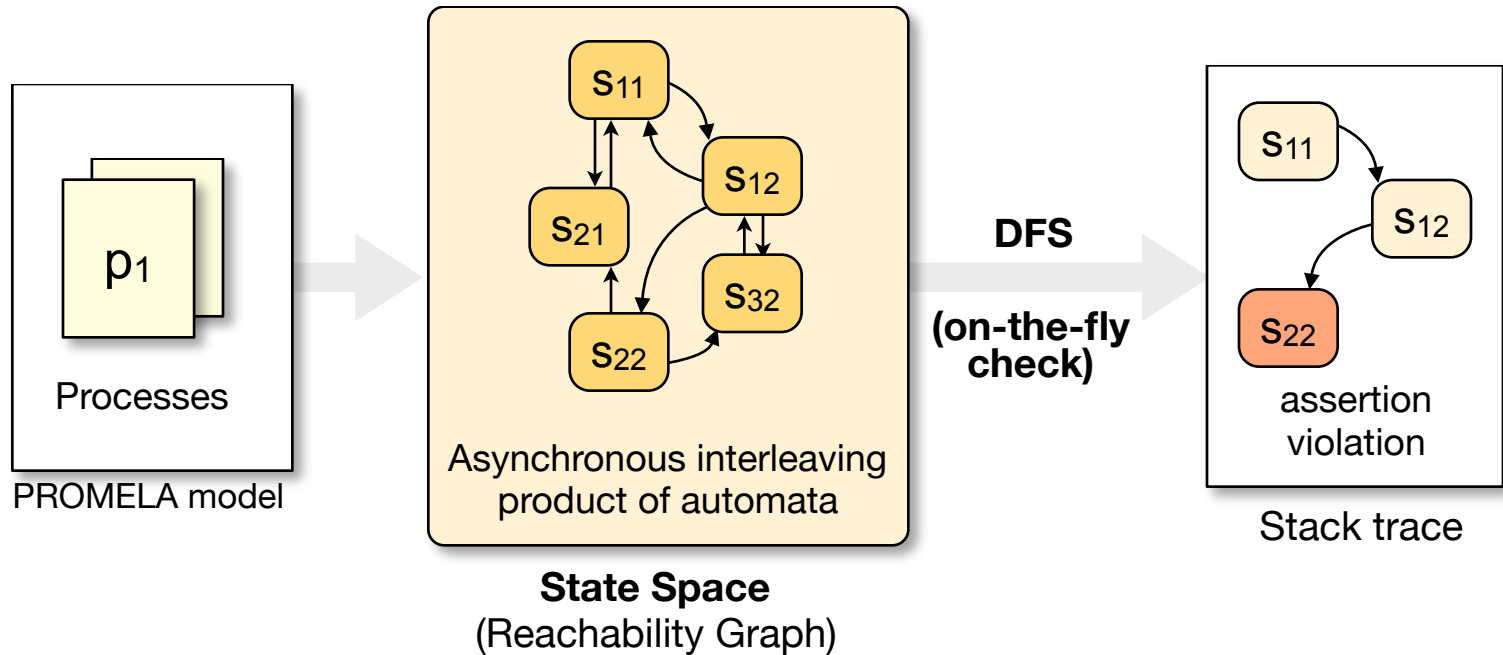
Acceptance cycles are counter-examples to a given claim.



Synchronous product of A and B

# State Space Search (1)

Checking Safety Properties:



[G.J. Holzmann: "The Model Checker SPIN", IEEE Transactions on Software Engineering, 23(5), 1997]



# State Space Search (2)

- ▶ SPIN checks safety properties (assertions, deadlocks) while the state space is constructed (on the fly).
- ▶ The check can be done by a standard DFS

```
Start() {
    Statespace.add(s0)
    Stack.push(s0)
    Search()
}

Search() {
    s = Stack.top()
    if !Safety(s) printStack()
    foreach successor t of s do
        if t not in Statespace then
            Statespace.add(t)
            Stack.push(t)
            Search()
        fi
    od
    Stack.pop()
}
```

