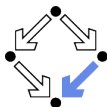
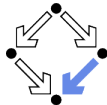


# Model-Checking Concurrent Systems

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria  
<http://www.risc.jku.at>





---

## 1. The Model Checker Spin

## 2. Automatic Model Checking

# The Model Checker Spin

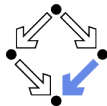


- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Symposium series since 1995 (25th symposium “Spin 2018”).
  - ACM System Software Award in 2001.
- Spin resources:
  - Web site: <http://spinroot.com>.
  - Survey paper: Holzmann “The Model Checker Spin”, 1997.
  - Book: Holzmann “The Spin Model Checker — Primer and Reference Manual”, 2004.

Goal: verification of (concurrent/distributed) software models.



# The Model Checker Spin



## On-the-fly LTL model checking of finite state systems.

- System  $S$  modeled by automaton  $S_A$ .
  - Explicit representation of automaton states.
  - There exist various other approaches (discussed later).
- On-the-fly model checking.
  - Reachable states of  $S_A$  are only expended on demand.
  - *Partial order reduction* to keep state space manageable.
- LTL model checking.
  - Property  $P$  to be checked described in PLTL.
    - Propositional linear temporal logic.
  - Description converted into property automaton  $P_A$ .
    - Automaton accepts only system runs that do not satisfy the property.

## Model checking based on automata theory.

# The Spin System Architecture

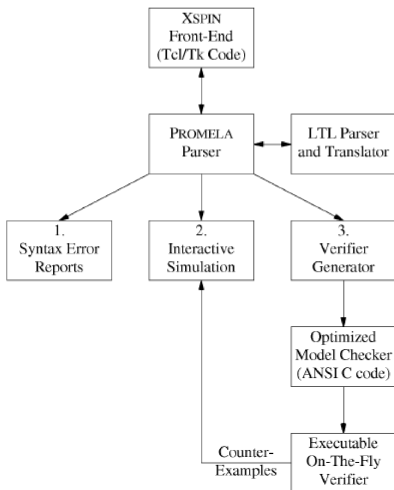
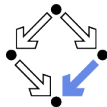
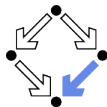


Fig. 1. The structure of SPIN simulation and verification.

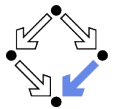
# Features of Spin



- System description in Promela.
  - Promela = Process Meta-Language.
    - Spin = Simple Promela Interpreter.
  - Express coordination and synchronization aspects of a real system.
  - Actual computation can be e.g. handled by embedded C code.
- **Simulation mode.**
  - Investigate individual system behaviors.
  - Inspect system state.
  - Graphical interface `ispin` for visualization.
- **Verification mode.**
  - Verify properties shared by all possible system behaviors.
  - Properties specified in PLTL and translated to “never claims”.
    - Promela description of automaton for negation of the property.
  - Generated counter examples may be investigated in simulation mode.

**Verification and simulation are tightly integrated in Spin.**

# The Spin User Interface



Spin Version 6.1.0 -- 4 May 2011 :: Bpln Version 1.0.5 -- 27 February 2011

File Edit View Simulate / Replay Verification Swarm Run <Help> Save Session Restore Session <Quit>

Open... ReOpen Save Save As... Syntax Check Redundancy Check Symbol Table Find

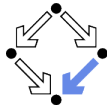
```
17 prototype server()
18 {
19   unsigned given : 2 = 0;
20   unsigned waiting : 2 = 0;
21   unsigned sender : 2;
22   do :: true ->
23     R: if
24       :: request[0] ? MESSAGE ->
25       S1: sender = 1
26       :: request[1] ? MESSAGE ->
27       S2: sender = 2
28     fi;
29
30     if
31       :: sender == given ->
32       if
33         :: waiting == 0 ->
34         given = 0
35       :: else ->
36         given = waiting;
37         waiting = 0;
38         answer[given-1] | MESSAGE;
39       fi;
40       :: given == 0 ->
41         given = sender;
42         answer[given-1] | MESSAGE
43       :: else
44         waiting = sender
45       fi;
46     od;
47   }
48 }
```

Automata View zoom in zoom out

Select: p\_client p\_server main\_mutex

Spin Version 6.1.0 -- 4 May 2011  
iSpin Version 1.0.5 -- 27 February 2011  
TclTk Version 8.4.6.4  
1 clientServer.txt:1  
2 verification  
3 simulate/replay  
4 verification

# The Client/Server System in Promela



```
/* definition of a constant MESSAGE */
mtype = { MESSAGE };

/* two arrays of channels of size 2,
   each channel has a buffer size 1 */
chan request[2] = [1] of { mtype };
chan answer [2] = [1] of { mtype };

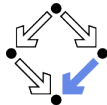
/* the system of three processes */
init
{
  run client(1);
  run client(2);
  run server();
}

/* the mutual exclusion property */
ltl mutex { []!(client[1]@C && client[2]@C)}
```

```
/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;
    W: answer[id-1] ? MESSAGE;
    C: skip; // the critical region
    request[id-1] ! MESSAGE
  od;
}
```



# The Client/Server System in Promela



```
/* the server process type */
proctype server()
{
  /* three variables of two bit each */
  unsigned given   : 2 = 0;
  unsigned waiting : 2 = 0;
  unsigned sender  : 2;

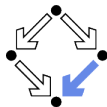
  do :: true ->

    /* receiving the message */
    R: if
    :: request[0] ? MESSAGE ->
      S1: sender = 1
    :: request[1] ? MESSAGE ->
      S2: sender = 2
    fi;

    /* answering the message */
    if
    :: sender == given ->
      if
      :: waiting == 0 ->
        given = 0
      :: else ->
        given = waiting;
        waiting = 0;
        answer[given-1] ! MESSAGE
      fi;
    :: given == 0 ->
      given = sender;
      answer[given-1] ! MESSAGE
    :: else
      waiting = sender
    fi;

  od;
}
```

# The Spin Simulation View



Spin Version 6.1.0 - 4 May 2011 :: Spin Version 1.0.5 - 27 February 2011

Mode: Random, with seed: 123  
Interactive (for resolution of all nondeterminism)  
Guided, with tool: clientServer.txt  
Initial steps skipped: 0  
Maximum number of steps: 10000  
Track Data Values (this can be slow)

