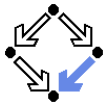


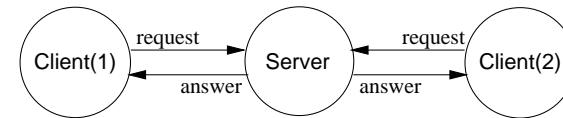
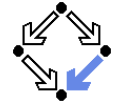
# Verifying Concurrent Systems with the Model Checker Spin

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.uni-linz.ac.at

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



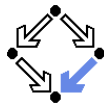
# A Client/Server System



- System of one server and two clients.
  - Three **concurrently** executing system components.
- Server manages a resource.
  - An object that only one system component may use at any time.
- Clients request resource and, having received an answer, use it.
  - Server ensures that not both clients use resource simultaneously.
  - Server eventually answers every request.

Set of system requirements.

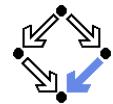
# System Implementation



```
Server:
local given, waiting, sender
begin
given := 0; waiting := 0
loop
sender := receiveRequest()
if sender = given then
if waiting = 0 then
given := 0
else
given := waiting; waiting := 0
sendAnswer(given)
endif
elseif given = 0 then
given := sender
sendAnswer(given)
else
waiting := sender
endif
endloop
end Server

Client(ident):
param ident
begin
loop
...
sendRequest()
receiveAnswer()
... // critical region
sendRequest()
endloop
end Client
```

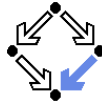
# Desired System Properties



- Property: **mutual exclusion**.
  - At no time, both clients are in critical region.
    - Critical region: program region after receiving resource from server and before returning resource to server.
  - The system shall only reach states, in which mutual exclusion holds.
- Property: **no starvation**.
  - Always when a client requests the resource, it eventually receives it.
  - Always when the system reaches a state, in which a client has requested a resource, it shall later reach a state, in which the client receives the resource.
- Problem: each system component executes its own program.
  - Multiple program states exist at each moment in time.
  - Total system state is **combination of individual program states**.
  - Not easy to see which system states are possible.

How can we verify that the system has the desired properties?

## System States

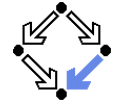


At each moment in time, a system is in a particular state.

- A **state**  $s : Var \rightarrow Val$ 
  - A state  $s$  is a mapping of every system variable  $x$  to its value  $s(x)$ .
    - Typical notation:  $s = [x = 0, y = 1, \dots] = [0, 1, \dots]$ .
  - $Var$  ... the set of system variables
    - Program variables, program counters, ...
  - $Val$  ... the set of variable values.
- The **state space**  $State = \{s \mid s : Var \rightarrow Val\}$ 
  - The state space is the set of possible states.
    - The system variables can be viewed as the coordinates of this space.
  - The state space may (or may not) be finite.
    - If  $|Var| = n$  and  $|Val| = m$ , then  $|State| = m^n$ .
    - A word of  $\log_2 m^n$  bits can represent every state.

A system execution can be described by a path  $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$  in the state space.

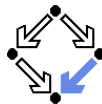
## Deterministic Systems



In a sequential system, each state typically determines its successor state.

- The system is **deterministic**.
  - We have a (possibly not total) **transition function**  $F$  on states.
    - $s_1 = F(s_0)$  means “ $s_1$  is the successor of  $s_0$ ”.
  - Given an initial state  $s_0$ , the execution is thus determined.
    - $s_0 \rightarrow s_1 = F(s_0) \rightarrow s_2 = F(s_1) \rightarrow \dots$
  - A **deterministic system (model)** is a pair  $\langle I, F \rangle$ .
    - A set of initial states  $I \subseteq State$ 
      - **Initial state condition**  $I(s) :\Leftrightarrow s \in I$
    - A transition function  $F : State \xrightarrow{\text{partial}} State$ .
  - A **run** of a deterministic system  $\langle I, F \rangle$  is a (finite or infinite) sequence  $s_0 \rightarrow s_1 \rightarrow \dots$  of states such that
    - $s_0 \in I$  (respectively  $I(s_0)$ ).
    - $s_{i+1} = F(s_i)$  (for all sequence indices  $i$ )
    - If  $s$  ends in a state  $s_n$ , then  $F$  is not defined on  $s_n$ .