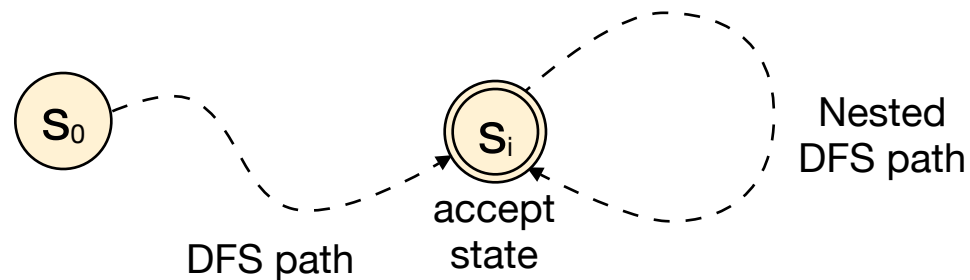
[Holzmann 2003]

# State Space Search (3)

- ▶ Liveness properties are connected to infinite runs and cyclic behaviour. Cycles in the state space can be found by a depth-first search.
- ▶ If an acceptance state is found and all successors of this state have been explored, SPIN starts a **Nested DFS** in order to check whether it can be reached from itself.
- ▶ The algorithm terminates after finding an acceptance cycle or after the complete state space has been explored.

# State Space Search (4)

- ▶ **Nested DFS for checking liveness properties:**  
The first DFS checks whether an accept state is reachable. The second (nested) DFS checks, whether this state is part of a cycle.



# State Space Search (5)

- ▶ Cycles can be detected by Tarjan's DFS algorithm, which finds strongly connected components in linear time. It assigns index numbers and so-called lowlink numbers to nodes of the graph. (Lowlink numbers are the minimum index in the connected component)
- ▶ SPIN uses a **Nested DFS** instead of this algorithm, because the numbers to be stored require a huge amount of memory as the state space might become very large (billions of nodes).
- ▶ The Nested DFS requires storing each state only once and uses 2 bits overhead per state.
- ▶ It cannot detect all cycles, but *at least one* cycle (if existing)

# Positive and Negative Claims

- ▶ Why does SPIN use negative claims (never claims)?
- ▶ **Positive claim:** Prove that the language of the system automaton is included in the language of the claim automaton. Drawback: The state space for language inclusion has at most the size of the Cartesian product.
- ▶ **Negative claim:** Prove that the language of the automata intersection is empty. Advantage: Smaller state space (zero) in the best case.

[G.J. Holzmann: "The Model Checker SPIN", IEEE Transactions on Software Engineering, 23(5), 1997]

# Efficiency of checking

- ▶ Efficiency for checking properties  
(most efficiently first)
  1. Assertions and end state labels
  2. Progress state labels (search for non-progress cycles)
  3. Accept-state labels (search for accept cycles)
  4. Temporal claims

[Holzmann 1993]

# Some Recipes

- ▶ **Abstraction.** You are constructing a validation model and not an implementation. Try to make this model abstract.
- ▶ **Redundancy.** Remove redundant computations and redundant variables (counters, “book-keeping” variables). Everything that is not directly related to the property you are trying to prove should be avoided.
- ▶ **Channels.** Reduce the capacity of asynchronous channels to a minimum (2 or 3). Use synchronous channels where possible.

[T.C. Ruys: “SPIN Tutorial: How to become a SPIN Doctor”]

# Some more Recipes

- ▶ Make variables local if possible.
- ▶ Local computations should be merged into atomic or `d_step` blocks.
- ▶ Non-deterministic random choices should be modeled using an if-clause (having guard statements that are executable at the same time).
- ▶ Lossy channels are modeled best by letting the sending process lose messages or by a process that “steals” messages.

[T.C. Ruys: “SPIN Tutorial: How to become a SPIN Doctor”]



# Lessons learned

- ▶ SPIN does not directly prove correctness. It tries to find counterexamples to the specified correctness claims.
- ▶ Liveness properties are expressed by never claims or LTL formulae. They require the largest computation overhead for verification.
- ▶ Remember to keep the models abstract and simple!