A Full Channel  
blocks new messages  
loses new messages  
MSC max text width: 20  
MSC update delay: 25

Output Filtering (reg. exp.s.)  
process ids:  
queue ids:  
var names:  
tracked variable:  
track scaling:

(Re)Run  
Stop  
Restart  
Step Forward  
Step Backward

Background command executed:  
spin -p -s -r -X -v -n123 -l -g -u10000 clientServer.txt

Save in: msc.ps

```
1 mtype = { MESSAGE };
2
3 chan request[2] = [1] of { mtype };
4 chan answer[2] = [1] of { mtype };
5
6
7 prototype client(byte id)
8 {
9   do :: true ->
10    request[id-1] ! MESSAGE;
11    W: answer[id-1] ? MESSAGE;
12    C: skip;
13    request[id-1] ! MESSAGE;
14  od;
15 }
16
17 prototype server()
18 {
19
```

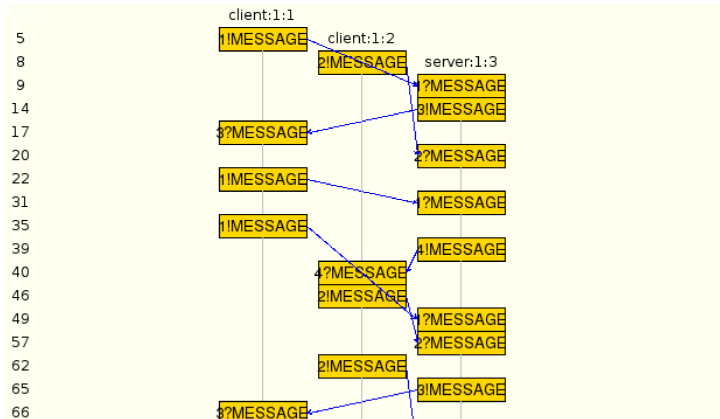
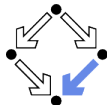
```
5 client 1 -->> client 2: !MESSAGE
6 client 1 -->> server 3: !MESSAGE
7 client 2 -->> server 3: ?MESSAGE
8 client 1 -->> server 3: ?MESSAGE
9 client 1 -->> client 2: ?MESSAGE
10 client 1 -->> server 3: ?MESSAGE
11 client 1 -->> client 2: ?MESSAGE
12 client 1 -->> server 3: ?MESSAGE
13 client 1 -->> client 2: ?MESSAGE
14 client 1 -->> server 3: ?MESSAGE
15 client 1 -->> client 2: ?MESSAGE
16 client 1 -->> server 3: ?MESSAGE
17 client 1 -->> client 2: ?MESSAGE
18 client 1 -->> server 3: ?MESSAGE
19 client 1 -->> client 2: ?MESSAGE
20 client 1 -->> server 3: ?MESSAGE
21 client 1 -->> client 2: ?MESSAGE
22 client 1 -->> server 3: ?MESSAGE
23 client 1 -->> client 2: ?MESSAGE
24 client 1 -->> server 3: ?MESSAGE
25 client 1 -->> client 2: ?MESSAGE
26 client 1 -->> server 3: ?MESSAGE
27 client 1 -->> client 2: ?MESSAGE
28 client 1 -->> server 3: ?MESSAGE
29 client 1 -->> client 2: ?MESSAGE
30 client 1 -->> server 3: ?MESSAGE
31 client 1 -->> client 2: ?MESSAGE
32 client 1 -->> server 3: ?MESSAGE
33 client 1 -->> client 2: ?MESSAGE
34 client 1 -->> server 3: ?MESSAGE
35 client 1 -->> client 2: ?MESSAGE
36 client 1 -->> server 3: ?MESSAGE
37 client 1 -->> client 2: ?MESSAGE
38 client 1 -->> server 3: ?MESSAGE
39 client 1 -->> client 2: ?MESSAGE
40 client 1 -->> server 3: ?MESSAGE
41 client 1 -->> client 2: ?MESSAGE
42 client 1 -->> server 3: ?MESSAGE
43 client 1 -->> client 2: ?MESSAGE
44 client 1 -->> server 3: ?MESSAGE
45 client 1 -->> client 2: ?MESSAGE
46 client 1 -->> server 3: ?MESSAGE
47 client 1 -->> client 2: ?MESSAGE
48 client 1 -->> server 3: ?MESSAGE
49 client 1 -->> client 2: ?MESSAGE
50 client 1 -->> server 3: ?MESSAGE
51 client 1 -->> client 2: ?MESSAGE
52 client 1 -->> server 3: ?MESSAGE
53 client 1 -->> client 2: ?MESSAGE
54 client 1 -->> server 3: ?MESSAGE
55 client 1 -->> client 2: ?MESSAGE
56 client 1 -->> server 3: ?MESSAGE
57 client 1 -->> client 2: ?MESSAGE
58 client 1 -->> server 3: ?MESSAGE
59 client 1 -->> client 2: ?MESSAGE
60 client 1 -->> server 3: ?MESSAGE
61 client 1 -->> client 2: ?MESSAGE
62 client 1 -->> server 3: ?MESSAGE
63 client 1 -->> client 2: ?MESSAGE
64 client 1 -->> server 3: ?MESSAGE
65 client 1 -->> client 2: ?MESSAGE
66 client 1 -->> server 3: ?MESSAGE
67 client 1 -->> client 2: ?MESSAGE
68 client 1 -->> server 3: ?MESSAGE
69 client 1 -->> client 2: ?MESSAGE
70 client 1 -->> server 3: ?MESSAGE
71 client 1 -->> client 2: ?MESSAGE
72 client 1 -->> server 3: ?MESSAGE
73 client 1 -->> client 2: ?MESSAGE
74 client 1 -->> server 3: ?MESSAGE
75 client 1 -->> client 2: ?MESSAGE
76 client 1 -->> server 3: ?MESSAGE
77 client 1 -->> client 2: ?MESSAGE
78 client 1 -->> server 3: ?MESSAGE
79 client 1 -->> client 2: ?MESSAGE
80 client 1 -->> server 3: ?MESSAGE
81 client 1 -->> client 2: ?MESSAGE
82 client 1 -->> server 3: ?MESSAGE
83 client 1 -->> client 2: ?MESSAGE
84 client 1 -->> server 3: ?MESSAGE
85 client 1 -->> client 2: ?MESSAGE
86 client 1 -->> server 3: ?MESSAGE
87 client 1 -->> client 2: ?MESSAGE
88 client 1 -->> server 3: ?MESSAGE
89 client 1 -->> client 2: ?MESSAGE
90 client 1 -->> server 3: ?MESSAGE
91 client 1 -->> client 2: ?MESSAGE
92 client 1 -->> server 3: ?MESSAGE
93 client 1 -->> client 2: ?MESSAGE
94 client 1 -->> server 3: ?MESSAGE
95 client 1 -->> client 2: ?MESSAGE
96 client 1 -->> server 3: ?MESSAGE
97 client 1 -->> client 2: ?MESSAGE
98 client 1 -->> server 3: ?MESSAGE
99 client 1 -->> client 2: ?MESSAGE
100 client 1 -->> server 3: ?MESSAGE
```

```
153: proc 3 (server) clientServer.txt: 27 (state 5) [sender = 2]
155: proc 3 (server) clientServer.txt: 30 (state 22) [else]
156: proc 3 (server) clientServer.txt: 44 (state 21) [waiting = sender]
159: proc 3 (server) clientServer.txt: 22 (state 24) [{}]
160: proc 3 (server) clientServer.txt: 23 (state 6) [request[0]?MESSAGE]
161: proc 3 (server) clientServer.txt: 25 (state 3) [sender = 1]
163: proc 3 (server) clientServer.txt: 30 (state 22) [!(sender == given)]
164: proc 3 (server) clientServer.txt: 32 (state 15) [else]
165: proc 3 (server) clientServer.txt: 36 (state 12) [given = waiting]
166: proc 3 (server) clientServer.txt: 37 (state 13) [waiting = 0]
```