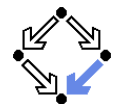
## Nondeterministic Systems



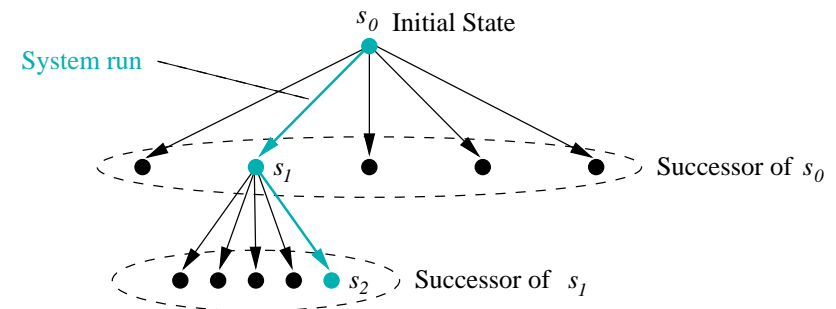
In a concurrent system, each component may change its local state, thus the successor state is not uniquely determined.

- The system is **nondeterministic**.
  - We have a **transition relation**  $R$  on states.
    - $R(s_0, s_1)$  means “ $s_1$  is a (possible) successor of  $s_0$ ”.
  - Given an initial state  $s_0$ , the execution is not uniquely determined.
    - Both  $s_0 \rightarrow s_1 \rightarrow \dots$  and  $s_0 \rightarrow s'_1 \rightarrow \dots$  are possible.
  - A **non-deterministic system (model)** is a pair  $\langle I, R \rangle$ .
    - A set of initial states (initial state condition)  $I \subseteq State$ .
    - A transition relation  $R \subseteq State \times State$ .
  - A **run**  $s$  of a nondeterministic system  $\langle I, R \rangle$  is a (finite or infinite) sequence  $s_0 \rightarrow s_1 \rightarrow s_2 \dots$  of states such that
    - $s_0 \in I$  (respectively  $I(s_0)$ ).
    - $R(s_i, s_{i+1})$  (for all sequence indices  $i$ ).
    - If  $s$  ends in a state  $s_n$ , then there is no state  $t$  such that  $R(s_n, t)$ .

## Reachability Graph

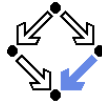


The transitions of a system can be visualized by a graph.



The nodes of the graph are the reachable states of the system.

# Concurrent Systems



Asynchronous composition of system components.

```

P :: l_0 : while true do
    NC_0 : wait turn = 0
    CR_0 : turn := 1
end
||
Q :: l_1 : while true do
    NC_1 : wait turn = 1
    CR_1 : turn := 0
end
    
```

■ A mutual exclusion program  $M = \langle I_M, R_M \rangle$ .

```

State := PC × PC × ℕ₂. // shared variable
I_M(p, q, turn) := p = l_0 ∧ q = l_1.
R_M((p, q, turn), (p', q', turn')) :=
    (P((p, turn), (p', turn')) ∧ q' = q) ∨ (Q((q, turn), (q', turn')) ∧ p' = p).
P((p, turn), (p', turn')) :=
    (p = l_0 ∧ p' = NC_0 ∧ turn' = turn) ∨
    (p = NC_0 ∧ p' = CR_0 ∧ turn = 0 ∧ turn' = turn) ∨
    (p = CR_0 ∧ p' = l_0 ∧ turn' = 1).
Q((q, turn), (q', turn')) :=
    (q = l_1 ∧ q' = NC_1 ∧ turn' = turn) ∨
    (q = NC_1 ∧ q' = CR_1 ∧ turn = 1 ∧ turn' = turn) ∨
    (q = CR_1 ∧ q' = l_1 ∧ turn' = 0).
    
```

# Concurrent Systems

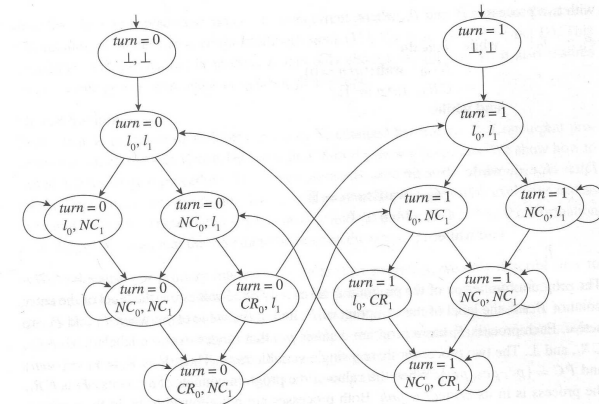
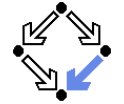
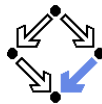


Figure 2.2 Reachable states of Kripke structure for mutual exclusion example.