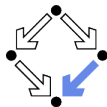
```
[variable values, step 166]
server(3) given = 2
server(3) sender = 3
server(3) waiting = 0

[queues, step 166]
q 1 :: (request[0])
q 2 :: (request[1])
q 3 :: (answer[0])
q 4 :: (answer[1])
```

# Simulating the System Execution in Spin



# The Spin Verification View



Spin Version 6.1.0 - 4 May 2011 - BSpin Version 1.0.5 - 27 February 2011

File/View Simulate / Replay Verification Search Run <help> Save Session Restore Session <Quit>

Safety	Storage Mode	Search Mode	Remove
<input checked="" type="checkbox"/> + invalid endstates (deadlock) <input checked="" type="checkbox"/> + assertion violations <input checked="" type="checkbox"/> + x/ys assertions	<input checked="" type="checkbox"/> exhaustive <input type="checkbox"/> minimized automata (slow) <input type="checkbox"/> outloop compression <input checked="" type="checkbox"/> hash-compact <input type="checkbox"/> bitstate/superspace Never Claims	<input checked="" type="checkbox"/> depth-first search <input checked="" type="checkbox"/> + partial order reduction <input type="checkbox"/> + bounded context switching with bound: 0 <input type="checkbox"/> + iterative search for short trail <input type="checkbox"/> breadth-first search <input checked="" type="checkbox"/> + partial order reduction <input checked="" type="checkbox"/> report unreachable code	Advanced Parameters Physical Memory Available (in Mbytes): 1024 explain Estimated State Space Size (states x 10 <sup>3</sup> ): 1000 explain Maximum Search Depth (telegs): 10000 explain Nr of hash functions in Bitstate mode: 9 explain Size for Minimized Automata: 100 explain Extra Verifier Generation Options: explain Extra Compile-Time Directives: -O2 explain Extra Run-Time Options: explain
<input type="checkbox"/> non-progress cycles <input checked="" type="checkbox"/> acceptance cycles <input type="checkbox"/> enforce weak fairness constraint	<input type="checkbox"/> do not use a never claim or III property <input checked="" type="checkbox"/> use claim claim name (opt):	<input type="checkbox"/> Show Error Trapping Options	

Run Stop

```
1 mtype = { MESSAGE };
2
3
4 chan request[2] = [1] of { mtype };
5 chan answer [2] = [1] of { mtype };
6
7 prototype client(byte id)
8 {
9   do : true ->
10    request[id-1] ! MESSAGE;
11    W: answer[id-1] ? MESSAGE;
12    C: skip;
13    request[id-1] ! MESSAGE
14  }
15
16
17 prototype server()
18 {
19   unsigned given : 2 = 0;
20   unsigned waiting : 2 = 0;
21   unsigned sender : 2;
```

```
spin -a clientServer.txt
it mutex: [1] ! ((client[1]@C) && ((client[2]@C)))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10200 -a
Pid: 16183
warning: for p.o. reduction to valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.1.0 - 4 May 2011)
+ Partial Order Reduction

Full state-space search for:
never claim          + (mutex)
assertion violations + (if within scope of claim)
acceptance cycles   + (fairness disabled)
invalid end states  - (disabled by never claim)

State-vector 48 byte, depth reached 137, errors: 0
157 states, stored
88 states, matched
225 transitions (= stored + matched)
```

# Specifying a System Property in Spin



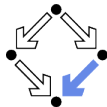
The easiest way to specify an LTL property is to specify it inline. The formula is specified globally (i.e., outside all proctype or init declarations) with the following syntax:

```
ltl [ name ] '{' formula '}'
```

... The names of operators can either be abbreviated with the symbols shown above, or spelled out in full (as always, eventually, until, implies, and equivalent. The alternative operators weakuntil, stronguntil, and release (for the V operator, see above), are also supported. This means that the following two are equivalent:

```
ltl p1 { []<> p }
ltl p2 { always eventually p }
ltl p3 { eventually (a > b) implies len(q) == 0 }
```

There is just one restriction: you cannot use the predefined operators empty, nempty, full, nfull in inline ltl formula...



Grammar:

```
ltl ::= opd | ( ltl ) | ltl binop ltl | unop ltl
```

Operands (opd):

true, false, user-defined names starting with a lower-case letter,  
or embedded expressions inside curly braces, e.g.: { a+b>n }.

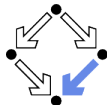
Unary Operators (unop):

```
[] (the temporal operator always)  
<>(the temporal operator eventually)  
! (the boolean operator for negation)
```

Binary Operators (binop):

```
U (the temporal operator strong until)  
W (the temporal operator weak until)  
V (the dual of U): (p V q) means !(p U !q)  
&& (the boolean operator for logical and)  
|| (the boolean operator for logical or)  
\ (alternative form of &&)  
/ (alternative form of ||)  
-> (the boolean operator for logical implication)  
<-> (the boolean operator for logical equivalence)
```

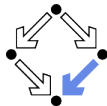
# Spin Atomic Predicates



(a+b > c)  
(len(q) < 5)  
(process@Label)  
(process[pid]@Label)

- PROMELA conditions with references to *global* system variables.
  - C-like boolean expressions.
  - len(*q*): the number of messages in channel *q*.
  - *process@Label*: true if the execution of the process with process type *process* is in the state marked by *Label*.
  - *process[pid]@Label*: true if the execution of the process with type *process* and process identifier *pid* is in the state marked by *Label*.
    - Process *init* receives identifier 0.
    - Processes instantiated with *run* receive identifiers 1, 2, ...

Atomic predicates can describe arbitrary state conditions.



# The Spin Verification Options

---

## ■ Verification mode

- Safety: predefined safety properties.
- Liveness: predefined liveness properties.
- Use Claim: user-defined (LTL) properties.

## ■ Storage mode

- Exhaustive: full error search.
  - May require (too) much memory.
- Hash-compact and bitstate/supertrace: probabilistic search.
  - Less memory consuming but not guaranteed to find all errors.
  - Number of reported hash collisions should not be too high.

## ■ Advanced Parameters

- Physical Memory Size: increase, if checker runs out of memory.
- Maximum Search Depth: reduce to find shorter counterexamples.

By default, select “use claim” and “exhaustive”.



# The Spin Verification Output



(Spin Version 6.1.0 -- 4 May 2011)

+ Partial Order Reduction

Full statespace search for:

never claim + (mutex)  
assertion violations + (if within scope of claim)  
acceptance cycles + (fairness disabled)  
invalid end states - (disabled by never claim)

State-vector 48 byte, depth reached 137, errors: 0

157 states, stored

68 states, matched

225 transitions (= stored+matched)

0 atomic steps

hash conflicts: 0 (resolved)

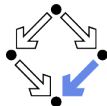
Stats on memory usage (in Megabytes):

...

pan: elapsed time 0 seconds

No errors found -- did you verify all claims?

# More Promela Features



Active processes, inline definitions, atomic statements, output.

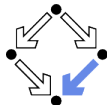
```
mtype = { P, C, N }
mtype turn = P;

inline request(x, y) { atomic { x == y -> x = N } }
inline release(x, y) { atomic { x = y } }
#define FORMAT "Output: %c\n"

active proctype producer()
{
  do
    :: request(turn, P) -> printf(FORMAT, 'P'); release(turn, C);
  od
}

active proctype consumer()
{
  do
    :: request(turn, C) -> printf(FORMAT, 'C'); release(turn, P);
  od
}
```

# More Promela Features

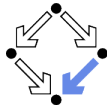


## Embedded C code.

```
/* declaration is added locally to proctype main */
c_state "float f" "Local main"

active proctype main()
{
  c_code { Pmain->f = 0; }
  do
    :: c_expr { Pmain->f <= 300 };
    c_code { Pmain->f = 1.5 * Pmain->f ; };
    c_code { printf("%4.0f\n", Pmain->f); };
  od;
}
```

Can embed computational aspects into a Promela model (only works in verification mode where a C program is generated from the model).

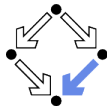


---

## 1. The Model Checker Spin

## 2. Automatic Model Checking

# The Basic Approach

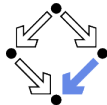


Translation of the original problem to a problem in automata theory.

- **Original problem:**  $S \models P$ .
  - $S = \langle I, R \rangle$ , PLTL formula  $P$ .
  - Does property  $P$  hold for every run of system  $S$ ?
- Construct **system automaton**  $S_A$  with language  $\mathcal{L}(S_A)$ .
  - A **language** is a set of infinite words.
  - Each such word describes a system run.
  - $\mathcal{L}(S_A)$  describes the set of runs of  $S$ .
- Construct **property automaton**  $P_A$  with language  $\mathcal{L}(P_A)$ .
  - $\mathcal{L}(P_A)$  describes the set of runs satisfying  $P$ .
- **Equivalent Problem:**  $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$ .
  - The language of  $S_A$  must be contained in the language of  $P_A$ .

There exists an efficient algorithm to solve this problem.

# Finite State Automata

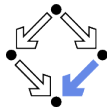


A (variant of a) labeled transition system in a finite state space.

- Take finite sets *State* and *Label*.
  - The **state space** *State*.
  - The **alphabet** *Label*.
- A **(finite state) automaton**  $A = \langle I, R, F \rangle$  over *State* and *Label*:
  - A set of **initial states**  $I \subseteq \text{State}$ .
  - A **labeled transition relation**  $R \subseteq \text{Label} \times \text{State} \times \text{State}$ .
  - A set of **final states**  $F \subseteq \text{State}$ .
    - **Büchi automata**:  $F$  is called the set of **accepting states**.

We will only consider infinite runs of Büchi automata.