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

Model guarantees mutual exclusion.

# Linear Time Logic (LTL)

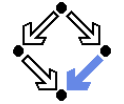


We use temporal logic to specify a system property  $P$ .

- **Core question:**  $S \models P$  ("P holds in system S").
  - System  $S = \langle I, R \rangle$ , temporal logic formula  $P$ .
- **Linear time logic:**
  - $S \models P := r \models P$ , for every run  $r$  of  $S$ .
  - Property  $P$  must be evaluated on every run  $r$  of  $S$ .
  - Given a computation tree with root  $s_0$ ,  $P$  is evaluated on every path of that tree originating in  $s_0$ .
    - If  $P$  holds for every path,  $P$  holds on  $S$ .

LTL formulas are evaluated on system runs.

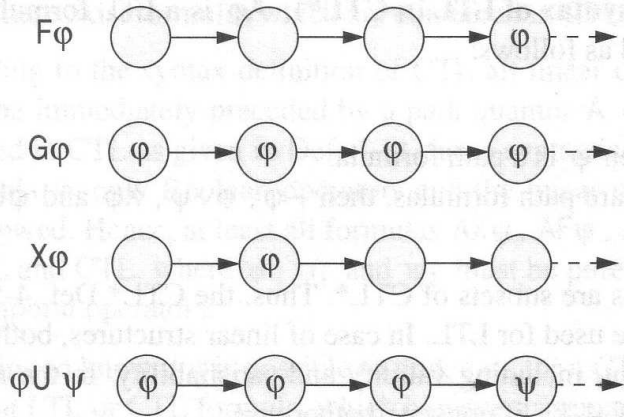
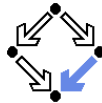
# Formulas



All formulas are path formulas.

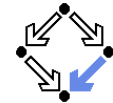
- Every formula is evaluated on a path  $p$ .
  - Also every state formula  $f$  of classical logic (see below).
  - Let  $F$  and  $G$  denote formulas.
  - Then also the following are formulas:
    - $\mathbf{X} F$  ("next time F"), often written  $\bigcirc F$ ,
    - $\mathbf{G} F$  ("always F"), often written  $\square F$ ,
    - $\mathbf{F} F$  ("eventually F"), often written  $\diamond F$ ,
    - $\mathbf{F} \mathbf{U} G$  ("F until G").
- **Semantics:**  $p \models P$  ("P holds in path p").
  - $p^i := \langle p_i, p_{i+1}, \dots \rangle$ .
  - $p \models f := p_0 \models f$ .
  - $p \models \mathbf{X} F := p^1 \models F$ .
  - $p \models \mathbf{G} F := \forall i \in \mathbb{N} : p^i \models F$ .
  - $p \models \mathbf{F} F := \exists i \in \mathbb{N} : p^i \models F$ .
  - $p \models \mathbf{F} \mathbf{U} G := \exists i \in \mathbb{N} : p^i \models G \wedge \forall j \in \mathbb{N}_i : p^j \models F$ .

## Formulas



Thomas Kropf: "Introduction to Formal Hardware Verification", 1999.

## Frequently Used LTL Patterns

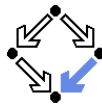


In practice, most temporal formulas are instances of particular patterns.

Pattern	Pronounced	Name
$\Box F$	always $F$	invariance
$\Diamond F$	eventually $F$	guarantee
$\Box \Diamond F$	$F$ holds infinitely often	recurrence
$\Diamond \Box F$	eventually $F$ holds permanently	stability
$\Box(F \Rightarrow \Diamond G)$	always, if $F$ holds, then eventually $G$ holds	response
$\Box(F \Rightarrow (G \mathbf{U} H))$	always, if $F$ holds, then $G$ holds until $H$ holds	precedence

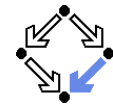
Typically, there are at most two levels of nesting of temporal operators.