# Runs and Languages



- An **infinite run**  $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \xrightarrow{l_2} \dots$  of automaton  $A$ :
  - $s_0 \in I$  and  $R(l_i, s_i, s_{i+1})$  for all  $i \in \mathbb{N}$ .
  - Run  $r$  is said to **read** the infinite word  $w(r) := \langle l_0, l_1, l_2, \dots \rangle$ .
- $A = \langle I, R, F \rangle$  **accepts** an infinite run  $r$ :
  - Some state  $s \in F$  occurs infinitely often in  $r$ .
  - This notion of acceptance is also called **Büchi acceptance**.
- The **language**  $\mathcal{L}(A)$  of automaton  $A$ :
  - $\mathcal{L}(A) := \{w(r) : A \text{ accepts } r\}$ .
  - The set of words which are read by the runs accepted by  $A$ .
- **Example:**  $\mathcal{L}(A) = (a^*bb^*a)^*a^\omega + (a^*bb^*a)^\omega = (b^*a)^\omega$ .
  - $w^i = ww \dots w$  ( $i$  occurrences of  $w$ ).
  - $w^* = \{w^i : i \in \mathbb{N}\} = \{\langle \rangle, w, ww, www, \dots\}$ .
  - $w^\omega = wwww \dots$  (infinitely often).
  - An infinite repetition of an arbitrary number of  $b$  followed by  $a$ .

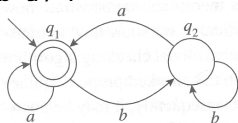
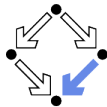


Figure 9.1  
A finite automaton.

# A Finite State System as an Automaton

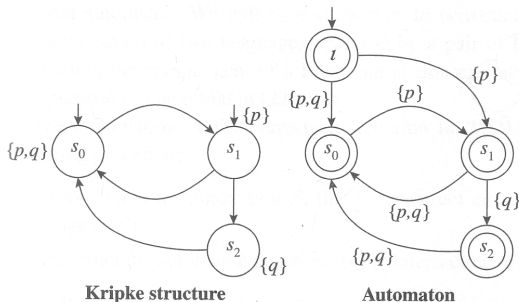


The **automaton**  $S_A = \langle I, R, F \rangle$  for a finite state system  $S = \langle I_S, R_S \rangle$ :

- $State := State_S \cup \{\iota\}$ .
  - The state space  $State_S$  of  $S$  is finite; additional state  $\iota$  ("iota").
- $Label := \mathbb{P}(AP)$ .
  - Finite set  $AP$  of **atomic propositions**.  
All PLTL formulas are built from this set only.
  - Powerset  $\mathbb{P}(S) := \{s : s \subseteq S\}$ .
  - Every element of  $Label$  is thus a set of atomic propositions.
- $I := \{\iota\}$ .
  - Single initial state  $\iota$ .
- $R(I, s, s') := \Leftrightarrow I = L(s') \wedge (R_S(s, s') \vee (s = \iota \wedge I_S(s')))$ .
  - $L(s) := \{p \in AP : s \models p\}$ .
  - Each transition is labeled by the set of atomic propositions satisfied by the **successor state**.
- $F := State$ .
  - Every state is accepting.



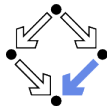
# A Finite State System as an Automaton



**Figure 9.2**  
Transforming a Kripke structure into an automaton.

Edmund Clarke et al: "Model Checking", 1999.

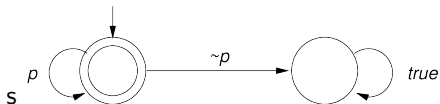
If  $r = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$  is a run of  $S$ , then  $S_A$  accepts the labelled version  $r_l := l \xrightarrow{L(s_0)} s_0 \xrightarrow{L(s_1)} s_1 \xrightarrow{L(s_2)} s_2 \xrightarrow{L(s_3)} \dots$  of  $r$ .



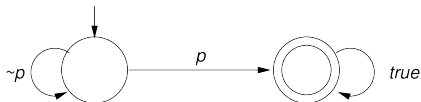
# A System Property as an Automaton

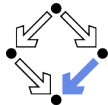
Also an PLTL formula can be translated to a finite state automaton.

- We need the **automaton**  $P_A$  for a PLTL property  $P$ .
  - Requirement:  $r \models P \Leftrightarrow P_A$  accepts  $r_I$ .
  - A run satisfies property  $P$  if and only if automaton  $A_P$  accepts the labeled version of the run.
- **Example:**  $\Box p$ .



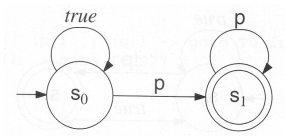
- **Example:**  $\Diamond p$ .





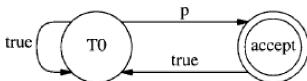
# Further Examples

- Example:  $\diamond \square p$ .



Gerard Holzmann: "The Spin Model Checker", 2004.

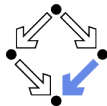
- Example:  $\square \diamond p$ .



Gerard Holzmann: "The Model Checker Spin", 1997.

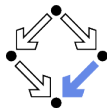
Arbitrary PLTL formulas can be converted to automata (we omit the details).

# System Properties



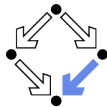
- **State equivalence:**  $L(s) = L(t)$ .
  - Both states have the same labels.
  - Both states satisfy the same atomic propositions in  $AP$ .
- **Run equivalence:**  $w(r_l) = w(r'_l)$ .
  - Both runs have the same sequences of labels.
  - Both runs satisfy the same PLTL formulas built over  $AP$ .
- **Indistinguishability:**  $w(r_l) = w(r'_l) \Rightarrow (r \models P \Leftrightarrow r' \models P)$ 
  - PLTL formula  $P$  cannot distinguish between runs  $r$  and  $r'$  whose labeled versions read the same words.
- **Consequence:**  $S \models P \Leftrightarrow \mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$ .
  - If every run of  $S$  satisfies  $P$ , then every word  $w(r_l)$  in  $\mathcal{L}(S_A)$  equals some word  $w(r'_l)$  in  $\mathcal{L}(P_A)$ , and vice versa.