## Examples



- Mutual exclusion:**  $\Box \neg (pc_1 = C \wedge pc_2 = C)$ .
  - Alternatively:  $\neg \Diamond (pc_1 = C \wedge pc_2 = C)$ .
  - Never both components are simultaneously in the critical region.
- No starvation:**  $\forall i : \Box (pc_i = W \Rightarrow \Diamond pc_i = R)$ .
  - Always, if component  $i$  waits for a response, it eventually receives it.
- No deadlock:**  $\Box \neg \forall i : pc_i = W$ .
  - Never all components are simultaneously in a wait state  $W$ .
- Precedence:**  $\forall i : \Box (pc_i \neq C \Rightarrow (pc_i \neq C \mathbf{U} lock = i))$ .
  - Always, if component  $i$  is out of the critical region, it stays out until it receives the shared lock variable (which it eventually does).
- Partial correctness:**  $\Box (pc = L \Rightarrow C)$ .
  - Always if the program reaches line  $L$ , the condition  $C$  holds.
- Termination:**  $\forall i : \Diamond (pc_i = T)$ .
  - Every component eventually terminates.

## The Model Checker Spin

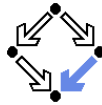


- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Workshop series since 1995 (12th workshop "Spin 2005").
  - ACM System Software Award in 2001.
- Spin resources:
  - Web site: <http://spinroot.com>.
  - Survey paper: Holzmann "The Model Checker Spin", 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

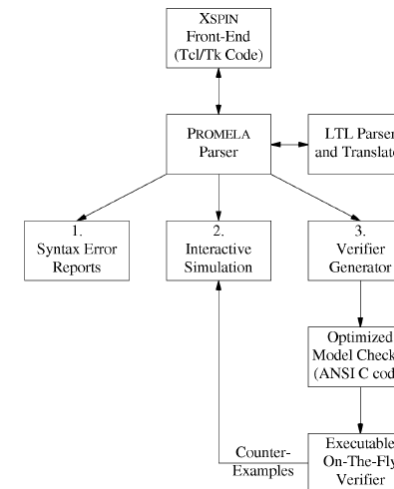
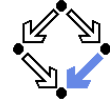
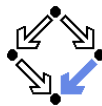


Fig. 1. The structure of SPIN simulation and verification.

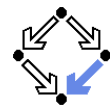
## Features of Spin



- 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 XSpin 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 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 };

/* two global arrays for monitoring
   the states of the clients */
bool inC[2] = false;
bool wait[2] = false;

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

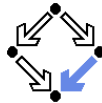
/* the client process type */
proctype client(byte id)
{
    do :: true ->
        request[id-1] ! MESSAGE;

        wait[id-1] = true;
        answer[id-1] ? MESSAGE;
        wait[id-1] = false;

        inC[id-1] = true;
        skip; // the critical region
        inC[id-1] = false;

        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 ->

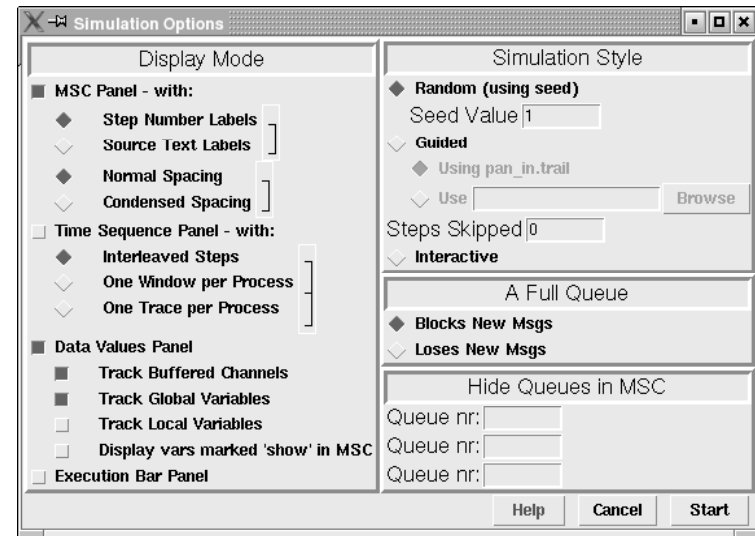
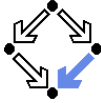
    /* receiving the message */
    if
    :: request[0] ? MESSAGE ->
      sender = 1
    :: request[1] ? MESSAGE ->
      sender = 2
    fi;

    /* 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;