# The Next Steps



- **Problem:**  $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$ 
  - Equivalent to:  $\mathcal{L}(S_A) \cap \overline{\mathcal{L}(P_A)} = \emptyset$ .
    - Complement  $\bar{L} := \{w : w \notin L\}$ .
  - Equivalent to:  $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$ .
    - $\overline{\mathcal{L}(P_A)} = \mathcal{L}((\neg P)_A)$ .
- **Equivalent Problem:**  $\mathcal{L}(S_A) \cap \mathcal{L}((\neg P)_A) = \emptyset$ .
  - We will introduce the **synchronized product automaton**  $A \otimes B$ .
    - A transition of  $A \otimes B$  represents a simultaneous transition of  $A$  and  $B$ .
  - Property:  $\mathcal{L}(A) \cap \mathcal{L}(B) = \mathcal{L}(A \otimes B)$ .
- **Final Problem:**  $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$ .
  - We have to check whether the language of this automaton is empty.
  - We have to look for a word  $w$  accepted by this automaton.
    - If no such  $w$  exists, then  $S \models P$ .
    - If such a  $w = w(r)$  exists, then  $r$  is a **counterexample**, i.e. a run of  $S$  such that  $r \not\models P$ .

# Synchronized Product of Two Automata

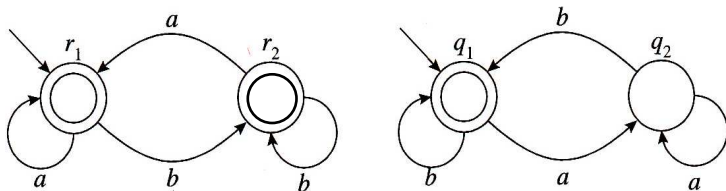
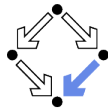


Given two finite automata  $A = \langle I_A, R_A, State_A \rangle$  and  $B = \langle I_B, R_B, F_B \rangle$ .

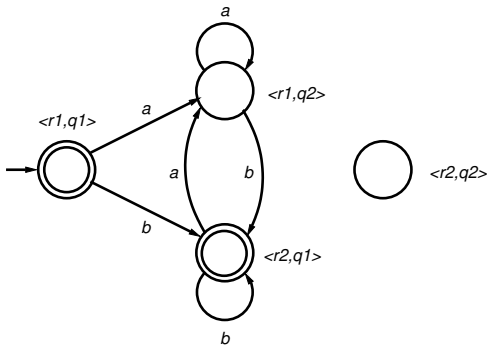
- **Synchronized product**  $A \otimes B = \langle I, R, F \rangle$ .
  - $State := State_A \times State_B$ .
  - $Label := Label_A = Label_B$ .
  - $I := I_A \times I_B$ .
  - $R(I, \langle s_A, s_B \rangle, \langle s'_A, s'_B \rangle) := \Leftrightarrow R_A(I, s_A, s'_A) \wedge R_B(I, s_B, s'_B)$ .
  - $F := State_A \times F_B$ .

Special case where all states of automaton  $A$  are accepting.

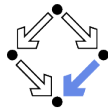
# Synchronized Product of Two Automata



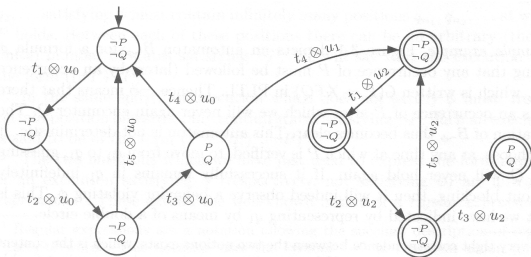
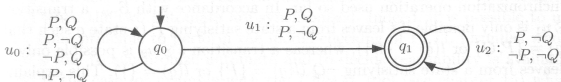
Edmund Clarke: "Model Checking", 1999.



# Example



Check whether  $S \models \Box(P \Rightarrow \bigcirc \Diamond Q)$ .

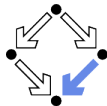


B. Berard et al: "Systems and Software Verification", 2001.

The product automaton accepts a run, thus the property does not hold (shown  $S$  has no  $\iota$  but  $t_i$  labeled with properties of predecessor state).



# Checking Emptiness

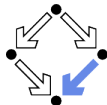


How to check whether  $\mathcal{L}(A)$  is non-empty?

- If  $\mathcal{L}(A)$  is non-empty,  $A$  accepts some run  $r$ .
  - $r$  represents a **counterexample** for the property to be checked.
- Since  $r$  is accepted, it contains infinitely many occurrences of some accepting state  $s$ .
$$r = \iota \rightarrow \dots \rightarrow s \rightarrow \dots \rightarrow s \rightarrow \dots \rightarrow s \rightarrow \dots$$
- Since the state space is finite,  $r$  must contain a cycle  $s \rightarrow \dots \rightarrow s$ .
  - Finite prefix  $\iota \rightarrow \dots \rightarrow s$ .
  - Infinite repetition of cycle  $s \rightarrow \dots \rightarrow s$ .
- We have to search for such an **acceptance cycle**.
  - An accepting state  $s$  that is reachable from itself.

The search for an acceptance cycle in the reachability graph is the core problem of PLTL model checking; it can be solved by *depth-first search*.

# Basic Structure of Depth-First Search



Visit all states of the reachability graph of an automaton  $\langle \{\iota\}, R, F \rangle$ .

**global**

*StateSpace*  $V := \{\}$

*Stack*  $D := \langle \rangle$

**proc** *main*()

*push*( $D, \iota$ )

*visit*( $\iota$ )

*pop*( $D$ )

**end**

**proc** *visit*( $s$ )

$V := V \cup \{s\}$

**for**  $\langle l, s, s' \rangle \in R$  **do**

**if**  $s' \notin V$

*push*( $D, s'$ )

*visit*( $s'$ )

*pop*( $D$ )

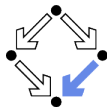
**end**

**end**

**end**

State space  $V$  holds all states visited so far; stack  $D$  holds path from initial state to currently visited state.

# Searching for a State



Algorithm to check violation of assertion  $f$  (i.e.,  $\Box f$  with state formula  $f$ ).

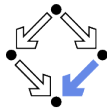
```
global
  StateSpace  $V := \{\}$ 
  Stack  $D := \langle \rangle$ 

proc main()
  //  $r$  becomes true, iff
  // counterexample run is found
  push( $D, \iota$ )
   $r := \text{search}(\iota)$ 
  pop( $D$ )
end

function search( $s$ )
   $V := V \cup \{s\}$ 
  if  $s \not\models f$  then
    print  $D$ 
    return true
  end
  for  $\langle l, s, s' \rangle \in R$  do
    if  $s' \notin V$ 
      push( $D, s'$ )
       $r := \text{search}(s')$ 
      pop( $D$ )
      if  $r$  then return true end
    end
  end
  return false
end
```

Stack  $D$  can be used to print counterexample run.