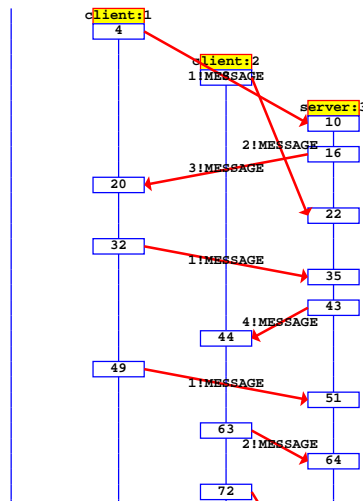
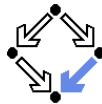
  od;
}

```

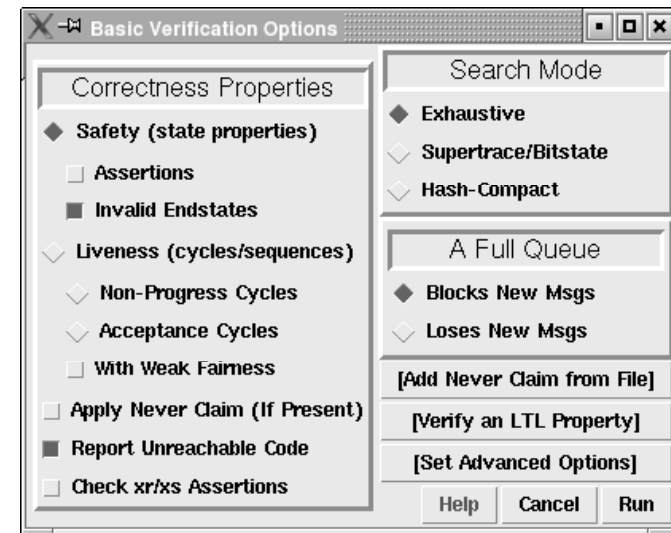
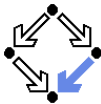
## Spin Simulation Options



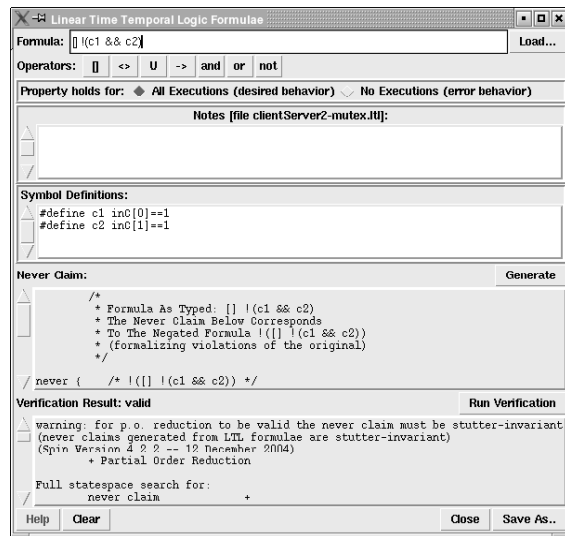
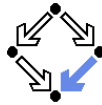
## Simulating the System Execution in Spin



## Spin Verification Options



## Specifying a System Property in Spin

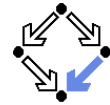


Wolfgang Schreiner

<http://www.risc.uni-linz.ac.at>

25/27

## Spin Verification Output



(Spin Version 4.2.2 -- 12 December 2004)  
+ Partial Order Reduction

Full statespace search for:

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

```
State-vector 48 byte, depth reached 477, errors: 0
499 states, stored
395 states, matched
894 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
```

Stats on memory usage (in Megabytes):

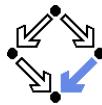
```
...
0.00user 0.01system 0:00.01elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+737minor)pagefaults 0swaps
```

Wolfgang Schreiner

<http://www.risc.uni-linz.ac.at>

26/27

## Summary



- Concurrent systems can be modelled as graphs:
  - Concurrency replaced by non-determinism.
  - Graph nodes represent system states, edges represent transitions.
  - Paths in graph describe possible runs of the system.
- Properties of such graphs may be specified by LTL formulas:
  - Formula must hold on every path in graph originating in root.
  - Formulas must hold for every possible system run.
- If graph is finite, property can be automatically verified.
  - Finite (approximation of) system state space.
  - E.g. by the model checker SPIN.

Verify correctness of concurrent systems, communication protocols, security protocols, etc.

Wolfgang Schreiner

<http://www.risc.uni-linz.ac.at>

27/27