# Searching for an Acceptance Cycle



Algorithm to check whether a general temporal formula is violated.

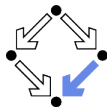
```
global
  StateSpace  $V := \{\}$ 
  Stack  $D := \langle \rangle$ 
  Stack  $C := \langle \rangle$ 

function searchCycle( $s$ )
  for  $\langle l, s, s' \rangle \in R$  do
    if has( $D, s'$ ) then
      print  $D$ ; print  $C$ ; print  $s'$ 
      return true
    else if  $\neg$ has( $C, s'$ ) then
      push( $C, s'$ );
       $r :=$  searchCycle( $s'$ )
      pop( $C$ );
      if  $r$  then return true end
    end
  end
  return false
end

proc main()
  push( $D, \iota$ );  $r :=$  search( $\iota$ ); pop( $D$ )
end

boolean search( $s$ )
   $V := V \cup \{s\}$ 
  for  $\langle l, s, s' \rangle \in R$  do
    if  $s' \notin V$ 
      push( $D, s'$ )
       $r :=$  search( $s'$ )
      pop( $D$ )
      if  $r$  then return true end
    end
  end
  if  $s \in F$  then
     $r :=$  searchCycle( $s$ )
    if  $r$  then return true end
  end
  return false
end
```

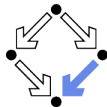
# Depth-First Search for Acceptance Cycle



- At each call of  $search(s)$ ,
  - $s$  is a reachable state,
  - $D$  describes a path from  $\iota$  to  $s$ .
- $search$  calls  $searchCycle(s)$ 
  - on a reachable accepting state  $s$
  - in order to find a cycle from  $s$  to itself.
- At each call of  $searchCycle(s)$ ,
  - $s$  is a state reachable from a reachable accepting state  $s_a$ ,
  - $D$  describes a path from  $\iota$  to  $s_a$ ,
  - $D \rightarrow C$  describes a path from  $\iota$  to  $s$  (via  $s_a$ ).
- Thus we have found an accepting cycle  $D \rightarrow C \rightarrow s'$ , if
  - there is a transition  $s \xrightarrow{l} s'$ ,
  - such that  $s'$  is contained in  $D$ .

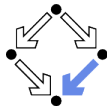
If the algorithm returns “true”, there exists a violating run; the converse follows from the exhaustiveness of the search.

# Implementing the Search



- The **state space  $V$** ,
  - is implemented by a hash table for efficiently checking  $s' \notin V$ .
- Rather than using explicit **stacks  $D$  and  $C$** ,
  - each state node has two bits  $d$  and  $c$ ,
  - $d$  is set to denote that the state is in stack  $D$ ,
  - $c$  is set to denote that the state is in stack  $C$ .
- The **counterexample** is printed,
  - by searching, starting with  $\iota$ , the unique sequence of reachable nodes where  $d$  is set until the accepting node  $s_a$  is found, and
  - by searching, starting with a successor of  $s_a$ , the unique sequence of reachable nodes where  $c$  is set until the cycle is detected.
- Furthermore, it is **not necessary to reset the  $c$  bits**, because
  - *search* first explores all states reachable by an accepting state  $s$  **before** trying to find a cycle from  $s$ ; from this, one can show that
  - called with the first accepting node  $s$  that is reachable from itself, *search2* will not encounter nodes with  $c$  bits set in previous searches.
  - **With this improvement, every state is only visited twice.**

# Complexity of the Search

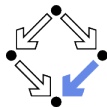


The complexity of checking  $S \models P$  is as follows.

- Let  $|P|$  denote the **number of subformulas of  $P$** .
- $|State_{(\neg P)_A}| = O(2^{|P|})$ .
- $|State_{A \otimes B}| = |State_A| \cdot |State_B|$ .
- $|State_{S_A \otimes (\neg P)_A}| = O(|State_{S_A}| \cdot 2^{|P|})$
- The time complexity of *search* is linear in the size of *State*.
  - Actually, in the number of **reachable states** (typically much smaller).
  - Only true for the improved variant where the  $c$  bits are **not reset**.
  - Then every state is visited at most **twice**.

**PLTL model checking is linear in the number of reachable states but exponential in the size of the formula.**

# The Overall Process



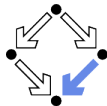
Basic PLTL model checking for deciding  $S \models P$ .

- Convert system  $S$  to automaton  $S_A$ .
  - Atomic propositions of PLTL formula are evaluated on each state.
- Convert negation of PLTL formula  $P$  to automaton  $(\neg P)_A$ .
  - How to do so, we have not yet described.
- Construct synchronized product automaton  $S_A \otimes (\neg P)_A$ .
  - After that, formula labels are not needed any more.
- Find acceptance cycle in reachability-graph of product automaton.
  - A purely graph-theoretical problem that can be efficiently solved.
  - Time complexity is linear in the size of the state space of the system but exponential in the size of the formula to be checked.
  - Weak scheduling fairness with  $k$  components: runtime is increased by factor  $k + 2$  (worst-case, “in practice just factor 2” [Holzmann]).

The basic approach immediately leads to *state space explosion*; further improvements are needed to make it practical.



# On the Fly Model Checking

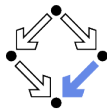


For checking  $\mathcal{L}(S_A \otimes (\neg P)_A) = \emptyset$ , it is not necessary to construct the states of  $S_A$  in advance.

- Only the property automaton  $(\neg P)_A$  is constructed in advance.
  - This automaton has comparatively small state space.
- The system automaton  $S_A$  is constructed **on the fly**.
  - Construction is guided by  $(\neg P)_A$  while computing  $S_A \otimes (\neg P)_A$ .
  - Only that part of the reachability graph of  $S_A$  is expanded that is consistent with  $(\neg P)_A$  (i.e. can lead to a counterexample run).
- Typically only a part of the state space of  $S_A$  is investigated.
  - A smaller part, if a counterexample run is detected early.
  - A larger part, if no counterexample run is detected.

**Unreachable system states and system states that are not along possible counterexample runs are never constructed.**

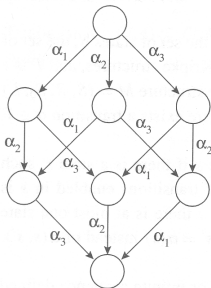
# Partial Order Reduction



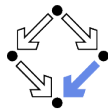
Core problem of model checking: state space explosion.

- Take **asynchronous composition**  $S_0 || S_1 || \dots || S_{k-1}$ .
  - Take state  $s$  where one transition of each component is enabled.
    - Assume that the transition of one component does not disable the transitions of the other components and that no other transition becomes enabled before all the transitions have been performed.
  - Take state  $s'$  after execution of all the transitions.
    - There are  $k!$  paths leading from  $s$  to  $s'$ .
    - There are  $2^k$  states involved in the transitions.

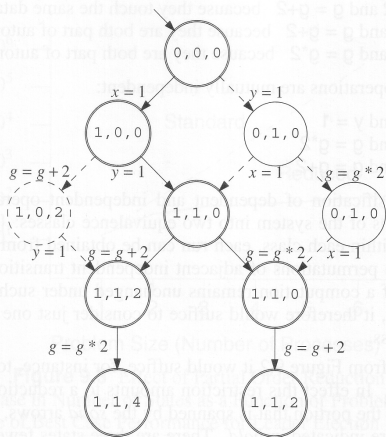
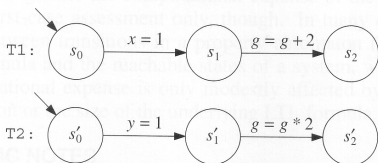
Sometimes it suffices to consider  
a *single path* with  $k + 1$  states.



# Example



Check  $(T1 \parallel T2) \models \diamond g \geq 2$ .



Gerard Holzmann: "The Spin Model Checker", 1999.

For checking  $\diamond g \geq 2$ , it suffices to check only one ordering of the independent transitions  $x = 1$  and  $y = 1$  (not true for checking  $\square x \geq y$ ).

# Example

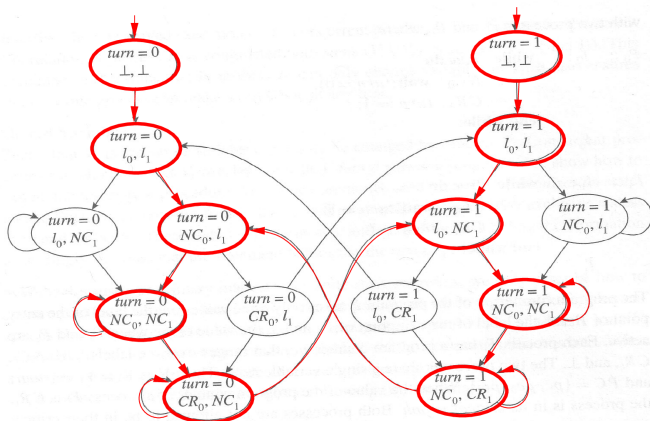
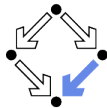
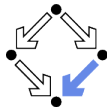


Figure 2.2  
Reachable states of Kripke structure for mutual exclusion example.

Edmund Clarke et al: "Model Checking", 1999.

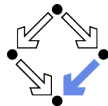
System after partial order reduction.

# Other Optimizations



- **Statement merging.**
  - Special case of partial order reduction where a sequence of transitions of same component is combined to a single transition.
- **State compression.**
  - **Collapse compression:** each state holds pointers to component states; thus component states can be shared among many system states.
  - **Minimized automaton representation:** represent state set  $V$  not by hash table but by finite state automaton that accepts a state (sequence of bits)  $s$  if and only if  $s \in V$ .
  - **Hash compact:** store in the hash table a hash value of the state (computed by a different hash function). Probabilistic approach: fails if two states are mapped to the same hash value.
  - **Bitstate hashing:** represent  $V$  by a bit table whose size is much larger than the expected number of states; each state is then only represented by a single bit. Probabilistic approach: fails if two states are hashed to the same position in the table.

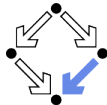
# Other Approaches to Model Checking



There are fundamentally different approaches to model checking than the automata-based one implemented in Spin.

- **Symbolic Model Checking** (e.g. SMV, NuSMV).
  - Core: **binary decision diagrams (BDDs)**.
    - Data structures to represent boolean functions.
    - Can be used to describe state sets and transition relations.
  - The set of states satisfying a CTL formula  $P$  is computed as the BDD representation of a fixpoint of a function (predicate transformer)  $F_P$ .
    - If all initial system states are in this set,  $P$  is a system property.
  - **BDD packages** for efficiently performing the required operations.
- **Bounded Model Checking** (e.g. NuSMV2).
  - Core: **propositional satisfiability**.
    - Is there a truth assignment that makes propositional formula true?
  - There is a counterexample of length at most  $k$  to a LTL formula  $P$ , if and only if a particular propositional formula  $F_{k,P}$  is satisfiable.
    - Problem: find suitable bound  $k$  that makes method complete.
  - **SAT solvers** for efficiently deciding propositional satisfiability.

# Other Approaches to Model Checking



- **Counter-Example Guided Abstraction Refinement** (e.g. BLAST).
  - Core: **model abstraction**.
    - A finite set of predicates is chosen and an abstract model of the system is constructed as a finite automaton whose states represent truth assignments of the chosen predicates.
  - The abstract model is checked for the desired property.
    - If the abstract model is error-free, the system is correct; otherwise an abstract counterexample is produced.
    - It is checked whether the abstract counterexample corresponds to a real counterexample; if yes, the system is not correct.
    - If not, the chosen set of predicates contains too little information to verify or falsify the program; new predicates are added to the set. Then the process is repeated.
  - **Core problem**: how to refine the abstraction.
    - Automated theorem provers are applied here.

Many model checkers for software verification use this approach.